Event Based Service Coordination over Dynamic and Heterogeneous Networks

Gianluigi Ferrari[1], Roberto Guanciale[2], and Daniele Strollo[1,2]

[1] Dipartimento di Informatica,
Università degli Studi di Pisa, Italy
{giangi, strollo}@di.unipi.it
[2] Istituto Alti Studi IMT Lucca, Italy
{roberto.guanciale, daniele.strollo}@imtlucca.it

Abstract. This paper describes the design and the prototype implementation of a programming middleware for coordinating services distributed over dynamic and heterogeneous networks without a public addressing schema (i.e. service addresses are not always public available). We illustrate the problems posed by relaxing the public addressing schema in the context of service orchestration. We discuss the design choices of our middleware. Then, we discuss the actual network technologies underlying the prototype implementation and the formal foundations that drive our approach.

1 Introduction

Modern distributed systems demand not only heterogeneity but also a higher degree of adaptability. The so called Service Oriented Architectures (SOAs) provide evidence of this issue. In the SOA approach, applications are developed by coordinating the behavior of autonomous components distributed over an overlay network. Several research and implementation efforts are currently devoted to design and to implement middleware for coordinating distributed services (see ORC [6], BPEL [8] and WS-CDL [10] to cite a few). These efforts have focused on overlay networks based on a public addressing schema, namely the *address* of each service is directly visible and reachable from any part of the network. Indeed, very few approaches address coordination of services over overlay networks where services reside on hosts without a public address or are hosted behind a firewall hiding their addresses. Other modern distributed systems raise similar demands with respect to the visibility of addresses. Illustrative examples are peer-to-peer networks. Coping with these issues is therefore a challenging task for the SOA paradigm.

This paper attempts to explore the features of the SOA approach within computing environments without a public addressing schema where visibility of service addresses is not always guaranteed. Our goal is twofold. First, we describe the design and the implementation of a programming middleware for service coordination where identification of services endpoints is more structured while preserving, at the same time, independence from the underlining network technologies. Second, we aim at developing a formal model that drive our implementation choices. In our approach, the primitives of the calculus represent the basic programming constructs supplied by the middleware.

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 453–458, 2006.

The starting point of our work is the event-notification paradigm where service behaviors are coordinated through the exchange of (typed) signals. This coordination model has been adopted for developing a middleware for service choreography called JSCL (Java Signal Core Layer [9]). To cope with the event notification paradigm within a formal setting, in [4] we introduced the Signal Calculus (SC). The SC calculus has driven the design and the prototype implementation of JSCL. The JSCL prototype has been used in [1] for implementing a framework for programming Long Running Transactions (LRTs) [2]. In this paper we extend the JSCL framework to deal with partial visibility of services. Services have no public addresses and their visibility over an overlay network is obtained through intermediate entities called *gateways*. Gateways are directly reachable from the network, and communication from services to gateways can be performed (e.g. by using SOAP). However, services can open a "channel" with several gateways by exploiting registration facilities, and can receive messages from other services through gateways. We discuss how JSCL has been extended to accommodate the new features. In particular, we investigate how the addressing of private services can be implemented by exploiting the SOAP binding proposed in 3.1, preserving the ability to support different coexisting bindings. Then we provide the formal semantic characterization for the operations that SC makes available to handle the new model.

2 Java Signal Core Layer

JSCL is a Java middleware to implement the choreography of distributed services exploiting the event notification style. JSCL has been originally proposed in [4] to deal with services with a public addressing schema (e.g. each service is directly reachable from the network). To abstract from the particular underlaying network adopted, the middleware has been programmed with a pluggable part, the Inter Object Communication Layer (*iocl*), which provides an unique interface to deal with communication primitives (e.g. message exchange, addressing, etc.).

The main concepts of JSCL are *signals*, *components*, *gateways*, *input ports* and *signal links*. The messages exchanged among participants are modeled as signals that are uniquely identified by a name and are tagged with a *topic*, that represents the event class to which they belong to. A JSCL component represents an autonomous service deployed over at least one overlay network. Here we assume that the public addressing schema of components is relaxed. Hence, to supply service visibility, we introduce intermediate entities called *gateways*. Gateways have a unique public name and are directly reachable from each service. In order to receive signals, a component must join a gateway that acts like bridge among different heterogeneous networks (using several *iocl* instances). To avoid centralization, while allowing interoperability among networks, the same component can join with more gateways. The JSCL component interface is structured into *input ports* and *signal links*. *Input ports* describe component behavior and the parameters bound upon signal reception. Indeed, the reception of a signal acts like a trigger that activates the execution of a new computation within the component. Orchestration among components is implemented through *signal links* that connect outgoing signals to input ports of other components. Signal links, on their turn, are strictly related to a particular topic thus offering the possibility to express different topologies of connectivity,

depending on the topic of the outgoing signals. Both input ports and signal links can be dynamically modified by the components.

3 Implementation Overview

In this section, we outline the design choices adopted for implementing the JSCL extension supporting the two level addressing. Gateways become the unique public visible entities in the global network. Having several *iocl* plugins, one for each network overlay, gateways need to make available on the *iocls* they are interested to operate in. In the following we only deal with two kinds of overlay networks: SOAP with standard HTTP binding and SOAP with the binding proposed in 3.1. Depending on the protocol used to identify a component or a gateway, JSCL instantiate the proper *iocl* (e.g. to *rhttp* corresponds the *iocl* with multipart, etc.). Communication from a gateway to a public service hosted on the same "domain" can be obtained through HTTP binding or through more scalable and efficient ad-hoc solutions (e.g. JMS).

3.1 X-Mixed-Replace SOAP Binding

We propose an alternative SOAP binding for HTTP 1.1 to supply an envelope transport mechanism for services that cannot open local *tcp* ports (e.g. firewalled applications), or that are executed on machines without public address (e.g. internet applications) or that are hosted in an environment that disallows socket management (e.g. Ajax and Comet applications inside a Web Browser). The proposed binding is based on the X-MIXED-REPLACE [7] mimetype and is structured as follows. *(Step1)* The service opens a HTTP 1.1 connection to a potential requester and performs a GET request specifying the information needed for the publication. *(Step2)* The requester sends back a response having mimetype X-MIXED-REPLACE. Usually, this mimetype informs a client that the server will send a stream of multiple versions of the same document. The client and the server must keep opened the HTTP connection, until the server terminates to deliver the stream. *(Step3)* When the requester wants to send a SOAP request to a previously published service, it sends a SOAP envelope over the active HTTP connection, as a new version of the multipart document. *(Step4)* When a new version of the multipart document is received, the SOAP envelope is extracted and the local service is invoked.

As we will see in section 3.2, the first and second steps are performed by the *gateway.register* method, which creates the *virtual channel* between the gateway and the component, and the third and forth steps are performed for routing signals from the gateway to the component.

3.2 JSCL Implementation Outline

The *UML-like sequence diagram* (Figure 1) illustrates the steps performed by JSCL to implement the component registration to a gateway (*block* 1) and the signal exchanging between two components (*blocks* 2, 3 and 4). In the following we will use the notation P_X^S to represent proxies for an entity S (a component or a gateway) communicating through the network via protocol X. Analogously, A_X^S represents an address of the entity S over the network via protocol X.

The *block* 1, defined in Figure 1, describes the steps performed by the component S_1, hosted on $Host_1$, to activate a registration on the gateway G, located on $Host_2$. S_1 demands to the *iocl* to create a proxy P_X^G for the gateway G, having address A_X^G, which will be encapsulated into the proxy instance. The registration method is invoked on the proxy which makes an HTTP request, specifying the encapsulated gateway address and the component identifier (see Step 1 in section 3.1). The request is received by the *iocl* on the $Host_2$ that creates the local proxy P_X^{S1} for the component requester. Notice that the connection ($socket$) established with P_X^G and the component identifier $S_1.id$ are stored into the component proxy. The gateway stores, into the table H, the association between the component identifier and the proxy bound to it. Finally, the *iocl* sends back an HTTP response to declare that further messages will be send on that stream (see Step 2 of section 3.1). As result of a signal emission, the component S_2 retrieves the set of component link descriptors of the form $(A_H^G, S1_{id})$. For each component, S_2 requests a proxy for the intermediate gateway (P_H^G), then invokes its method $spawn$ (*block* 2). The gateway proxy sends a HTTP Post request, containing the signal and the target component identifier, using standard SOAP HTTP binding. At the reception of the message, the *iocl* on $Host_2$ retrieves the proper gateway and invokes its method $spawn$. The gateway retrieves, from the table H, the proxy for the target component that has been created at the registration phase (*block* 3). The component proxy forwards the signal through the multipart stream using the previously encapsulated connection with P_X^G (see Step 3 of section 3.1). Finally, in *block* 4, the gateway proxy retrieves the locally registered component and demands to it the signal handling (see Step 4 of section 3.1).

4 Signal Calculus

SC is a process calculus in the style of [5, 3] introduced in [4] as foundational model of the JSCL middleware. In this section we describe the extension of SC introduced to formal represent the network model considered in this paper.

Component behaviors (B) are defined by the following grammar:

$$B ::= \ 0 \ \mid \ +R[x : \tau \rightarrow B] \ \mid \ +F[\tau \triangleright g[a]] \ \mid \ \triangleright g \ \mid \ \bar{s} : \tau.B \ \mid \ B|B \ \mid \ !B$$

Behaviors represent JSCL computations executed inside a component, while the set of primitives represents the JSCL programmer API. The flow update ($+F[\tau \triangleright g[a]]$) represents the JSCL API to create a new outgoing signal link, it extends the component flow, appending the gateway g handling all signal communications of schema τ with component named a. The gateway join ($\triangleright g$) represents JSCL API to publish the component, it opens a *channel* between the service and the gateway g, suppling the addressing schema for the service. A gateway body contains a tuple (possibly empty) of envelopes to route to the joined components. Gateway bodies (G) are defined by the following grammar:

$$S ::= \ \emptyset \ \mid \ S|S \ \mid \ < s@a : \tau >$$

Networks (N) are defined by the following extended grammar:

$$N ::= \ \emptyset \ \mid \ a[B]_{(R,F,\boldsymbol{g})} \ \mid \ g[S]_{(\boldsymbol{a})} \ \mid \ N \| N \ \mid \ < s@g[a] : \tau >$$

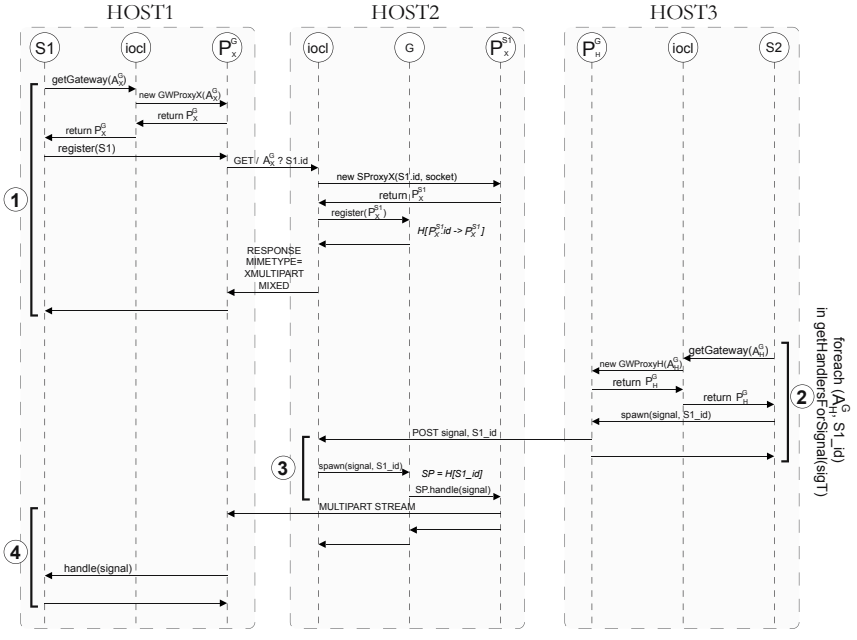

Fig. 1. Registration and signal emission protocol

A network describes the component and gateway topologies, and it is an abstraction of the set of JSCL *iocls* shared among services and gateways. The new primitive gateway ($g[S]_{(\boldsymbol{a})}$) describes a public gateway. The set $\boldsymbol{a}$ identifies the set of component names that have joined the gateway. The component primitive ($a[B]_{(R,F,\boldsymbol{g})}$) has been extended with the tuple ($\boldsymbol{g}$) of gateway names to which the component is linked. Analogously, the signal envelope primitive ($< s@g[a] : \tau >$) has been adapted to contain the gateway g that will effectively deliver the message.

SC components and gateways are closely related to the notion of $Ambient$ [3], but enriched with mechanisms to control the interaction policies among ambients. The SC semantics is defined in a reduction style. Hereafter, we simply provide an example of the reduction rules.

$$\frac{a \in \boldsymbol{a} \;\; g \in \boldsymbol{g} \;\; R \downarrow_{s:\tau} = (\sigma, B)}{g[< s@a : \tau > |S]_{(\boldsymbol{a})} \| a[Q]_{(R,F,\boldsymbol{g})} \rightarrow g[S]_{(\boldsymbol{a})} \| a[\sigma B | Q]_{(R,F,\boldsymbol{g})}} \; (IN)$$

This rule allows an envelope contained into the gateway to react with the component whose name is specified inside the envelope (see step 4 in Figure 1).

5 Concluding Remarks

We have introduced a framework to program coordination policies of distributed services with a two level addressing schema. Unlike current industrial coordination technologies (e.g. BPEL [8]), our solution is based on top of a clear foundational approach.

This should provide strategies to prove coordination properties based on model checking or type systems. A semantic definition of the basic set of primitives can also drive the implementation of translators from industrial specification languages (e.g. WS-CDL [10]) to our framework. Our approach differs from other event based proposals, since it focuses the implementation on the more distributed environment of services. Moreover, neither industrial technologies nor formal approaches handle with a two-level addressing schema without introducing a centralization point.

Acknowledgments. Research partially supported by the EU, within the FETPI Global Computing, Project IST-2005-16004 SENSORIA and by MURST-FIRB Project TOCAI.IT

Bibliography

[1] R. Bruni, G. L. Ferrari, H. C. Melgratti, U. Montanari, D. Strollo, and E. Tuosto. From theory to practice in transactional composition of web services. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2005.

[2] R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In J. Palsberg and M. Abadi, editors, *POPL*, pages 209–220. ACM, 2005.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *FoSSaCS*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[4] G. Ferrari, R. Guanciale, and D. Strollo. JSCL: a Middleware for Service Coordination. In *Proc. FORTE'06*, Lecture Notes in Computer Science, 2006. To appear.

[5] R. Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification, Proceedings of International NATO Summer School (Marktoberdorf, Germany, 1991)*, volume 94 of *Series F*. NATO ASI, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[6] J. Misra. A programming model for the orchestration of web services. In *SEFM*, pages 2–11. IEEE Computer Society, 2004.

[7] Netscape. An Exploration of Dynamic Documents. `http://wp.netscape.com/assist/net_sites/pushpull.html`, 1999.

[8] OASIS Bpel Specifications. OASIS - BPEL. `http://www.oasis-open.org/cover/bpel4ws.html`.

[9] D. Strollo. Java Signal Core Layer (JSCL). Technical report, Dipartimento di Informatica, Università di Pisa, 2005. Available at `http://www.di.unipi.it/~strollo`.

[10] W3C. Web Services Choreography Description Language (v.1.0). Technical report.