Service-oriented Distributed Applications in the Future Internet: The Case for Interaction Paradigm Interoperability

Nikolaos Georgantas, Georgios Bouloukakis, Sandrine Beauche, Valérie Issarny

Service-oriented Distributed Applications in the Future Internet: The Case for Interaction Paradigm Interoperability

Nikolaos Georgantas[1], Georgios Bouloukakis[1], Sandrine Beauche[2], Valérie Issarny[1]

[1] Inria Paris-Rocquencourt, France
firstname.lastname@inria.fr
[2] Ambientic, France
sandrine.beauche@ambientic.com

Abstract. The essential issue of interoperability in distributed systems is becoming even more pressing in the Future Internet, where complex applications will be composed from extremely heterogeneous systems. Open system integration paradigms, such as service oriented architecture (SOA) and enterprise service bus (ESB), have provided answers to the interoperability requirement. However, when it comes to integrating systems featuring heterogeneous interaction paradigms, such as client-service, publish-subscribe and tuple space, existing solutions are typically ad hoc and partial, applying to specific interaction protocol technologies. In this paper, we introduce an interoperability solution based on abstraction and merging of the common high-level semantics of interaction paradigms, which is sufficiently general and extensible to accommodate many different protocol technologies. We apply this solution to revisit the SOA- and ESB-based integration of heterogeneous distributed systems.

Key words: Interoperability, interaction paradigms, interaction abstractions, service oriented architecture, enterprise service bus.

1 Introduction

The Future Internet (FI) is emerging as, among others, a global application space where People, Services and Things will be always-connected and interact in numerous ways. Accordingly, complex distributed applications in the FI will be based, to a large extent, on the open integration of extremely heterogeneous systems, such as lightweight embedded systems (e.g., sensors, actuators and networks of them), mobile systems (e.g., smartphone applications), and resource-rich IT systems (e.g., systems hosted on enterprise servers and Cloud infrastructures). These heterogeneous system domains differ significantly in terms of interaction paradigms, communication protocols, and data representation models, which are most often provided by supporting middleware platforms. In particular with regard to middleware-supported interaction, the client-service (CS), publish-subscribe (PS), and tuple space (TS) paradigms are among the most

widely employed ones, with numerous related middleware platforms, such as: Web Services, Java RMI for CS; JMS, SIENA for PS [1, 2]; and JavaSpaces, Lime for TS [3, 4]. In the following, we outline a representative application scenario, where a complex distributed application needs to be devised by integrating heterogeneous networked systems that interact with differing interaction paradigms.

Search and Rescue (S&R) operations after a disaster, such as a flood or earthquake, are carried out in hazardous environments and require personnel from multiple agencies (e.g., fire-fighters, police) to coordinate. To detect survivors, sensors are installed at various places of the hazardous area. Such sensors communicate their location. S&R personnel also notify at short intervals of their current positions via their PDAs. Upon sensing some life sign, sensor nodes send out notifications. At the same time, nearby light-emitting actuators start lighting the place to facilitate the rescuing effort. Sensors, PDAs, and actuators interact among them and with external actors via a TS. TS location and life sign data are sent via CS invocations to a planning service that recommends at real time the optimal deployment of rescue forces. This output is notified via a PS system to the coordinator of the operation on her smartphone and also to a number of control/monitoring centers. The coordinator may approve and command S&R personnel via the PS system and the TS system to rush into the spot.

To enable such a scenario, the heterogeneity between the involved system domains needs to be tackled. Existing cross-domain interoperability efforts are based on, e.g., bridging communication protocols [5], wrapping systems behind standard technology interfaces [6], and providing common API abstractions [7–10]. In particular, such techniques have been applied by the two currently dominant system integration paradigms, that is, service oriented architecture (SOA) and enterprise service bus (ESB) [11]. Both SOA and ESB employ the CS paradigm. Certainly, there are extensions, such as event-driven SOA [11] or industrial-strength ESBs supporting the PS paradigm. Additionally, research efforts have proposed the TS paradigm as interaction substrate for Web services or for ESBs [9, 12]. Nevertheless, most of these cross-paradigm interoperability efforts are ad hoc and partial, applying to specific cases. On the other hand, interaction paradigms have been widely studied, with theoretical approaches providing them with formal semantics by relying on concurrency theory, process algebras and architectural connectors (e.g., see [13]). These approaches typically identify semantics for individual paradigms but not cross-paradigm semantics.

In this paper[3], we introduce a model-based system integration solution that can deal with diverse existing systems, focusing in particular on integrating their heterogeneous interaction paradigms. Our systematic approach is carried out in two stages. First, a middleware platform is abstracted under a corresponding interaction paradigm among the three base ones, i.e., CS, PS and TS. To this aim, we elicit a *connector model* for each paradigm, which comprehensively covers its essential semantics. Then, these three models are abstracted further into a single *generic application (GA) connector* model, which encompasses their

[3] This work has been partially supported by the European Union's Seventh Framework Programme FP7/2007-2013 under grant agreement number 257178 (project CHOReOS).

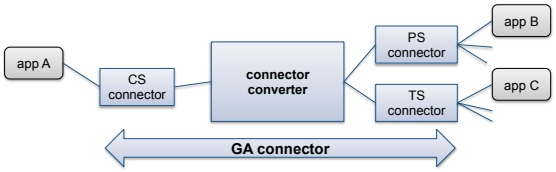

Fig. 1. GA-based connector interoperability

common interaction semantics. Based on GA, we build abstract *connector converters* that enable interconnecting the base interaction paradigms. A high-level representation of our approach is depicted in Figure 1. We realize our interoperability solution as an *extensible service bus (XSB)*, which is an abstract service bus that employs GA as its common bus protocol. Furthermore, we provide an implementation of the XSB, building upon existing SOA and ESB realizations. Based on our XSB platform, we propose a comprehensive solution to the peer-to-peer integration of services relying on heterogeneous interaction paradigms into complex applications. Our overall approach generalizes the way to design and implement service-oriented distributed applications, where the employed interaction paradigms are explicitly represented and systematically integrated. We demonstrate the applicability of our solution by implementing the scenario introduced above, and evaluate it in terms of extensibility and performance.

The rest of this paper is structured as follows. In Section 2, we introduce our connector models for abstracting and interconnecting interaction paradigms. In Section 3, we present the application of our models to the XSB solution, as well as its implementation. Then, in Section 4, we discuss the results of our evaluation. We finally complement this paper with a comparison of our approach with related work in Section 5, and conclude, also discussing future work, in Section 6.

2 Abstractions for Interaction Paradigm Interoperability

In this section, we identify the semantics of the three principal interaction paradigms, i.e., CS, PS and TS, and elicit a connector model for each paradigm (Section 2.1). Our modeling proposition is the outcome of an extensive survey of these paradigms as well as related middleware platforms in the literature. In a second step, we introduce our GA connector model, which enables cross-paradigm interoperability (Section 2.2). Before getting into the specifics of each connector, we briefly introduce in the following our global approach to connector modeling and point out the specific focus of this paper.

Our models represent the essential semantics of interaction paradigms, concerning *space coupling*, *time coupling* [14] and *concurrency*. Space coupling determines how peer applications interconnected via the connector identify each other and, consequently, how *interaction elements* (e.g., messages for a CS connector) are routed from one peer to the other. Time coupling determines if peers need to be present and available at the same time for an interaction or if the

interaction can take place in phases occurring at different times. Concurrency characterizes the exclusive or shared access semantics of the virtual channel established between interacting peers. These three categories of semantics are of primary importance, because these are end-to-end semantics: when interconnecting different connectors, we seek to map and preserve these semantics.

We represent interaction paradigm semantics in the connector's abstract *API (Application Programming Interface)*. This API presents the programming model supported by the connector and offered to the peer applications that use the connector for their interaction. The API is a set of *primitives* expressed as operations or functions supported by the middleware. This abstract API can be refined to a specific middleware platform by mapping to the primitives and incorporating the data structures and types of the middleware platform. Besides a connector's API, we introduce an abstract *interface description language (IDL)* for specifying the open interfaces of systems that rely on middleware represented by the specific connector. Our IDLs are largely inspired from WSDL. We specify the IDLs conceptually, while we have also implemented each one of them as an XML schema document. Based on the flexibility of XML schema, an IDL can be easily refined in order to enable the description of a concrete system that is based on the connector, e.g., we can refine the abstract XML elements into the precise data structures and types of the specific middleware and system.

Based on the informal identification of semantics as discussed in the previous, we further specify the connector's *formal behavioral semantics* in terms of LTS (Labeled Transition Systems). This formal behavior specification focuses on time coupling and concurrency semantics, while space coupling semantics is mainly represented by the connector's API and IDL. Additionally, we *formally verify* the correctness of these behavioral specifications with respect to time coupling and concurrency properties expressed in LTL temporal logic. This allows stating the correctness of our base connector models with respect to the semantics that they must have. This further enables identifying the semantics of the GA connector derived from the interconnection of base connectors.

The focus of this paper is the application of our connector modeling and analysis approach to the practical integration of heterogeneous services. Hence, and due to space limitations, we introduce in the following sections our connectors only informally – concentrating on their space coupling, time coupling and concurrency semantics – and mainly in terms of their respective IDLs, which are used to describe open interfaces of services.

2.1 Connector Models for Base Interaction Paradigms

This section introduces connector models for the CS, PS and TS paradigms.

Client-service connector. The CS connector model integrates a wide range of semantics, covering both the *non queue-based messaging* and *remote procedure call* paradigms. In terms of *space coupling* between two interacting peers, CS requires that the sender must hold a reference of the receiver. With respect to *time coupling*, both entities must be connected at the time of the interaction.

CS-based service interface			
element	**sub-element**	**attributes**	**S&R scenario - planning service**
message	data fields	semantics, name, type	{sensorId, sensorType, locationData, lifeSign}
main scope of message	service system identity	name, address type, address value	planningService
sub-scope of message	operation	semantics, name, type, value	planOperation
interaction semantics of message		{one-way, notification, request-response, solicit-response}	request

Fig. 2. CS IDL

Regarding *concurrency*, a dedicated virtual channel is used between a sender and a receiver: as long as servers do not have an excessive load of messages to process, all messages sent by different clients will be received by the designated servers. CS semantics is reflected on the CS-IDL presented in Figure 2, where the last column presents the example of the planning service of the S&R scenario (note that we provide an example for the underlined attributes of the third column). *Message* is the essential interaction element in CS-IDL; its interaction semantics is borrowed from WSDL. The main new concept here is that a message is assigned two qualifiers, *main scope* and *sub-scope*, which are, in inverse order, the operation served by the message and the URL of the service providing the operation. These qualifiers delimit the set of peer entities that will receive the message – actually only one service and, more finely, its specific operation.

Publish-subscribe connector. The PS connector model abstracts comprehensively different types of publish-subscribe systems, such as *queue-*, *topic-* and *content-based systems* [14]. In PS, multiple peers interact via an intermediate *broker*. Publishers produce events, which are received by peers that have previously subscribed for receiving the specific events. In terms of *space coupling*, interacting PS peers do not need to know each other; e.g., in topic-based systems, events are diffused to subscribers only based on the topic. With respect to *time coupling*, peers do not need to be present at the same time: subscribers may be disconnected at the time that events are published; they can receive the pending events when reconnected and before the events expire. Regarding *concurrency*, the broker maintains a dedicated buffer for each subscriber. Hence, all published non expired events will be eventually received by interested subscribers. We note that standardization of open PS interfaces (in the way SOA has done for CS systems) is far less developed. Hence, to introduce our PS-IDL (Fig. 3), we rely on our PS connector semantics, which has been extracted from a wide range of PS systems. The figure includes the example of the coordinator of the S&R scenario. The essential interaction element in PS-IDL is *event*; its interaction semantics denotes whether this event is published or received by the system in question and its lifetime, determined by *lease*. An event's *main scope* and *sub-scope* are the PS system URL and the *filter*, respectively, used for qualifying the event. *Filter* may represent a queue, topic or content. Similarly to CS, these qualifiers delimit the set of peers that will receive the event.

PS-based service interface			
element	**sub-element**	**attributes**	**S&R scenario - coordinator**
event	data fields	semantics, name, type	{personnelId, personnelType, locationData}
main scope of event	pub-sub system identity	name, address type, address value	SRcoordinationBroker
sub-scope of event	filter	semantics incl. {queue, topic, content}, name, type, value	topic, planningServiceInput
interaction semantics of event	produce/consume	{publish, subscribe}	subscribe
	lease	type, value	forever

Fig. 3. PS IDL

TS-based service interface			
element	**sub-element**	**attributes**	**S&R scenario - sensor**
tuple	data fields	semantics, name, type	{sensorId, sensorType, locationData, lifeSign}
main scope of tuple	tuple space system identity	name, address type, address value	SRdataSpace
sub-scope of tuple	extent	semantics, name, type, value	-
	template	semantics, name, type, value	sensorTemplate
interaction semantics of tuple	produce/consume	{write, take, read}	write
	consume policy	{one, all}	-
	lease	type, value	forever

Fig. 4. TS IDL

Tuple space connector. The *TS connector model* is based on the classic tuple space semantics [15]. In TS, multiple peers interact via an intermediate *shared data space*. Peers can post data into the space and retrieve data from it, either by taking a copy or removing the data. Data take the form of tuples; a tuple is an ordered list of typed elements. Data are retrieved by matching based on a tuple *template*, which may define values or expressions for some of the elements. Regarding *space coupling*, TS peers may write and read/take data from the space with no knowledge of each other. As for *time coupling* semantics, peers can act without any synchronization. With respect to *concurrency*, peers have access to a single, commonly shared copy of the data. Then, concurrent access semantics of the data space is non-deterministic: the order among accessing peers is determined arbitrarily. Hence, if a peer that intends to take specific data is given access to the space before other peers that are interested in the same data, the latter will never access these data. The TS-IDL is depicted in Figure 4, including the example of a sensor of the S&R scenario. Same as for PS systems, there are no standard open interfaces for TS systems, hence we rely on the generality of our TS connector. The essential interaction element in TS-IDL is *tuple*. Its interaction semantics denotes whether this tuple is produced or consumed by the system in question and its lifetime, determined by *lease*. In the case of tuple consumption, only *one* or *all* tuples matching a template may be retrieved. A tuple's *main scope* and *sub-scope* are the TS system URL and the pair {*extent*, *template*}, respectively, used for qualifying the tuple. *Extent* may be used to access only an identified part of the shared space. These qualifiers delimit the set of peer entities that will potentially receive the tuple.

2.2 Generic Application Connector Model

Given the three base connector models, we now introduce the *Generic Application (GA)* connector model. Our objective is to devise a single generic connector that comprehensively represents the end-to-end cross-paradigm interaction semantics of application peers that employ different base connectors.

We identify two main high-level API primitives for the GA connector: (i) *post* employed by a peer for sending data to one or more other peers, and (ii) *get* employed by a peer for receiving data. For example, a PS *publish* primitive can be abstracted by a *post*. We identify *space coupling* semantics for the GA connector by appropriately mapping among the space coupling semantics of the base connectors. Hence, we define the essential interaction element for GA to be *data*. *Data* can represent any one of CS *message*, PS *event* or TS *tuple*. Same as for the base connectors, GA uses the qualifiers *main scope* and *sub-scope* to characterize a data element. These qualifiers can represent the corresponding qualifiers of any of the CS, PS or TS. Hence, GA's {*main scope, sub-scope*} maps, for CS, to {*CS system identity, operation*}, for PS, it maps to {*PS system identity, filter*}, and for TS, to {*TS system identity,* {*extent, template*}}. In this way, GA generalizes and unifies addressing for the different interaction paradigms.

In order to identify the *time coupling* and *concurrency* semantics of GA and construct a converter among the base connectors (see Fig. 1), we have built upon the formal method of *protocol conversion via projections* [16]. According to this method, conversion between two different protocols is possible if both protocols can be projected (where projection is an abstraction defined as a set of transformations on the protocol LTS) to a *functionally sufficient* common *image protocol*. Then, the end-to-end protocol of the interconnection of the two protocols is this image protocol. However, this work is out of the scope of this paper. In the following, we present informally some of the outcomes of this work.

In the case of CS-PS-TS interconnection, GA is the common image protocol and represents the common time coupling and concurrency semantics. However, as shown in Section 2.1, time coupling and concurrency semantics of CS, PS, TS are not directly compatible. In particular, we saw that for successful interaction, for CS, the CS server must be online, for PS, a subscription is necessary, and for TS, all interested peers must be allowed to read the shared data before one of the peers takes them. This means that, in Fig. 1, *app A*, *B* and *C* may assume and perceive different semantics, which can be problematic for the composed application. The solution is to constrain the semantics of the heterogeneous connectors to a compatible subset *by application-side enforcement*. This means that if each one of *app A*, *B* and *C* enforces with its behavior the identified condition for successful interaction proper to its connector, common time coupling and concurrency semantics will apply to the end-to-end GA connector. In another example, a CS two-way interaction does not have an equivalent in the PS and TS connectors. In this case, the PS and TS applications should take care of enforcing the additional semantics. In general, CS is the more restrictive of the three paradigms, while PS and TS allow more flexibility to the applications; hence, the PS and TS applications should apply the missing semantics, if required by

GA-based service interface			
element	**sub-element**	**attribute**	**S&R scenario - coordinator**
data	data fields	semantics, name, type	{personnelId, personnelType, locationData}
main scope of data	system identity	name, address type, address value	SRcoordinationBroker
sub-scope of data	data qualifier(s)	semantics, name, type, value	topic, planningServiceInput
interaction semantics of data		{post, get, post-get, get-post}	get

Fig. 5. GA IDL

the CS application. While each case should be treated individually, we can state in general that in a CS-PS-TS interconnection, the resulting end-to-end GA semantics is the one of CS.

Based on the above and by mapping among the IDLs of the base connectors, we elicit the IDL for the GA connector as shown in Fig. 5. We can see that interaction semantics of *data* corresponds to the one of CS. The figure includes the example of the coordinator of the S&R scenario as mapped from PS-IDL (see Fig. 3) to GA-IDL. Concluding, we point out three important features of GA that result from the previous. First, although GA semantically intersects the CS, PS and TS paradigms, it represents *rich* interaction functionality, which means that interconnecting CS, PS and TS systems – under certain identified conditions – results in satisfactorily functional systems. Second, GA-IDL, which unifies the description of heterogeneous systems, is not heavier or more complex than the native CS/PS/TS-IDL descriptions. Third, GA applies at the middleware layer, and hence it allows *full expressivity* – only subject to the intersected end-to-end interaction semantics – of application-layer languages that specify the internal or external behavior of application components, such as WS-BPEL.

3 eXtensible Service Bus

We apply our connector models and resulting middleware interoperability method to an enhanced service bus paradigm, the *eXtensible Service Bus (XSB)*. XSB features richer interaction semantics than common ESBs to deal effectively with the increased Future Internet heterogeneity. Moreover, from its very conception, XSB incorporates special consideration for the cross-integration of heterogeneous interaction paradigms. In particular, XSB is an abstract bus that prescribes only the high-level semantics of the common bus protocol, which is the GA semantics. Services relying on different interaction paradigms can be plugged into XSB by employing *binding components (BCs)* that adapt between their native middleware and the common bus protocol. This adaptation is based on the abstractions discussed in Section 2, and in particular on the conversion between the native middleware, the corresponding CS/PS/TS abstraction, and the GA abstraction, as depicted in Figure 6. Hence, XSB BCs are half-converters in relation to Fig. 1.

XSB, being an abstract bus, can have different implementations. This means that it needs to be complemented with a *substrate bus* which supports deployment of services and a communication protocol that implements GA semantics.

This substrate bus may be designed and built from scratch or, alternatively, an existing one can be used, as long as GA primitives can be conveyed on top of the available protocol. The latter solution can be attractive, as it enables XSB realizations in different domains. We provide a generic *architectural framework* for XSB. This enables implementing XSB on top of a substrate bus of choice, and offers systematic support for building XSB BCs for different middleware platforms that apply one of the CS, PS, TS interaction paradigms. Furthermore, the framework can be extended with support for a new interaction paradigm. In the following, we present our architectural framework and the implementations we carried out by using this framework.

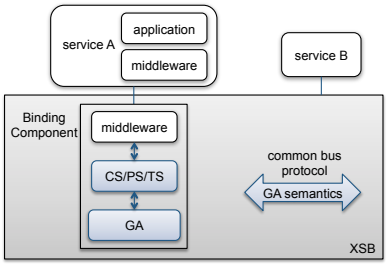

Fig. 6. eXtensible Service Bus.

Architectural framework. The architecture of an XSB BC as provided by our architectural framework is depicted in Fig. 7, where the main components are the *xDL Processor*, *Core Engine*, and *Envelope for Substrate Bus*. On its lower side, the BC communicates with the substrate bus, while on its upper side, it communicates with the middleware of the corresponding service by employing an instance of the same middleware, e.g., as an external library. In order to support extensibility, each component of an XSB BC is designed with three *architectural levels*: the first one is the most generic and can be refined stepwise into the two other levels, where refinement refers to class inheritance and XML schema transformation. The *generic level* provides APIs and functionalities that are shared among all supported interaction paradigms. The *interaction paradigm level* specializes the APIs and functionalities of the previous level for each one of the CS, PS and TS interaction paradigms. The *middleware platform level* specializes the APIs and functionalities of the previous level for a concrete middleware. In the following, we briefly sketch the main components of an XSB BC, and how a developer can make use of them.

The *xDL Processor* processes the descriptions of services deployed on the XSB. It performs both parsing of CS/PS/TS-IDL descriptions and mapping of them to GA-IDL descriptions, where the latter relies on XSLT-based transformations [17]. We use the XML schema extensibility mechanisms to specialize these functions from one architectural level to another. The *Core Engine* provides mechanisms to: (i) transform and map between service data and CS/PS/TS/GA

XML data; (ii) execute service primitives, and map between them and CS/PS/TS /GA primitives; and (iii) manage connections to the service middleware. The above mechanisms cooperate with each other, as well as with the xDL Processor for retrieving service information. The *Envelope for Substrate Bus* makes the BCs deployable on top of different substrate buses. It provides the mechanisms to: (i) communicate GA primitives over substrate bus connections, while exchanging them with the Core Engine; and (ii) manage the lifecycle of the service on the substrate bus, after retrieving service information from the xDL Processor. These mechanisms can be refined to support a new substrate bus.

Use by the developer. Targeting facilitated extensibility of our solution, we provide a highly-optimized design, where the common reusable part of the BC functionalities is already implemented by the different architectural levels, leaving to the developer the required specialization for introducing a new service, middleware platform, or interaction paradigm. More specifically, a developer wishing to deploy a new service on the XSB should write an xDL description of the service, and then invoke the tools provided by our solution to generate a corresponding BC deployable on the bus. A developer wishing to develop an XSB BC supporting a new middleware platform should refine the interaction paradigm levels of the xDL Processor and Core Engine. A developer wishing to support a new interaction paradigm should refine the generic levels of the xDL Processor and Core Engine.

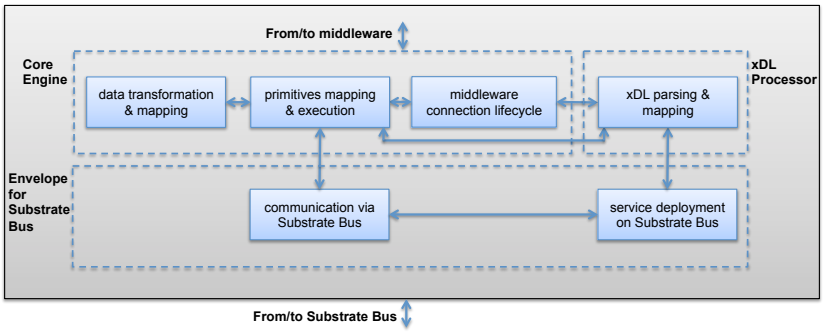

Fig. 7. Binding Component architecture.

Implementation. We have implemented XSB on top of the EasyESB[4] enterprise service bus, which is an open source lightweight service bus. In particular, we have refined our architectural framework to support building XSB BCs on top of EasyESB, and have provided interaction paradigm level BCs for CS, PS and TS. We demonstrate the applicability of our approach by implementing the S&R scenario. Our scenario implementation integrates: (1) sensors, actuators

[4] https://research.linagora.com/display/easyesb

and personnel equipment communicating over a Jini JavaSpaces TS[5]; (2) the planning service implemented as a JMEDS DPWS Web Service[6]; and (3) a JMS PS system based on Apache ActiveMQ[7] that the coordinator of the operation uses to receive recommendations and to send commands. We provide support for the three mentioned middleware platforms by producing appropriate middleware platform level BCs. Our XSB prototype implementation is available as open source software at `http://xsb.inria.fr`.

4 Evaluation

Based on the implementation of our solution discussed in Section 3, we evaluate our approach with respect to three criteria. First, we evaluate the effort for the application developer and accordingly the provided support by our solution for developing complex applications from the integration of services that employ heterogeneous interaction paradigms. Second, we have designed our architectural framework with particular consideration for its extensibility. Thus, we evaluate the easiness in integrating new middleware platforms, in particular with regard to building related binding components (BCs). Third, we have introduced a number of extensions to the typical SOA & ESB infrastructure, such as transfer of GA primitives as payload of ESB communication primitives, and, more importantly, runtime model transformations inside the BC. Hence, we evaluate the performance of our solution and the time overhead introduced. We discuss our evaluation results in the following.

Effort for application design. Table 1 summarizes our measurements of the development effort required for the S&R scenario. Essentially, this effort includes writing an xDL description for each constituent service, and providing mapping directives between the data exchanged among the services. GA-IDL service descriptions are then generated automatically by using the tools provided by our platform. We see that application development effort is considerably low, since our platform takes care of resolving the interaction paradigm and middleware heterogeneity among the constituent services.

	xDL description (XML lines)	Generated desc. (XML lines)	Mapping directives (XML lines)
Java Spaces system	148	98	72
DPWS system	50	61	76
JMS system	209	90	78
Total	407	249	226

Table 1. Development effort for the application developer

[5] http://www.jini.org/wiki/JavaSpaces_Specification
[6] http://ws4d.e-technik.uni-rostock.de/jmeds/
[7] http://activemq.apache.org/

Extensibility. Referring to the architectural framework of Fig. 7, we measure the effort for building a BC for the JMS Apache ActiveMQ middleware platform. Table 2 summarizes this effort, in terms of implemented numbers of: (1) Lines of code, (2) XML schema lines regarding the xDL descriptions, and (3) XML lines of configuration files for the architectural framework. We have performed our measurements with the Metrics 1.3.6 Eclipse plugin[8]. We provide measurements for each one of the three components of the framework, as well as the ratio of the effort specific to the JMS platform (refinement of subcomponents) over the total effort (i.e., including the generic code written once and reusable each time). We see that considerably small effort, no more than 6% of the total effort, is required for the integration of a new middleware platform. This points out the significant support offered, resulting in considerable easiness for integrating new middleware platforms and related high extensibility of our approach.

	Lines of code	XML schema (lines)	Configuration (XML lines)
xDL Processor	7520	2617	111
Core Engine	9993	219	137
Envelope for Substrate Bus	508	0	0
Total	18021	2836	248
Written by the developer	1162	191	12
Effort	6%	6%	4%

Table 2. Development effort for the JMS binding component

Performance. We measure execution times for a number of layouts: (i) one-way and two-way interaction inside our implemented CS system; (ii) end-to-end interaction between a publisher and a subscriber inside our implemented PS system; (iii) end-to-end interaction between a writer and a reader inside our implemented TS system; (iv) one-way and two-way interaction between two CS peers via EasyESB; and (v) interaction between all pair combinations of CS, PS and TS peers via XSB. We repeat each measurement a 100 times and calculate mean values. Based on these experiments, we evaluate the latency overhead introduced by the EasyESB for an one-way CS-CS communication, and the latency overhead introduced by the XSB for an one-way CS-CS as well as all other pair combinations of communication. Our results are summarized in Table 3. We see that the latency overhead introduced by the XSB for a CS-CS interconnection is only 1% greater than the latency overhead introduced by the EasyESB itself. When conversion between heterogeneous interaction paradigms is involved, the XSB latency overhead ranges from 7% to 15,5%, where we note that we always compare with the EasyESB CS-CS homogeneous interconnection, since EasyESB support for other interaction paradigms is not available. We see that the performance cost introduced by the XSB remains at reasonable levels.

[8] http://metrics.sourceforge.net

Interconnection	Latency (ms)
one-way CS - CS via EasyESB	258
one-way CS - CS via XSB	261,5
CS - PS via XSB	283
CS - TS via XSB	276
PS - TS via XSB	298

Table 3. Interaction latency on the bus for each interconnection

5 Related Work

Distributed system interoperability approaches at the middleware level are classically based on bridging communication protocols, wrapping systems behind standard technology interfaces, and providing common API abstractions. Most efforts focus on a single interaction paradigm, which is already a hard problem. Nevertheless, there are some solutions combining diverse interaction paradigms.

Common API abstractions enable developing applications that are agnostic to the underlying interaction paradigm. Then, some local mapping is performed between the API operations and the diverse interaction paradigms/related interaction protocols supported. In our previous work [18], we made a first attempt towards modeling the CS, PS and TS interaction paradigms. We also proposed a TS-based model as higher-level API abstraction for representing all three paradigms. Even if under certain conditions any of the three paradigms can be used as common abstraction, our introduction of GA in this paper makes things clearer and facilitates extension with new interaction paradigms. Additionally, that work was about heterogeneous service orchestrations, while our current work is more general and enables service choreographies. In the same category, ReMMoC [7] is an adaptive middleware for mobile systems, enabling clients that can interact with both RPC servers and PS systems via a common programming interface. Such systems are described with extended WSDL descriptions. Our solution is much more general: it covers as well TS systems and introduces the higher-level GA abstraction that can accommodate new interaction paradigms. Following a similar approach, an API conforming to one interaction paradigm can be locally mapped to an interaction protocol conforming to another paradigm. Thus in [8], the authors implement the LIME TS middleware on top of a PS substrate. Similarly, work in [9] enables Web services SOAP-based interactions over a TS binding. Contrary to these specific solutions, our approach aims to cover a much wider range of interaction paradigm interoperability.

Wrapping systems behind standard technology interfaces enables accessing these systems by using interaction paradigms that are different from their native ones. In [6], a gateway allows high-level access to the data and operations of a wireless sensor network via Web service interfaces. Again, our solution is much more general, relying on technology-independent abstractions.

Bridging is about interworking between heterogeneous interaction protocol stacks. The ESB paradigm is currently the dominant bridging solution for the integration of heterogeneous systems, with realizations that are established in-

dustrial (open- and closed-source) products, such as Apache ServiceMix[9] and IBM Websphere ESB[10]. Certain efforts have provided binding components (BCs) for ESBs that map between different interaction paradigms. For instance in [5], an external TS is connected through a BC to a distributed ESB topology and is accessible via the bus messaging-based interface. However, such solutions are typically ad hoc and concern each time a specific case, while we propose a generic and systematic approach that can be applied to many different middleware technologies. Other efforts propose extensions to SOA and ESB infrastructures, such as event-driven SOA [11], while now most industrial-strength ESBs support the PS paradigm. Still, these remain partial, they do not support the TS paradigm. Acknowledging the flexibility of the TS model, a number of system integration efforts have adopted TS as the common interaction facility. Some of these approaches enrich TS with PS semantics, or offer a REST-based API in addition to the TS-based API [19]. Similar efforts introduce extended TS as an alternative solution to the realization of the ESB paradigm [12]. Some of these ESBs offer various interaction semantics (by emulating different interaction paradigms) and related APIs, such as CS- and PS- in addition to TS-based. With respect to these efforts, the comparative advantage of our approach is its generality and extensibility thanks to the introduction of the higher-level GA abstraction.

6 Conclusion

Integrating services that employ heterogeneous interaction paradigms is challenging. We have introduced a modeling approach abstracting heterogeneous middleware platforms into their corresponding interaction paradigms, and the latter to a single higher-level interaction paradigm that enables cross-paradigm interconnection. We apply our modeling abstractions to extend an SOA & ESB infrastructure for supporting development of complex applications by seamless peer integration of heterogeneous services. A development platform is provided to application designers. Using this platform, they can easily develop composite applications: they only need to build descriptions for the constituent services and directives for data mapping among them. Our platform then deals with reconciling among the heterogeneous interaction paradigms and protocols of the services. Additionally, support for new middleware platforms, new ESB infrastructures, or even new interaction paradigms can be incorporated in a facilitated way thanks to our architectural framework. Our evaluation demonstrates the application design support, high extensibility, and low performance cost of our solution.

In our current and future work, besides publishing on the formal foundation of our interoperability approach, we aim to enrich our modeling abstractions with support for continuous interactions in addition to discrete ones. Continuous interactions are commonly found in data streaming protocols, which are increasingly important in the Future Internet, due to the vast spread of media content and sensor-generated data streams.

[9] http://servicemix.apache.org

[10] http://www.ibm.com/developerworks/websphere/zones/businessintegration/wesb.html

References

1. Monson-Haefel, R. and Chappell, D.: Java Message Service. O'Reilly & Associates, Inc. Sebastopol, CA, USA (2000)
2. Carzaniga, A., Wolf, A.: Content-based Networking: A New Communication Infrastructure. Lecture Notes in Computer Science (2002) 59–68
3. Freeman, E. and Arnold, K. and Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley Longman Ltd. Essex, UK, UK (1999)
4. Murphy, A.L. and Picco, G.P. and Roman, G.C.: LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. ACM Transactions on Software Engineering and Methodology (TOSEM) **15**(3) (2006) 328
5. Baude, F., Filali, I., Huet, F., Legrand, V., Mathias, E., Merle, P., Ruz, C., Krummenacher, R., Simperl, E., Hammerling, C., Lorre, J.P.: ESB Federation for Large-scale SOA. In: Proceedings of the 2010 ACM Symposium on Applied Computing. SAC '10, New York, NY, USA, ACM (2010) 2459–2466
6. Avilés-López, E., García-Macías, J.: TinySOA: A Service-oriented Architecture for Wireless Sensor Networks. Service Oriented Computing and Applications **3**(2) (2009) 99–108
7. Grace, P., Blair, G.S., Samuel, S.: A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments. SIGMOBILE Mob. Comput. Commun. Rev. **9**(1) (2005) 2–14
8. Ceriotti, M., Murphy, A.L., Picco, G.P.: Data Sharing vs. Message Passing: Synergy or Incompatibility?: An Implementation-driven Case Study. In: Proceedings of the 2008 ACM Symposium on Applied Computing, New York, USA (2008) 100–107
9. Wutke, D., Martin, D., Leymann, F.: Facilitating Complex Web Service Interactions through a Tuplespace Binding. In: Proceedings of the 8th IFIP International Conference on Distributed Applications and Interoperable Systems. (2008) 275–280
10. Pietzuch, P., Eyers, D., Kounev, S., Shand, B.: Towards a Common API for Publish/Subscribe. In: Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems, New York, USA, ACM (2007) 152–157
11. Papazoglou, M.P., Heuvel, W.J.: Service Oriented Architectures: Approaches, Technologies and Research Issues. The VLDB Journal **16** (July 2007) 389–415
12. Mordinyi, R., Kühn, E., Schatten, A.: Space-Based Architectures as Abstraction Layer for Distributed Business Applications. In: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems. CISIS '10, Washington, DC, USA, IEEE Computer Society (2010) 47–53
13. Busi, N., Zavattaro, G.: A Process Algebraic View of Shared Dataspace Coordination. The Journal of Logic and Algebraic Programming **75**(1) (2008) 52–85
14. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The Many Faces of Publish/Subscribe. ACM Comput. Surv. **35**(2) (2003) 114–131
15. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS) **7**(1) (1985) 80–112
16. Lam, S.S.: Protocol Conversion. IEEE Trans. Softw. Eng. **14**(3) (1988) 353–362
17. Kay, M.: XSLT 2.0 Programmer's Reference. Wiley Pub. (2004)
18. Georgantas, N., Rahaman, M., Ameziani, H., Pathak, A., Issarny, V.: A Coordination Middleware for Orchestrating Heterogeneous Distributed Systems. In Riekki, J., Ylianttila, M., Guo, M., eds.: Advances in Grid and Pervasive Computing. Volume 6646 of LNCS. Springer Berlin / Heidelberg (2011) 221–232
19. Nixon, L.j.b., Simperl, E., Krummenacher, R., Martin-Recuerda, F.: Tuplespace-based Computing for the Semantic Web: A Survey of the State-of-the-Art. Knowl. Eng. Rev. **23**(2) (2008) 181–212