Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 194 (2008) 5–22
www.elsevier.com/locate/entcs

A Service-Oriented Model for Embedded Peer-to-Peer Systems

Antonio Brogi, Răzvan Popescu

Department of Computer Science, University of Pisa, Italy

Francisco Gutiérrez, Pablo López, Ernesto Pimentel

Dpto. de Lenguajes y Ciencias de la Computación, Univ. de Málaga, Spain

Abstract

This paper introduces a service-oriented model for the development of Peer-to-Peer (P2P) systems. We first describe the set of abstract primitives supported by the model, and then the semantics for a simple calculus over these primitives that provides a solid ground to develop tools for the analysis and verification of P2P specifications.

Keywords: Peer-to-peer system (P2P), service-oriented model, embedded P2P system (EP2P)

1 Introduction

P2P systems are distributed computing systems where all network elements act both as service consumers (clients) and service providers. Moreover, most P2P communication mechanisms are not based on pre-existing infrastructures, but rather on dynamic ad-hoc networks among peers [13].

Embedded Peer-to-Peer (EP2P) systems [5] represent a new challenge in the development of software for distributed systems. An EP2P system is a P2P system where small, low-powered, low-cost embedded devices cooperate in exchanging and processing information using wireless channels. EP2P systems can be employed in a number of different application areas, including mobile telephony, home systems, or environmental monitoring. EP2P systems present a high degree of heterogeneity (applications may run on different devices, from PDAs to sensor network nodes, with quite different network bandwidths and computing power) and autonomy (the elements enter and exit the system in an independent way, causing frequent reorganisations of the systems). These aspects raise important technological challenges

doi:10.1016/j.entcs.2008.03.096

such as decentralisation, links with transient communications (connections and disconnections happen in an unpredictable and frequent manner), and a constantly changing topology.

One of the keys for the success of these systems is the possibility to suitably abstract from all the above problems by means of convenient middleware. Without middleware (including formal tools and methodologies), the definition, creation and maintenance of an application is expensive and error-prone, having to face from scratch the EP2P problems in every prototype. A convenient middleware should hide the complexity of the underlying infrastructure while providing open interfaces to third parties for application development. The development of such a middleware is challenging, since a number of critical requirements have to be taken into account (such as mobility, new security problems, discovery and localisation protocols, or new quality of software criteria).

The objective of the Secure Middleware for Embedded Peer-To-Peer Systems (SMEPP) European Project is precisely to develop such a middleware, that will have to be secure, generic and highly customisable, allowing for its adaptation to different devices (from PDAs and new generation mobile phones to embedded sensor actuator systems) and domains (from critical systems to consumer entertainment or communication). One of the objectives of SMEPP is to provide a high-level, service-oriented model to program the interaction among peers, thus hiding low-level details that concern the supporting infrastructure. In this paper, we present the set of primitives that have been designed for the SMEPP model in order to satisfy the requirements identified for the SMEPP project. Three key features of the model are the notion of *group* of peers, the notion of *service* offered by peers (or by groups), and the concern of *security*. The core of the service-oriented model borrows concepts from state-of-the-art Web service technologies. On the one hand, we model service contracts using eXtensible Markup Language (XML [15]) schemas and, in particular, we model service signatures similarly to Web Service Description Language (WSDL [17]) interfaces, and the ontology information using the Web Ontology Language (OWL [11]). The model supports one-way (viz., input-only) and request-response (viz., input-output) service operations. On the other hand, we model service behaviour similarly to Business Process Execution Language (BPEL [4]) processes. The model employs communication primitives for invoking operations, for receiving messages and events, as well as for replying to invocations.

After introducing the syntax of the SMEPP primitives (Section 2), we present an abstract semantics for a simple calculus over these primitives (Section 3). Such a semantics formally establishes whether a set of processes (viz., peer or service codes) can be executed together without locking, and it provides a solid ground to develop tools for the analysis and verification of SMEPP specifications.

2 Abstract Primitives

This section overviews the main SMEPP middleware requirements (Subsect. 2.1), followed by an informal description of the key ingredients of the SMEPP service

model (Subsect. 2.2), and finally, by a description of the primitives (Subsect. 2.3).

2.1 Overview of the SMEPP Requirements

This subsection overviews the basic requirements of the SMEPP middleware that drive the definition of the SMEPP service model. For more details on the requirements please see [14].

*Peer*s have to be uniquely identifiable. *Group*s have to provide basic abstraction for *service* providing. Furthermore, *security* has to be bound to groups. Peers have to be able to securely create, locate, and join groups. (Provider) peers have to offer services in groups. Then, (client) peers have to join the respective groups to be able to invoke the offered services. Services have to be identified by *contract*s, which will include information to allow their discovery, adaptation, use or composition with other services. The middleware has to support asynchronous, synchronous, and event-based communication between peers and services.

2.2 Key Concepts

Peers are service containers. A peer has a peer program (viz., code), and it may offer one or more services [1].

Services have contracts and implementations. On the one hand, a contract provides a machine understandable description of a service. On the other hand, the implementation is the executable service (e.g., a Java service). A service contract describes what the service does (service signature), how it does it (service behaviour), how clients can invoke it (service grounding), and it may include other extra-functional service properties (e.g., QoS) and ontology information. Services are of two kinds: behaviour-less and behaviour-full. The contracts of behaviour-less services do not need to expose behaviour, since their operations can be invoked several times and in any order. Behaviour-less services are similar to WSDL services [17]. Behaviour-full services should expose the (partial) behaviour of their interaction protocol. Their operations have to be invoked as indicated by the behaviour information. One may think of behaviour-full services as BPEL processes [4].

Service operations are an abstract way of representing work units. For example, an operation may denote a Java method, or a Web service operation. The communication between entities basically consists in message passing through operation invocations, one entity invokes an operation offered by another entity. Operations are of two kinds: synchronous (viz., “input-output") and asynchronous (viz., “input only"). The invoker of a synchronous operation blocks until the respective operation terminates computing. On the other hand, the invoker of an asynchronous operation continues processing as soon as the operation starts computing. Each entity can be both a provider (offering one or more operations) and a requester (invoking one or more operations).

[1] In the following we shall use *peer* to denote either the *peer container* or the *peer code*, depending on the context. We shall use *entity* to refer to either *peers* or *services*.

Events are a loose way of communication among entities. For simplicity, we model events as operations, hence raising an event is somewhat similar to invoking an operation, while waiting for an event is somewhat similar to waiting for an operation to be invoked. Modelling events as operations allows us to use only one schema to define both (asynchronous) operations and events, as well as, to use only one primitive for the reception of messages modelling both, operation invocations, and events. Different from operations, the entity that raises an event does not have to wait for another entity (viz., an event subscriber) to receive it.

2.3 Informal Description

The analysis of current state-of-the-art models in EP2P systems (e.g., [1,2,6,7,8,9,10]) revealed the fact that existing frameworks for the development of EP2P applications generally do not provide a simple, high-level service (interaction) model that presents a suitable level of abstraction which allows the easy development of P2P applications, Furthermore, such frameworks do not satisfy all the requirements discussed in Subsection 2.1, and they do not model all the concepts described in Subsection 2.2 (such as service contracts or exception handling), as well as they do not provide a formal, abstract language that can be used for application prototyping and for simulating and verifying the behaviour of peers and services, and their interactions. SMEPP aims at overcoming such limitations. One of its main goals is the definition of a set of abstract primitives (driven by the SMEPP requirements) that will eventually be implemented by one or more application programming interfaces. Such primitives are the basic bricks for specifying the code of both peers and services.

Primitives have the following form:

output primitiveName(input$_1$*, ..., input*$_N$*) throws exception* where optional parameters will be annotated by "?".

In the following we informally describe the primitives, which make the core of the SMEPP service model.

peerId **new***(credentials?) throws exception*
Exceptions: invalidCredentials, invalidCall.
Programs call the *new* primitive to become peers. The optional *credentials* parameter serves to authenticate the peer. The first invocation of *new* returns a new unique *peerId* peer identifier. Subsequent invocations to *new* always return the same *peerId*. Calls to other primitives before a call to *new* raise an *invalidCall* exception. *New* raises an *invalidCredentials* exception if the peer cannot be authenticated. Since peers are service containers, services are not allowed to generate (fork) new peers, hence they cannot call the *new* primitive. Consequently, service calls to the *new* primitive raise *invalidCall* exceptions.

groupId **createGroup***(securityLevel?) throws exception*
Exceptions: permissionDenied, invalidCall.
Peers call *createGroup* to start a new peer group with a desired *securityLevel*. *CreateGroup* returns a new unique *groupId* group identifier. The group lasts as long as it contains at least one member. For each freshly created group its list of members

contains only the group creator. If a peer is not be allowed to create a group with a specified *securityLevel*, a *permissionDenied* exception is raised. Note also that a peer service [2] is not allowed to use the *createGroup* primitive, because this may lead to exposing other services of the respective peer as group services. Consequently, service calls to *createGroup* raise *invalidCall* exceptions. For similar reasons, *joinGroup*, *leaveGroup*, *publish*, and *unpublish* (see below) raise *invalidCall* exceptions when called by services.

void **joinGroup***(groupId, credentials?) throws exception*
Exceptions: accessDenied, invalidGroupId, invalidCall.
Peers use the *joinGroup* primitive to enter a group (viz., to become members of a group). The *groupId* specifies the group to join, while the *credentials* serve to validate the peer, that is, to check whether the caller peer is allowed to join the group. If the call succeeds, the middleware adds the caller peer to the list of members of the *groupId* group. A peer may join several groups. Joining the same group several times does not raise an exception. However, a mismatch between the *credentials* supplied by the peer wishing to join the group, and the *securityLevel* of the group raises an *accessDenied* exception. Furthermore, calling *joinGroup* with an invalid *groupId* raises an *invalidGroupId exception.*

void **leaveGroup***(groupId) throws exception*
Exceptions: invalidGroupId, peerNotInGroup, invalidCall.
Peers use the *leaveGroup* primitive to exit a group identified by *groupId.* When a *peerId* peer leaves a *groupId* group (or when *peerId* terminates unexpectedly), the *peerId* will be removed from the list of members of the *groupId* group. Similarly, all the services published by *peerId* in *groupId* will be removed from the list of services offered by the *groupId* group (see *publish* on page 5).

<groupServiceId, peerServiceId> **publish***(groupId, serviceContract) throws exception*
Exceptions: invalidService, invalidGroupId, peerNotInGroup, invalidCall.
Peers offer their services inside a *groupId* group through the *publish* primitive. As previously mentioned, the *serviceContract* provides both abstract (viz., signature, behaviour, QoS, and ontology) and concrete (viz., grounding) information about the service. *Publish* returns a pair <groupServiceId, peerServiceId>, where the former stands for "the identifier of the service seen as a group service", and the latter for "the identifier of the service seen as a peer service". If the call succeeds, the middleware is in charge of adding the service identified by *serviceContract* to the list of services published (viz., available) in the *groupId* group. Service invokers can use the *groupServiceId* to *blindly* invoke a group service (viz., without requesting a specific provider), or the *peerServiceId* to *directly* invoke a peer service. Note that multiple providers publishing services with the same (abstract) *serviceContract* in the same *groupId* group get the same *groupServiceId*, yet different *peerServiceIds*. A call to *publish* fails with an *invalidService* exception if the *serviceContract* does not refer to

[2] In the following we shall use *service* to denote either a *peer service* or a *group service*, depending on the context. If necessary we shall use the complete name to disambiguate.

a valid service, or with an *invalidGroupId* exception if the *groupId* does not point to an existing group. Furthermore, *publish* raises a *peerNotInGroup* exception if the caller peer does not belong to the *groupId* group.

void **unpublish***(groupId, serviceContract) throws exception*
Exceptions: invalidService, invalidGroupId, peerNotInGroup, invalidCall.
Peers stop to offer services in a group by unpublishing them from the respective group. If the call succeeds, the middleware removes the service identified by *serviceContract* from the list of services published in the *groupId* group. *Unpublish* raises the same exceptions as *publish*.

<groupId, id, serviceContract>[] **getServices***(groupId?, peerId?, serviceContract'?) throws exception*
Exceptions: invalidGroupId, invalidPeerId, invalidService.
getServices allows to match published services. The output consists of a list of triples: services identified by *serviceContract*s, which are published in *groupId* groups, and that have *id* identifiers. The *groupId* and *peerId* (optional) input parameters restrict the service discovery to the *groupId* group, and to the *peerId* peer, respectively. *Id* stands for *groupServiceId* or *peerServiceId*. *getServices* raises exceptions when the input group, the peer identifier, or the contract of the desired service are not valid.

The output of *getServices* is one of the following:

- if the matched service is behaviour-less, then *getServices* returns its *groupServiceId*. Consequently, invokers may blindly use this service, so that, two invocations of the service will (possibly) be processed by two different providers (if multiple peers have published the same *serviceContract*). Note that the middleware is in charge of selecting the service provider.

- if the matched service is behaviour-full, then *getServices* returns the *peerServiceId* of a specific provider peer. Consequently, all service invocations done through the *peerServiceId* will be processed by the same peer service.

The core of the matching process compares the contract of the desired service (viz., *serviceContract'*) with the contracts of the published services. The match between two contracts can be done at several levels: syntactic (viz., checking whether the published service offers the operations of the requested service), behavioural (viz., checking whether the published service has a behaviour similar to the requested one, e.g., [3]), or QoS-based matching (viz., checking whether the QoS of the published service satisfies the QoS of the requested service). Furthermore, the discovery process can also exploit the ontology information (if available) of the two contracts (e.g., the Web Ontology Language for Web Services [12]) to improve the service matching.

output? **invoke***(id, operation, input?, QoS?) throws exception*
Exceptions: invalidServiceId, invalidPeerId, invalidOperation, invalidInputParameter, invalidOutputParameter, cannotGuaranteeQoS, accessDenied.
The *invoke* primitive serves to call an *operation* of an entity identified by *id*, which can be either *groupServiceId*, or *peerServiceId*, or *peerId*. It is important to note that the invoker and the provider must belong to the same group. As previously

mentioned blind calls to services are done using *groupServiceId*s, while direct calls to services are done using *peerServiceId*s (see *publish* on page 5). Furthermore, clients can call peers through *peerId*s. *Input* is the input message of the operation, while *output* is its output message. The type of the operation determines the type of the call, that is, either synchronous, or asynchronous. In both cases, the *invoke* blocks until the provider does a corresponding *receive* (see *receive* on page 7). For asynchronous operations, the client (viz., invoker) does an *invoke*, and the provider does just a *receive*. For synchronous operations, the caller has to do an *invoke*, while the provider has to do first a *receive*, and at a later moment a *reply* (see *reply* on page 8). The *invoke* raises exceptions due to invalid service or peer identifiers. Furthermore, an *invalidOperation* exception indicates that either the provider does not support the respective operation, or that it is unable to *receive* a message on the respective operation at that point. *Invoke* also raises exceptions when there is a mismatch between the expected and the actual parameters of the invoked operation. Services can be invoked using a desired *QoS* level. A mismatch between the desired *QoS* and the actual *QoS* of the provider raises a *cannotGuaranteeQoS* exception. Finally, *invoke* signals an *accessDenied* exception if the client and the provider do not belong to the same group.

void **event***(id?, operation, input?) throws exception*
Exceptions: invalidServiceId, invalidPeerId, invalidOperation, invalidInputParameter, invalidOutputParameter, cannotGuaranteeQoS, accessDenied.
Event is somewhat similar to an asynchronous *invoke*. It raises an *operation* event that contains some *input* data, which is similar to invoking an *operation* with an *input* message. The main difference is that *event* is non-blocking, hence the caller does not have to wait for a matching *receive* in the provider. Note that the middleware is in charge of storing raised events, and of forwarding them to providers. In order to be notified of such events, providers have to subscribe first to the events of interest (see also *subscribe*). Then, providers catch the raised events using the *receive* operation (see also *receive*). Furthermore, specifying a provider is optional. In this case the event is to be sent to all providers that subscribed to the respective *operation* event, and that are in a same group with the provider. Note that if the provider attempts to catch events using *receive*s on synchronous operations, the middleware raises an *invalidOperation* exception. The rest of the exceptions raised by *event* are similar to the ones raised by the *invoke*.

<callerId, input?> **receive***(operation) throws exception*
Exceptions: invalidOperation, invalidInputParameter.
As previously mentioned, *receive* serves to wait for clients to invoke the respective (synchronous or asynchronous) *operation*, or for *operation* events to be raised. Note that, for services, the provider peer of the service that calls *receive* has to have previously published a contract that defines the respective *operation* in its signature. [3] The output of the *receive* contains the identifier of the caller (viz., *callerId*, which can be a *peerId*, or a *peerServiceId*), as well as the *input* message of the operation

[3] For simplicity, we use for both operations and events the same signature notation. Assume also that *operation* stands for the name of the operation or of the event.

(if any). Consequently, the caller of *receive* can send back information to its *callerId* client through an *invoke*, *event*, or *reply* primitive (described next). The exceptions raised by the *receive* are similar to the ones raised by the *invoke*.

void **reply***(callerId, operation, output?) throws exception*
Exceptions: invalidServiceId, invalidPeerId, invaildOperation.
The *reply* marks the termination of a synchronous *operation*. Hence, the caller of the *reply* has to have previously executed a corresponding *receive* on the same *operation*. Once the *reply* is executed, the invoker (viz., client) of the *operation* unlocks from its invocation. *Output* is the return message of the *operation*. Again, the exceptions raised by the *reply* are similar to the ones raised by the *invoke*.

void **subscribe***(id?, <groupId, operation>?) throws exception*
Exceptions: invalidServiceId, invalidPeerId, invalidGroupId, invalidOperation, peerNotInGroup.
Entities use the *subscribe* primitive to subscribe to:

- all events raised by a group service (if *id* is a *groupServiceId*), or by a particular peer service (if *id* is a *peerServiceId*), or by a peer (not by its services) (if *id* is a *peerId*), or
- to *operation* events raised in a *groupId* group by an *id* provider (if the *subscribe* specifies the *id*), or by any provider (if the *subscribe* does not specify the *id*).

Note that it is mandatory for *subscribe* to specify at least one input parameter. The exceptions raised by *subscribe* are similar to exception raised by the previously discussed primitives.

void **unsubscribe***(id?, <groupId, operation >?) throws exception*
Exceptions: invalidServiceId, invalidPeerId, invalidGroupId, invalidOperation, peerNotInGroup, notSubscribed.
The caller of *unsubscribe* (partially) cancels a previous subscription. Although *unsubscribe* requires a previous subscription, the two primitive calls do not have to match exactly. For example, a *unsubscribe(peerId, <groupId, operation>)* matches a previous *subscribe(peerId)*. In this case, the caller will not be notified anymore when the *peerId* peer raises *operation* events inside the *groupId* group. Unsubscribing from all events can be done by omitting all primitive's parameters. With respect to *subscribe*, *unsubscribe* raises an additional *notSubscribed* exception if there is no previous subscription of the caller that matches the *unsubscribe*.

3 A Calculus for SMEPP Primitives

To formally define the calculus for SMEPP primitives and the corresponding semantics, we need to introduce some preliminary concepts which will be used in the rest of the paper. Let $\mathcal{P}$ and $\mathcal{S}$ be the sets of peer and service identifiers, respectively. We will also consider a set of group identifers, $\mathcal{G}$, including a special symbol 0 which will be named the *universal* group. To identify the system's entities we will use addresses from a middleware uniform resource locator ($MURL$), given by $\mathcal{G} \times \mathcal{P} \times \mathcal{S}$, where $g.p.s$ will identify a service s in a peer p running in a group g.

When g and s are 0 the address denotes the peer code. Note that the peer code is being considered as a special service (denoted by 0) running in the universal group (also denoted by 0).

As it was previously mentioned, services will be characterized by a contract service and a set of operations. We will denote by $\mathcal{C}$ and $\mathcal{O}$ the sets of contract services and operations signatures, respectively. We will denote by $C^{\emptyset}$ the empty contract. The way in which these sets are defined is not relevant for our purposes, although usually they will be given by XML specifications.

A group is denoted by a labelled triple $\langle P, Sr, Sb\rangle_g$ consisting of a set $P \subseteq \mathcal{P}$ of peer identifiers (which represent the *members* of the group), a set Sr of service identifiers (*services provided* by the group), and a set Sb of subscriptions of entities to events provided by other services in the group. The set of services running in the group is a set of tuples, i.e., $Sr \subseteq \mathcal{S} \times \mathcal{C} \times \mathcal{P}$, where the first component is the service identifier, the second one is the contract exposed by the service, and the last one is the identifier of the peer providing the service. The set of subscriptions to events is a set of triples, i.e., $Sb \subseteq MURL \times MURL \times \mathcal{O}$, where the first component denotes the subscribed entity, the second one is the entity to which it is subscribed, and the third component represents the signature of the subscribed event.

3.1 Programs, Environments, and Executions

A peer executes a peer program and it may offer one or more services, also defined by programs. A program or agent is given by the following syntax:

$$A ::= x = new().B \mid [B]_{MURL}$$
$$B ::= \mathsf{0} \mid a.B \mid B \parallel B \mid B \oplus B \mid D(\boldsymbol{x})$$

where $\mathsf{0}$ is the empty program, a denotes a SMEPP primitive, and D represents a program definition of the form $D(\boldsymbol{y}) \stackrel{def}{=} B$. Programs may be built by using the prefix (.), the parallel composition ($\parallel$), and the guarded non-deterministic choice ($\oplus$). For the sake of simplicity, we will consider the following simplified syntax of SMEPP primitives, abstracting away from some of their parameters: For instance, we will not model explicitly security aspects (e.g., the *credentials*, *securityLevel* nor *quality of service* parameters of group primitives). On the other hand, the output of some primitives is also simplified, because we consider locators (MURL) to encode service identifiers both for groups and peers (this is the case for publishand getServicesprimitives). Finally, the primitive getServicesdoes not return any service contract because the input parameter represents a definite service contract instead of a partial specification of a contract. Note that neither of these simplifications affect the expressive power of the language.

$$\begin{aligned} a ::= {} & x = createGroup() \mid joinGroup(g) \mid leaveGroup(g) \\ & \mid x = publish(g, cs) \mid unpublish(g, cs) \mid x = getServices(m, cs) \\ & \mid x? = invoke(m, op, in) \mid event(m?, op, in) \mid \langle x, y?\rangle = receive(op) \mid reply(m, op, out) \\ & \mid subscribe(m, op) \mid unsubscribe(m, op) \end{aligned}$$

where g stands for *groupId*, cs for *serviceContract*, m for $MURL$, op for *operation*, in for *input*, and out for *output*.

A program P may be either a labelled code ($B_M URL$), when the program is being executed by a peer already identified in the middleware, or a non-labelled code (B), when the program is still waiting for being registered in the middleware.

To define the semantics of these programs we introduce judgements of the form:

$$\Theta \,;\Gamma \rightsquigarrow \Pi$$

where Γ is a set of groups (i.e. labelled triples) called the *environment* and Π is a set of programs. On the other hand, $\Theta \subseteq \mathcal{E} \times MURL \times MURL \times \mathcal{O}$ (where $\mathcal{E}$ is a set of event identifiers) denotes the set of events to be processed by the receivers, called the *polling context*. Thus, each time an event $n \in \mathcal{O}$ is produced by an entity $g.p.s$, an element in Θ of the form $(e, g'.p'.s', g.p.s, n)$ is generated for each subscribed entity $g'.p'.s'$. All the tuples corresponding to the same event are identified by a shared unique identifier $e \in \mathcal{E}$. To extract a polling context from a set of subscribed entities $Sb \subseteq MURL \times MURL \times \mathcal{O}$ associated to a given event identifier $e \in \mathcal{E}$ we define the following operator:

$$e \cdot Sb = \{(e, id, id', n) : (id, id', n) \in Sb\}$$

Given a set of entities running in a group $Sr \subseteq \mathcal{S} \times \mathcal{C} \times \mathcal{P}$ and a peer identifier p, we define the *removal* of p from Sr, denoted by $Sr \downharpoonleft p$ as the set obtained by removing from Sr all the triples containing p:

$$Sr \downharpoonleft p = \{(s, c, q) \in Sr : q \neq p\}$$

Similarly, given a set of locators in $MURL$, S, we overload the operator $\downharpoonleft$ to apply it over polling contexts and event subscriptions as follows:

$$\Theta \downharpoonleft S = \{(e, id, id', n) \in \Theta : id \notin S \wedge id' \notin S\}$$
$$Sb \downharpoonleft S = \{(id, id', n) \in Sb : id \notin S \wedge id' \notin S\}$$

Abusing notation, $g.p$ will represent the set of all possible services $g.p.s$ provided by p in g. Finally, we define:

$$\Theta \downharpoonleft (id, id', n) = \{(e, id_1, id'_1, n_1) \in \Theta : (id_1, id'_1, n_1) \neq (id, id', n)\}$$

The intended meaning of these judgements is that the programs in Π can be concurrently executed to termination, starting from the environment Γ, i.e. these judgements express *termination*.

The semantics is described by means of inference rules of the form:

$$\frac{Pr \qquad \Theta \,;\Gamma \rightsquigarrow \Pi}{\Theta' \,;\Gamma' \rightsquigarrow \Pi'} \qquad (\textsc{Name})$$

where the judgement above the line is a hypothesis, the one below the line a conclusion, and Pr a proviso.

$$\frac{p \text{ fresh} \qquad \Theta\,;\Gamma,\langle P\uplus\{p\},Sr\uplus(0,C^{\emptyset},p),Sb\rangle_0 \rightsquigarrow [A[p/x]]_{0.p.0},\Pi}{\Theta\,;\Gamma,\langle P,Sr,Sb\rangle_0 \rightsquigarrow x=\mathsf{new}().A,\Pi} \quad \text{(New)}$$

$$\frac{g \text{ fresh} \qquad \Theta\,;\Gamma,\langle\{p\},\emptyset,\emptyset\rangle_g \rightsquigarrow [A[g/x]]_{0.p.0},\Pi}{\Theta\,;\Gamma \rightsquigarrow [x=\mathsf{createGroup}().A]_{0.p.0},\Pi} \quad \text{(CreateGroup)}$$

$$\frac{\Theta\,;\Gamma,\langle P\uplus\{p\},Sr,Sb\rangle_g \rightsquigarrow [A]_{0.p.0},\Pi}{\Theta\,;\Gamma,\langle P,Sr,Sb\rangle_g \rightsquigarrow [\mathsf{joinGroup}(g').A]_{0.p.0},\Pi} \quad \text{(JoinGroup)}$$

$$\frac{\Theta\downharpoonright g.p\,;\Gamma,\langle P,Sr\downharpoonright p,Sb\downharpoonright g.p\rangle_g \rightsquigarrow [A]_{0.p.0},\Pi}{\Theta\,;\Gamma,\langle P\uplus\{p\},Sr,Sb\rangle_g \rightsquigarrow [\mathsf{leaveGroup}(g).A]_{0.p.0},\Pi} \quad \text{(LeaveGroup)}$$

$$\frac{\begin{array}{c}s \text{ fresh} \qquad (s,cs,p)\notin Sr\\ \Theta\,;\Gamma,\langle P\uplus\{p\},Sr\uplus(s,cs,p),Sb\rangle_g \rightsquigarrow [A[s/x]]_{0.p.0},[program(cs,p)]_{g.p.s},\Pi\end{array}}{\Theta\,;\Gamma,\langle P\uplus\{p\},Sr,Sb\rangle_g \rightsquigarrow [x=\mathsf{publish}(g,cs).A]_{0.p.0},\Pi} \quad \text{(Publish)}$$

$$\frac{\Theta\downharpoonright g.p.s\,;\Gamma,\langle P\uplus\{p\},Sr,Sb\downharpoonright g.p.s\rangle_g \rightsquigarrow [A]_{0.p.0},\Pi}{\Theta\,;\Gamma,\langle P\uplus\{p\},Sr\uplus(s,cs,p),Sb\rangle_g \rightsquigarrow [\mathsf{unpublish}(g,cs).A]_{0.p.0},\Pi} \quad \text{(Unpublish)}$$

$$\frac{\begin{array}{c}g''.p''.s''\leq g'.p'.\star\\ \Theta\,;\Gamma,\langle P,Sr\uplus(s'',cs,p''),Sb\rangle_{g''} \rightsquigarrow [A[g''.p''.s''/x]]_{g.p.s},\Pi\end{array}}{\Theta\,;\Gamma,\langle P,Sr\uplus(s'',cs,p''),Sb\rangle_{g''} \rightsquigarrow [x=\mathsf{getServices}(g'.p'.\star,cs).A]_{g.p.s},\Pi} \quad \text{(Get Services)}$$

$$\frac{\Theta\,;\Gamma,\langle P\uplus\{p,p'\},Sr,Sb\uplus(g.p.s,g'.p'.s',n)\rangle_{g'} \rightsquigarrow [A]_{g.p.s},\Pi}{\Theta\,;\Gamma,\langle P\uplus\{p,p'\},Sr,Sb\rangle_{g'} \rightsquigarrow [\mathsf{subscribe}(g'.p'.s',n).A]_{g.p.s},\Pi} \quad \text{(Subscribe)}$$

$$\frac{id=(g.p.s,g'.p'.s',n) \qquad \Theta\downharpoonright id\,;\Gamma,\langle P\uplus\{p'\},Sr,Sb\rangle_{g'} \rightsquigarrow [A]g.p.s,\Pi}{\Theta\,;\Gamma,\langle P\uplus\{p'\},Sr,Sb\uplus id\rangle_{g'} \rightsquigarrow [\mathsf{unsubscribe}(g'.p'.s',n).A]_{g.p.s},\Pi} \quad \text{(Unsubscribe)}$$

Fig. 1. SMEPP Calculus: Groups, Peers and Services.

For every construct of the program —primitive actions and composition operators – we have inference rules describing its semantics. In addition, we shall have an axiom to indicate the termination of an execution:

$$\frac{}{\,;\Gamma \rightsquigarrow} \quad \text{(Terminate)}$$

stating that the empty program terminates in any "legal" environment Γ with the empty polling context. A (successful) finite execution is a sequence of judgements whose first element is a termination judgement and such that every other judgement is the conclusion of the previous one.

Operationally, we start from a given judgement and search for an execution by applying the inference rules bottom-up until we get a termination judgement (successful finite execution), or a non-termination judgement (lock). We can also consider infinite executions as infinite proof sequences.

3.2 Peers, groups and services

Figure 1 presents the judgements for the first set of primitives necessary to create peers, create, join or leave groups, to publish, unpublish and discover services, as well as to subscribe and unsubscribe to events. It is worth noting that some primitives can only be executed by peers code, and they are not available in services code.

Thus, rules modelling these primitives are applied only to peers tagged by $0.p.0$, the MURL identifier for p.

Every peer program must begin with a `new` primitive (rule New). By executing `new`, the program A is assigned a unique peer identifier p that is used to index its code with a locator, and the code is executed as a peer code (with locator $(0.p.0)$). Note that the peer is added to the universal group (0), and a new special service is added with the empty contract $C^{\emptyset}$ as contract service. The rest of the rules deal only with programs indexed by locators.

Any peer code $0.p.0$ can create a new group g (rule CreateGroup). Similarly, a peer p can always join (rule JoinGroup) and leave (rule LeaveGroup) an existing group g.

When a peer leaves a group, the pending events addressed to or originated from the peer p in group g are removed from Θ. Similarly, the services published by p in g are removed from Sr, and the subscriptions to and from p in g are withdrawn from Sb.

The publish primitive allows a peer p to publish a service described by a service contract cs in a group g. For the publication to take effect, p must be a member of g. The behavior of publish is modelled by rule Publish, which is applied when the service has not been previously published by p in g. In this case, the new service is added to Sr and a new provider $[program(cs, p)]_{g.p.s}$ (the code residing in p, whose contract service is cs) is started.

A service cs previously published by a peer p in a group g can be unpublished by means of the unpublish primitive. Besides removing the service offered (s, cs, p), the events and subscriptions affected by $g.p.s$ are removed from Θ and Sb, respectively. This is defined by rule Unpublish.

On the other hand, service discovery is accomplished by the primitive getServices, that receives a given contract service cs and a locator $g'.p'.\star$ where g', p' or even both can be left unspecified $(\star)$. This locator imposes a constraint upon the location of the service being searched for. An appropriate service (s'', cs, p'') is discovered such that its location is definite and precedes the locator constraint. To do this, we consider an order relationship in $MURL$ extended by adding $\star$ to $\mathcal{G}$, $\mathcal{P}$ and $\mathcal{S}$, defined by $g.p.s \leq g'.p'.s'$ if and only if $g = g'$ or $g' = \star$, and $p = p'$ or $p' = \star$, and $s = s'$ or $s' = \star$. Note that the primitive, such as it was presented in Section 2, returns a collection of services instead of non-deterministically selecting one of them, as we have considered in the judgement to simplify the semantics.

A program $[A]_{g.p.s}$ can subscribe to a particular event n generated by another program located at $g'.p'.s'$. A new tuple is added to Sb to record the subscription. It should be noted that both p and p' must be members of g'. The actual handling of events involves the use of Θ, that stores the pending events, and new primitives to be described later.

In the same vein, the primitive unsubscribe allows a program to cancel the subscription to a particular event n raised by a program located at $g'.p'.s'$. Besides canceling the subscription, all of the pending events of type n addressed from $g'.p'.s'$ to $g.p.s$ are removed from Θ.

3.3 Service Invocation and Event Handling

In Figure 2 we present the judgements concerning to service invocation and event handling. In fact, an entity $g.p.s$ may send an event n to another entity $g'.p'.s'$, such as it is modelled in rule EVENT. Sending an event should not prevent the progression of the sender, even if the receiver is not ready to process it. To formally define this asynchronous communication, the sender will create a special service (dispatch) in charge of communicating the event when the receiver is ready. The middleware must store the service locator and the unique (fresh) event identifier, e, provided by the middleware itself. Note that the receiver entity $g'.p'.s'$ must be previously registered as subscribed to the event n in the group g. Similarly, an event may also be broadcasted to any of the entities requesting it, as it is specified in rule BROAD EVENT.

When an entity $g'.p'.s'$ is ready to accept an event n, it executes the receive primitive. The special service dispatch, generated by the sender ($g.p.s$) —either through the rule EVENT or BROAD EVENT— will communicate the event to that entity if it is one of the authorised entities (i.e. Θ includes information about that). In any case, the peer p', where the receiver entity is being executed, must be located in the same group that the peer p, where the sender entity was located. As a consequence of the communication, actual parameters in are conveniently passed to the receiver process (see rule RECEIVE).

Entities also may invoke operations of other entities by means of the invoke primitive. Again, the invoked entity must execute the receive primitive to have a successful communication. Whereas the event primitive is "asynchronous", the invoke primitives needs to "synchronise" with a receive operation. Once the communication is produced, after transferring the information through the arguments, both sender and receiver entities must progress independently. Note that the sender and the receiver entities must be located at the same group.

The rule INVOKE ASYNC defines the behaviour of the invoke primitive with three arguments. Rule INVOKE SYNC models an alternative version of the invoke primitive. Again, the invoked entity must execute the receive primitive, but in this case the receiver entity gets the information provided as input arguments and also the invoker's locator, because it will be locked waiting for an answer. This locking situation is modelled by the auxiliar primitive suspend. Receiver and sender entities must be located at the same group.

The two previous rules have modelled primitives to invoke operations which were hosted by specific entities. However, the SMEPP model permits the invocation of services provided by a group, independently of the peer which is actually servicing it. These alternative invoke primitives also have the corresponding synchronous (rule INVOKE SYNC GROUP) and asynchronous (rule INVOKE ASYNC GROUP) versions.

Finally, an entity locked by the auxiliar primitive suspend may progress only if a reply primitive is executed answering the previous invocation (rule REPLY). The provider entity replies by using the invoker locator provided by the invocation. The entity waiting for the answer receives the result before continuing its execution. The entity which originally made the invocation and the entity providing the result must

$$\frac{e \textit{ fresh} \qquad id = (g'.p'.s', g.p.s, n) \qquad \Theta \uplus e \cdot \{id\} \,;\Gamma, \langle P, Sr, Sb \uplus id\rangle_g \rightsquigarrow [A]_{g.p.s}, [\mathsf{dispatch}(e,n,in)]_{g.p.s}, \Pi}{\Theta \,;\Gamma, \langle P, Sr, Sb \uplus id\rangle_g \rightsquigarrow [\mathsf{event}(g'.p'.s', n, in).A]_{g.p.s}, \Pi} \quad \text{(Event)}$$

$$\frac{e \textit{ fresh} \qquad Sb' = \{(id', id, m) \in Sb : id = g.p.s \wedge m = n\} \qquad \Theta \uplus e \cdot Sb' \,;\Gamma, \langle P, Sr, Sb\rangle_g \rightsquigarrow [A]_{g.p.s}, [\mathsf{dispatch}(e,n,in)]_{g.p.s}, \Pi}{\Theta \,;\Gamma, \langle P, Sr, Sb\rangle_g \rightsquigarrow [\mathsf{event}(n,in).A]_{g.p.s}, \Pi} \quad \text{(Broad Event)}$$

$$\frac{\Theta \,;\Gamma, \langle P \uplus \{p'\}, Sr, Sb\rangle_g \rightsquigarrow [A[in/x]]_{g'.p'.s'}, [\mathsf{dispatch}(e,n,in)]_{g.p.s}, \Pi}{\Theta \uplus (e, g'.p'.s', g.p.s, n) \,;\Gamma, \langle P \uplus \{p'\}, Sr, Sb\rangle_g \rightsquigarrow [x = \mathsf{receive}(n).A]_{g'.p'.s'}, [\mathsf{dispatch}(e,n,in)]_{g.p.s}, \Pi} \quad \text{(Receive)}$$

$$\frac{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr, Sb\rangle_{g'} \rightsquigarrow [A[in/x]]_{g'.p'.s'}, [B]_{g.p.s}, \Pi}{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr, Sb\rangle_{g'} \rightsquigarrow [x = \mathsf{receive}(n).A]_{g'.p'.s'}, [\mathsf{invoke}(g'.p'.s', n, in).B]_{g.p.s}, \Pi} \quad \text{(Invoke Async)}$$

$$\frac{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr, Sb\rangle_{g'} \rightsquigarrow [A[g.p.s/x, in/y]]_{g'.p'.s'}, [z = \mathsf{suspend}(n).B]_{g.p.s}, \Pi}{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr, Sb\rangle_{g'} \rightsquigarrow [\langle x, y\rangle = \mathsf{receive}(n).A]_{g'.p'.s'}, [z = \mathsf{invoke}(g'.p'.s', n, in).B]_{g.p.s}, \Pi} \quad \text{(Invoke Sync)}$$

$$\frac{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr \uplus (s', \_, p'), Sb\rangle_{g'} \rightsquigarrow [A[in/x]]_{g'.p'.s'}, [B]_{g.p.s}, \Pi}{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr \uplus (s', \_, p'), Sb\rangle_{g'} \rightsquigarrow [x = \mathsf{receive}(n).A]_{g'.p'.s'}, [\mathsf{invoke}(g'. \star .s', n, in).B]_{g.p.s}, \Pi} \quad \text{(Invoke Async Group)}$$

$$\frac{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr \uplus (s', \_, p'), Sb\rangle_{g'} \rightsquigarrow [A[g.p.s/x, in/y]]_{g'.p'.s'}, [z = \mathsf{suspend}(n).B]_{g.p.s}, \Pi}{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr \uplus (s', \_, p'), Sb\rangle_{g'} \rightsquigarrow [\langle x, y\rangle = \mathsf{receive}(n).A]_{g'.p'.s'}, [z = \mathsf{invoke}(g'. \star .s', n, in).B]_{g.p.s}, \Pi} \quad \text{(Invoke Sync Group)}$$

$$\frac{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr, Sb\rangle_{g'} \rightsquigarrow [A]_{g'.p'.s'}, [B[out/x]]_{g.p.s}, \Pi}{\Theta \,;\Gamma, \langle P \uplus \{p, p'\}, Sr, Sb\rangle_{g'} \rightsquigarrow [\mathsf{reply}(g.p.s, n, out).A]_{g'.p'.s'}, [x = \mathsf{suspend}(n).B]_{g.p.s}\Pi} \quad \text{(Reply)}$$

Fig. 2. SMEPP Calculus: Events and Messages.

belong to the same group.

3.4 Program Composition Rules

The model supports parallel composition, non-deterministic choice, and the void program, as shown in Figure 3. The calculus also provides some rules which guarantee that the Terminate axiom is achieved. These judgements are known as *weakening* rules. When a service which was created for event propagation is waiting for its consumption, and the polling context does not contain information about any subscribed service to the corresponding event, the signal must be removed from the context (rule Event W).

Similarly, when a service s running in a peer p and a group g is not among the active services in the group, Sr, the service must be finalized. Any reference to that service in the subscribed services set Sb and the polling context Θ must be deleted, both as a sender or a receiver (rule Service W).

Finally, when a group g' includes a service $g.p.s$ among its subscribed services, which is not active in the group g because of a certain reason (e.g. the peer leaves the group, the service is unsubscribed, ...), it must be removed from the set of subscribed services in g' (rule Subscribe W).

$$\frac{\Theta\,;\Gamma \rightsquigarrow A, B, \Pi}{\Theta\,;\Gamma \rightsquigarrow A \parallel B, \Pi}\ (\textsc{Parallel}_1) \qquad \frac{\Theta\,;\Gamma \rightsquigarrow [A]_{g.p.s}, \Pi}{\Theta\,;\Gamma \rightsquigarrow [A \oplus B]_{g.p.s}, \Pi}\ (\textsc{Choice}_1)$$

$$\frac{\Theta\,;\Gamma \rightsquigarrow [A]_{g.p.s}, [B]_{g.p.s}, \Pi}{\Theta\,;\Gamma \rightsquigarrow [A \parallel B]_{g.p.s}, \Pi}\ (\textsc{Parallel}_2) \qquad \frac{\Theta\,;\Gamma \rightsquigarrow [B]_{g.p.s}, \Pi}{\Theta\,;\Gamma \rightsquigarrow [A \oplus B]_{g.p.s}, \Pi}\ (\textsc{Choice}_2)$$

$$\frac{\Theta\,;\Gamma \rightsquigarrow \Pi}{\Theta\,;\Gamma \rightsquigarrow [\mathsf{0}]_{g.p.s}, \Pi}\ (\textsc{Nil})$$

$$\frac{(e, \_, \_, \_) \notin \Theta \qquad \Theta\,;\Gamma \rightsquigarrow \Pi}{\Theta\,;\Gamma \rightsquigarrow [\mathsf{dispatch}(e, n, in)]_{g.p.s}, \Pi}\ (\textsc{Event W})$$

$$\frac{(s, \_, p) \notin Sr \qquad \Theta \downarrow g.p.s\,;\Gamma, \langle P, Sr, Sb \downarrow g.p.s\rangle_g \rightsquigarrow \Pi}{\Theta\,;\Gamma, \langle P, Sr, Sb\rangle_g \rightsquigarrow [A]_{g.p.s}, \Pi}\ (\textsc{Service W})$$

$$\frac{(s, \_, p) \notin Sr \qquad \Theta\,;\Gamma, \langle P, Sr, Sb\rangle_g, \langle P', Sr', Sb'\rangle_{g'} \rightsquigarrow \Pi}{\Theta\,;\Gamma, \langle P, Sr, Sb\rangle_g, \langle P', Sr', Sb' \uplus (g.p.s, g'.p'.s', n)\rangle_{g'} \rightsquigarrow \Pi}\ (\textsc{Subscribe W})$$

Fig. 3. SMEPP Calculus: Composition and weakening rules.

4 An application example

To illustrate the usage of the proposed model to define peers and services in a P2P architecture, and in particular, how the two alternatives (invokeand eventprimitives) to make these entities to interact each other are used, we consider the following example which deals with a provider of temperatures read from an external device, and a service which maintains the average of all readings. We define programs on two different peers. The first one, `TempPeer`, creates a group and publishes a service described by the contract `TempServCS` (the service is called in the example `TempServ`). After processing other tasks, the peer unpublishes the service. An implementation of both `TempPeer` and the `TempServ` service is given in the upper side of the Figure 4, where either a terminating signal is received or the temperature is read from some external device and it is communicated to any subscriber to the event `"Temperature"`. The lower side of the Figure 4 defines the behaviour of a client, `PeerClient`, discovering the service `TempServ` (through the corresponding contract service) in any group, and joining to the group before publishing a new service, `TempServInvoker` (given by the contract service `TempServInvokerCS`), which computes the temperature average. After that, operations `"Start"` and `"GetAverage"` are invoked on that service, such that the service locator is transferred to `TempServInvoker` and the average is obtained. Note that we are using string literals for denoting operation names, and the symbol + to represent $\oplus$.

5 Concluding Remarks

In this paper we have presented a set of primitives devising a model for secure P2P architectures. Our aim was to provide a high-level, service-oriented model to specify the interaction among peers, and also to define a simple semantics enabling the

```
TempPeer =
  p=new().
  g=create().
  s=publish(g,TempServCS).
  // Processing other tasks
  // Now invoke TempServ of p
  invoke(g.p.s,"Terminate").
  unpublish(g,TempServCS).
  0
```

```
TempServ =
  // temp=...
  event("Temperature",temp).
  TempServ.
  +
  receive("Terminate").
  0
```

```
PeerClient =
  p=new().
  g.q.s=getServices(*.*.*,TempServCS).
  join(g).
  s'=publish(g,TempServInvokerCS).
  invoke(g.p.s','Start",g.q.s).
  // Processing other tasks
  avg=invoke(g.p.s','GetAverage").
  invoke(g.p.s','Terminate").
  unpublish(g,TempServInvokerCS).
  // Processing the average avg
  0
```

```
TempServInvoker =
  id=receive("Start").
  subscribe(id,"Temperature").
  AverageTemp(0,0,id)

AverageTemp(a,n,id) =
  t=receive("Temperature").
  // a'=a + t, n'=n + 1
  AverageTemp(a',n',id)
  +
  id'=receive("GetAverage").
  // avg = n==0?ERROR:a/n
  reply(id', "GetAverage", avg).
  AverageTemp(a,n,id)
  +
  receive("Terminate").
  unsubscribe(id,"Temperature").
  0
```

Fig. 4. Two peers providing services.

reasoning on abstract specifications of P2P systems. To this end, we have presented a semantics based on a calculus where judgements represent concurrent executions controlled by the middleware (which is explicitly modelled by two components, the *environment* and the *polling* context). The idea behind of providing such a semantics is giving the possibility of constructing a verification environment to analyse properties like lock freedom, correct termination, etc.

Although the actual usability of the service model in embedded system is one of the key objectives of the SMEPP project, the proposed model can be employed to specify general (i.e., not necessarily embedded) peer-to-peer systems. The embeddedness will be addressed at the software architecture and at the implementation level by deploying prototypes of the middleware for devices with different computing capabilities (ranging from laptops to pocket PCs to smart phones).

Various other service and interaction models have been proposed for modelling EP2P systems. Some of them are inspired by the service-oriented architecture paradigm (e.g., [6,10]), others are based on/extend JXTA [8] (e.g., [1,2]), while others are data-driven coordination models (e.g., [7,9]). Maheshwari et al. [10] propose a service model based on a message queue cluster that intercepts and delivers (Simple Object Access Protocol, or SOAP for short [16]) messages exchanged by peers (Web services) so as to achieve high scalability, availability, fault tolerance, and load balancing. Gehlen and Pham [6] model peer interfaces to the distributed environment through SOAP components, which serve for exchanging, encrypting and marshalling SOAP messages. Their approach employs local and remote registries to store WSDL descriptions of the services deployed in the framework, and remote services, respectively. Alda and Cremers [1] describe DeEvolve, a P2P architecture based on Juxtapose (JXTA [8]). DeEvolve introduces two languages: CAT – for expressing peer services as compositions of components, and PeerCAT – for ex-

pressing compositions of peer services. A main feature of DeEvolve is that PeerCAT can define exception handlers to cope with peer failures.

Bisignano et al. [2] introduce JMobiPeer, a P2P computing platform developed on top of JXTA. JMobiPeer defines modules for transport and service protocols, for peer and peer group management, and for peer advertisement and discovery management. Similarly to JXTA, advertisements provide information of available services, peers and groups, as well as pipes and end points. Handorean et al. [7] introduce *follow-me sessions* that express the interaction of a client with a service that is offered by several providers, in order to achieve a continuity of service provision. The paper discusses techniques for migrating processes between hosts, or partial results to alternate providers, for allowing temporary client disconnections while providers continue processing, and for letting clients use partial results until an alternate provider is found.

Lucchi and Zavattaro [9] describe WSSecSpaces (W3S), Linda-based interaction model for Web services. The model allows for loosely-coupled Web services, in the way that a Web service can issue a request and then terminate. Then, the request is processed at a later time e.g., by a service that becomes online.

A thorough comparative analysis can be however found in [14]. However, as previously mentioned, existing service and interaction models either do not take into account key requirements such as security bound to groups, asynchronous, synchronous, and event-based communication, as well as service contracts, or they do not provide a formal, abstract language that can be used for application prototyping and for simulating and verifying the behaviour of peers and services, and their interactions.

As future work, we have to explore the expressiveness of the model by specifying real and more complex case studies in order to validate it on specific domains such as environmental monitoring in industrial plants, and mobile telephony. In addition, the model has to be extended with more complex compositional structures, such as an event handler or a fault handler. Furthermore, we also aim to develop a verification tool based on the rules presented in this paper.

References

[1] S. Alda and A. B. Cremers. Towards composition management for component-based peer-to-peer architectures. *Electr. Notes Theor. Comput. Sci.*, 114:47–64, 2005.

[2] M. Bisignano, G. D. Modica, and O. Tomarchio. Jmobipeer: A middleware for mobile peer-to-peer computing in manets. In *ICDCS Workshops*, pages 785–791. IEEE Computer Society, 2005.

[3] F. Bonchi, A. Brogi, S. Corfini, and F. Gadducci. A Behavioural Congruence for Web Services. In F. Arbab and M. Sirjani, editors, *Fundamentals of Software Engineering (FSEN'07)*, LNCS. Springer, 2007. To appear.

[4] BPEL4WS Coalition. Business Process Execution Language for Web Services (BPEL4WS) Ver. 1.1, 2003. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[5] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The runes middleware: A reconfigurable component-based approach to networked embedded systems. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, Berlin (Germany), Sept. 2005.

[6] G. Gehlen and L. Pham. Mobile web services for peer-to-peer applications. In *Proceedings of the Consumer Communications and Networking Conference 2005*, pages 427–433, 2005.

[7] R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman. Supporting predictable service provision in manets via context aware session management. *International Journal of Web Services Research*, (3):1–26, 2006.

[8] JXTA homepage. http://www.jxta.org/.

[9] R. Lucchi and G. Zavattaro. Wssecspaces: a secure data-driven coordination service for web services applications. In H. Haddad, A. Omicini, R. L. Wainwright, and L. M. Liebrock, editors, *SAC*, pages 487–491. ACM, 2004.

[10] P. Maheshwari, S. Kanhere, and N. Parameswaran. Service-oriented middleware for peer-to-peer computing. In *3rd IEEE International Conference on Industrial Informatics (INDIN '05)*, pages 98–103, 2005.

[11] D. McGuiness and F. van Harmelen (Eds). Web Ontology Language (owl) Overview, 2004. Web guide. http://www.w3.org/TR/owl-features.

[12] OWL-S Coalition. OWL-S: Semantic Markup for Web Services Version 1.1, 2004. http://www.daml.org/services/owl-s/1.1/overview/.

[13] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[14] SMEPP Coalition. D1.1 state of the art and generic middleware requirements, 2007. http://www.smepp.org/.

[15] W3C. Extensible Markup Language (XML) 1.0 (second edition), 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[16] W3C. Simple Object Access Protocol (SOAP) 1.2, 2001. http://www.w3.org/TR/2001/WD-soap12-part0-20011217/.

[17] W3C. Web Service Description Language (WSDL) version 1.1. http://www.w3.org/TR/wsdl.