International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:1, No:3, 2007

CoSP2P: A Component-Based Service Model for Peer-to-Peer Systems

Cándido Alcaide, Manuel Díaz, Luis Llopis, Antonio Márquez, Bartolomé Rubio and Enrique Soler

***Abstract*— The increasing complexity of software development based on peer to peer networks makes necessary the creation of new frameworks in order to simplify the developer's task. Additionally, some applications, e.g. fire detection or security alarms may require real-time constraints and the high level definition of these features eases the application development. In this paper, a service model based on a component model with real-time features is proposed. The high-level model will abstract developers from implementation tasks, such as discovery, communication, security or real-time requirements. The model is oriented to deploy services on small mobile devices, such as sensors, mobile phones and PDAs, where the computation is light-weight. Services can be composed among them by means of the `port` concept to form complex ad-hoc systems and their implementation is carried out using a component language called UM-RTCOM. In order to apply our proposals a fire detection application is described.**

***Keywords*— Peer-to-peer, mobile systems, real-time, Service-Oriented Architecture.**

I. Introduction

PEER-TO-PEER (P2P) systems [1] represent a new challenge in the development of software for distributed systems and an interesting alternative to centralized and client-servers models.

Each node in the P2P network is symmetrical and the mechanisms of communications are based on dynamic ad-hoc networks among peers. The P2P approach can be ported to mobile systems, which are composed of a set of nodes intended to cooperate, while they are entering/leaving the network continuously. Some mixed approaches keep a minimum part of the infrastructure and central control, but our proposal is focussed on pure ad-hoc systems. Examples of this type of system are the sensor networks [2], consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or environmental conditions; and MANET (Mobile Ad-hoc NETworks) [3], a self-configuring network of mobile devices such as PDAs and mobile phones connected by wireless links.

Due to the increasing complexity of the development of applications based on the P2P architecture, new high level programming models are necessary in order to simplify the developer's task. This paper presents a component-based service model to enable developers to specify p2p applications. The model allows us to define the services that peers can provide or require in the network. The access points to the peer services are the `ports` which define the commands and events that other peers require. P2P applications may need to define real-time requirements, for example, the establishment of a priority order to attend service queries. In this sense, the model allows us to include real-time constraints in the ports, that is, developers can assign characteristics such as service periods, priorities or deadlines to commands and events. This way, two services could be required simultaneously in the peer and priority would establish a ordered delivery. Additionally, the period would allow us to receive events without carrying out the invocation. The proposed model is platform independent and tools can be defined in order to map the specification to the necessary implementation, for example using Windows P2P Networking [4] on Windows platforms or JXTA [5] on Java platforms. The service implementation is carried out using the component based paradigm by means of the component language UM-RTCOM [6]. It proposes a distributed model based on light-weight component composition with real-time features. The UM-RTCOM specification is also platform independent and includes the possibility of defining real-time requirements.

The Service-Oriented Architecture (SOA) has been successfully applied in a lot of infrastructure-based and client-server model based distributed systems, being Web Services the best known standard [7]. Recently, some work based on the SOA paradigm oriented to P2P systems has appeared. The following list shows some related work. They mainly differ from our approach because either they do not provide a high level abstraction model that facilities the application programmer task or they are focused on infrastructure based systems:

- The JXTA plaform, is a set of simple, open source P2P protocols that enable any device in the network to communicate, collaborate, and share resources. It is a useful tool for implementing P2P systems but can be difficult to apply in complex systems without an abstraction model. It is a good platform to support a service model.
- JMobiPeer [8], developed in the University of Catania, proposes an alternative to JXTA more oriented to MANETs. It tries to solve the faults existing in the JXME version of JXTA at the time of development. However, it does not offer an abstraction model.
- DeEvolve [9] approaches the problem giving a flexible notation for the composition of services including the handling of exceptions such as failure of peers. It is based on JXTA and offers an exception handling mechanism to

This work is supported by the EU funded project FP6 IST-5-033563 and the Spanish project TIN2005-09405-C02-01.

C. Alcaide, M. Díaz, L. Llopis, A. Márquez, B. Rubio and E. Soler are with the Department of Languages and Computer Science, Málaga University, 29071, Spain.

C. Alcaide is the corresponding author to provide phone: +34 952 137147; fax: +34 952 131397. Authors e-mail: {calcaide, mdr, luisll, amarquez, tolo, esc}@lcc.uma.es.

control peer failures. It is oriented to infrastructure-based P2P systems.

- The middleware for wireless sensor networks (WSNs) presented in [10], provides a layer between user applications and the network. Such middleware offers an autmatic choice of the netowrk configuration and data dissemination strategy. It is implemented in Java and uses the XML language and the SOAP protocol to represent all application communications.
- The work proposed in [11] aims to build a middleware for WSNs that is based on the service-oriented architecture using Web Services and Grid technologies. They include probability models in order to improve the quality of service for service-oriented WSN applications.

In the following sections a concise explanation of the UM-RTCOM model (Section II) and a detailed vision of the proposed model (Section III) is presented. Additionally, in Section IV a fire detection system example is described using our approach. Finally, in Section V some conclusions and future work are sketched.

II. The UM-RTCOM Model

UM-RTCOM components are light-weight components which do not depend on any specific execution platform or heavy framework. Instead, UM-RTCOM components are developed in a platform independent way and are later deployed in specific platforms like executables or libraries with a minimum overhead. In addition, UM-RTCOM components are complemented by an abstract model of their behavior (based on SDL). This abstract model allows us to perform different types of analysis such as for example real-time analysis, deadlock freedom, liveness properties, etc. In this sense, a UM-RTCOM component is the sum of the code and abstract model.

The model improves some features of standard component models, adding constructions to express temporal constraints, synchronization, quality of service, events, etc. It is a hierarchical model where components act as containers of other components and, at the same time, provide interfaces.

A. Component Types

There are two main component types: primitive and generic. Generic components are the standard components of the model. They provide services through interfaces and can require services of other generic components. On the other hand, primitive components (active or passive) are contained in generic components. They are the basis for building generic components, representing execution threads or shared resources.

1) Generic Components: These components are the basis of the model. They act ascontainers of other components, generic or primitive, and they can be composed of other generic components in order to complete their functionality.

A generic component has a public definition part with the provided and required interfaces, and an implementation part which includes the implementation of the services offered. The model distinguishes between input interfaces (provided) and output interfaces (required).

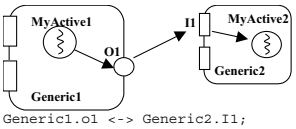
UM-RTCOM also allows events to be used through the declaration of produced and consumed events. In this programming-style, a component declares what events produce and what events consume. This way, components can communicate with each other without common interfaces.

2) Active Components: These components are primitive elements used to to express ”execution flows” inside a generic component. Concurrency is an important factor in real-time systems, so we use first-order elements to model it. In addition, the use of these components is also motivated by the later analysis phases.

Active components are responsible for the execution flow inside generic components through the interaction with other elements such as passive or generic components. Active components are also responsible for the treatment of the invocation requests to the generic container. Thus, the response to incoming requests is delegated to these primitive components.

The use of an Active component requires a definition part and a declaration part. In the definition part, the component defines a special ”execute” method to be invoked by the system in different ways: time-triggered, event-triggered, service requests, etc.

3) Passive Components: Shared resources are another important element in embedded and real-time systems. Passive components are primitive elements which allow us to use shared resources in the model. Basically, they cannot initiate any action and offer some basic services which can be invoked from active components. This behavior is used in order to facilitate later analysis phases. Passive components provide mutual exclusion with priority ceiling mechanisms which avoid priority inversions.




```
Generic1.o1 <-> Generic2.I1;

interface I1 {
      void method1(in short a1,
       in short a2,in short a3);
      void method2(in short a1);
}

Active MyActive1 {
  void execute() {
    short a1,a2,a3;
    a1=1; a2=2; a3=3;
    call o1.method1(a1,a2,a3);
    call o1.method2(a3);
    }
}

Active MyActive2 {
  void execute() {
    short var_aux;
    wait I1.method1(a1,a2,a3)
           {var_aux=a1+a2+a3; }
         I1.method2(arg1);
           { var_aux=arg1+1; }
    }
}
```

Fig. 1. Synchronization Primitives.

B. Component Interactions

Communication between components is performed through interfaces and events. UM-RTCOM provides synchronization primitives (wait, call, raise) which allow services and events to be invoked, raised or waited for.

- Wait Primitive: This primitive is used to waiting for new invocations on services or the creation of consumed events. It is used in Active components.
- Call Primitive: Primitive call is used to invoke services of generic components. It can be used in Active or Passive components.
- Raise Primitive: This primitive is used to raise events in an asynchronous way. When a component raises an event, this event is caught by all the components consuming that event.

Fig. 1 shows an example where two generic components (`Generic1` and `Generic2`) are interconnected. The example shows how the Active component `Generic1::MyActive1` calls the method `I1::method1` provided by `Generic2` where the request is attended to by `Generic2::MyActive2` using the wait primitive.

C. Real-time Constraints Specification

The user can indicate real-time constraints in the components. This is a very important element which is not included in other component models. This way, the user specifies the requirements regarding how a component is used.

The only elements visible of a component are the interfaces and events. The user can specify temporal constraints for methods or events indicating the minimum period between two invocations or a deadline for completing the request.

III. The CoSP2P Model

The proposed work is a service-based model, oriented to P2P systems without infrastructure, where developers can define the services to be offered by network nodes. It also specifies how these services can be composed in order to create distributed applications and declare the necessary ports so that each node has access to the services offered by other nodes.

Additionally, real-time constraints can be specified to the links between services in order to form systems with timing requirements in P2P environments.

Fig. 2 shows a graphical example of a service-composition scheme. When the port is facing inwards it means that it is a provided port. Optional ports are drawn in a dark color. Moreover, components and ports are connected by means of a dash-dotted lines. Although it is not shown in this figure, a peer can provide more than one service. In the example, the service offered by *Peer A* requires these different ports in order to be provided: the ports offered by *B* and *C* are required and the port offered by *D* is optional.

CoSP2P is an intermediate layer (Fig. 3) that allows applications to be developed composing services using components defined with the UM-RTCOM model.

The syntax to develop applications meeting the service paradigm is shown in Fig. 4, where ports, services and implementation declarations have been separated into blocks. In the following subsections a fuller explanation of the proposed syntax is presented.

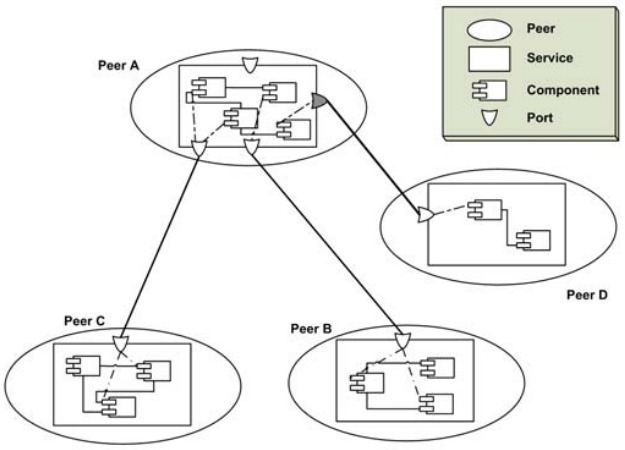


Fig. 2. Service Composition Example.

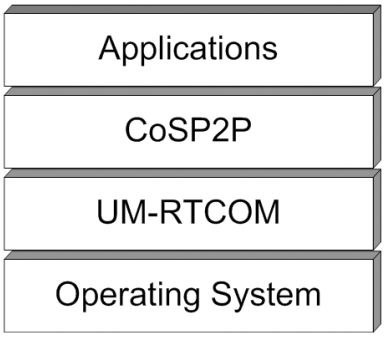


Fig. 3. The Abstract Layers.

A. Port Declaration

Ports are the access points to peers services. Additionally, each port is linked to the components that finally implement the commands and the events offered by it. Ports are also the mechanism used to compose the services offered to the system by the nodes. Inside a port definition, developers can define commands and events. Commands are synchronous communication mechanisms, and events are asynchronous communication mechanisms. For example, in our fire detection system (described in section IV), temperature sensors will communicate with the rest of the services in an asynchronous way by firing an event when a fixed temperature is reached.

Ports are global definitions to all the services, therefore no implementation details are associated with them. This way, different service offering a specific port can be implement in different ways: different execution platforms, programming languages, algorithms, etc. Bellow, a simple port is defined:

```
Port MyPort {
  command GetValue(out int value);
  event High;
}
```

```
Port <Port_name> '{'
   {command} [{command_real_time_constraints}]';'
   {event} [{event_real_time_constraints}]';'
 '}'

command ::= command <command_name> '(' {args} ')' ';'

event::= event <event _name> ';'

command_real_time_constraints::= constraints [Deadline T','] [Priority T]

event_real_time_constraints::= constraints [Period T','] [Priority T]

Service <Service_name> '{'
   Description <Service_description> ';'
   {provided_ ports }
   {required_ ports}
   {optional_ ports }
   {group_information}
'}'

provided_ports ::= Provides { Port_name } ';'

required_ports ::= Requires { Port_name } ';'

optional_ ports ::= Optional { Port_name } ';'

group_ information ::= Groups { Group_name } ';'

Service implementation <Service_name> '{'
   {components_definition}
   {port_binding}
   {component_composition}
'}'

components_definition ::= Components { component_name } ';'

port_binding ::= Port_bind '{'
   {bind}
'}'

bind ::=
   <port.ref_command> '=' <instance.ref_interface.ref_operation> ';' |
   <port.ref_event> '=' <instance.ref_interface.ref_event> ';'
```

Fig. 4. Model Syntaxis.

MyPort defines a command named *GetValue* that takes one output parameter, and an event called *High*. The real-time restrictions can be established to commands and events declared in a port declaration. These restrictions are specified by the keywords *Priority*, *Deadline* and *Period*.

- **Priority**: This primitive allows us to define the priority of the commands and events.
- **Deadline**: This restrictions defines the maximum execution time for a command.
- **Period**: This primitive allows us to establish the execution period for an event.

```
command MyCommand() constraints Deadline 5000,Priority 5;
event MyEvent constraints Period 1000, Priority 10;
```

The previous example shows the definition of a command and an event; *MyCommand* has five-second deadline constraint and priority of 5 is assigned, *MyEvent* has an execution period of one-second and priority of 10 is assigned.

B. Service Definition

A service definition is composed of: firstly, a service description followed by the ports that are going to take part in the service. The port classification is divided into three types: *provided_ports*, *required_ports* and *optional_ports*:

- **Provided ports**: Ports that the service offers to other services. The rest of the nodes can query to this service through the provided ports.
- **Required ports**: Ports that a service needs in order to be executed. Other nodes in the network must offer these services.
- **Optional ports**: These ports are associated with non crucial services, e.g. ports used to monitor the system.

Finally, it is possible to define communication groups so that a service can communicate with services of the same group. A service can belong to one or several groups.

The code below represents a service definition; firstly, we write a brief description, then, we define that the service provides one port (*MyPort*) and only one port is required (*OtherPeerPort*). In this example, the service has no optional ports. Finally, we specify that *MyService* belongs to the *ExampleGroup* group.

```
Service MyService {
  Description = "Example service";
  Provides MyPort;
  Requires OtherPeerPort;
  Groups ExampleGroup;
}
```

C. Service Implementation

When a service have been specified, its implementation must be declared:

1) Firstly, the components that implements the service are defined. The final implementation is carried out by the UM-RTCOM components.
2) The connection between ports and UM-RTCOM components must be specified. This is specified in the port binding declaration. In this block, port commands and events are linked to the component commands and events.
3) Since a service can be implemented by various components, it is necessary to declare how each component is composed with the rest. This is specified following the UM-RTCOM model syntaxis.

```
Service implementation MyService {
  Components AComponent compoment;
  Port_bind
  {
    MyPort.GetValue = component.GetValue;
    MyPort.High = component.High;
    OtherPeerPort.GetTemp = component.GetTemp;
    OtherPeerPort.TempHigh = component.TempHigh;
  }
}
```

IV. Example

In this section, we present a practical application that has been considered to test the component-based service model. Firstly, we show a brief description of the system, then we detail the different parts of our solution describing the design of the ports and services used and, finally, some implementation details are sketched.

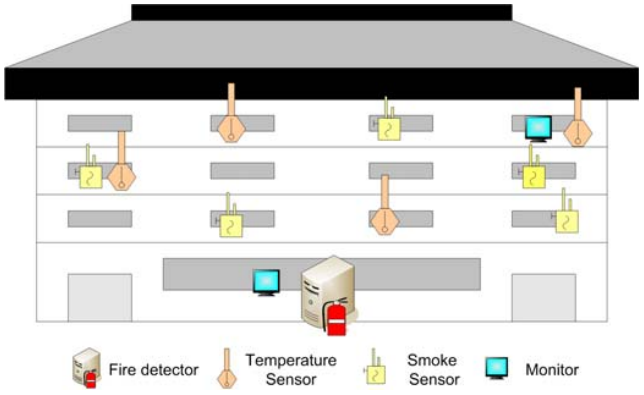


Fig. 5. Example System.

A. System Description

The example consists of a sensor and actor network for monitoring (in order to monitor and control fire in a building) (Fig. 5). We require four types of peers: the fire detector peer is an actor that is able to detect a possible fire and then act accordingly. The temperature-sensor and smoke-sensor peers provide temperature and smoke level measurements respectively. The monitor peer receives data about the actions carried out by the fire detector, updates the temperature and smoke threshold values used to trigger the corresponding events.

The system is composed of only one fire detector, but there could be one or more temperature/smoke sensors per room and any number of monitors. In order to provide the fire detection service, at least one temperature sensor and a smoke one smoke sensor must be available. Therefore, temperature and smoke ports are required monitoring ports will be optional.

The group creation can be done of many ways, for example one possible solution could be to form a group for each room and another for all monitors or simply one group for all communications. Finally, in our example we have proposed three kind of groups (Fig. 6):

- ***Temperature group***: where temperature sensors and the fire detector communicate with each other.
- ***Smoke group***: composed of smoke sensors and the fire detector.
- ***Monitor group***: the fire detector sends all monitor peers the information about building fire control.

B. Designing the Fire Detector System with CoSP2P

In order to create a system based on service composition, we must define the ports that will be provided by services, and it is also necessary to describe services and their implementations.

In the proposed system four ports exist, one for each service, although a service can usually provide more than one. For example a calculator service may provide two types of ports, one for standard calculus and another for a scientific one.

In the example below, the temperature port offers a command to obtain the temperature and raises an event when the temperature is higher than a given threshold. The smoke port is defined in the same way as the temperature port and the monitor port has only one command in order to get data displayed on the screen.

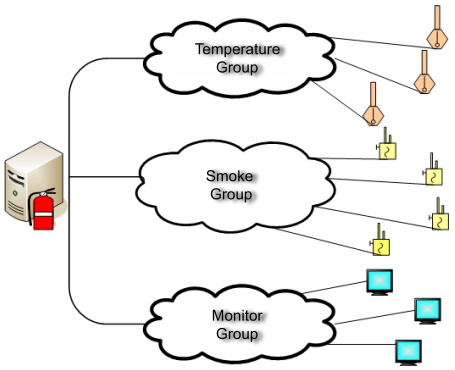


Fig. 6. Groups.

```
Port PTemperature {
   command GetTemp(out float data, out Unique id)
      constraints Deadline 5000;
   event TempHigh constraints Priority 5;
}
Port PSmoke {
   command GetSmoke(out float data, out Unique id)
      constraints Deadline 5000;
   event SmokeHigh constraints Priority 10;
}
Port PMonitor {
   command DisplayData(in String data)
}
```

As stated before, it is possible to define deadline constraints in commands, in this case, the commands *GetTemp* and *GetSmoke* have five seconds as maximum execution time. Furthermore, the priority constraint allows us to define the priority of the port commands and events. In the example, a priority of 5 is assigned to event *TempHigh* of the port temperature and a priority of 10 is assigned to event *SmokeHigh* of the port smoke.

The definition of the temperature service is really short, it include a brief description, it only provides the temperature port and specifies that it belongs to the temperature group.

```
Service STemperature {
   Description "...";
   Provides PTemperature;
   Groups TempGroup;
}
```

As we defined in subsection , the fire detector service requires that at least one port of each sensor type is available. The monitor port is optional. Moreover, this service belongs to all three groups.

```
Service SFireDetection {
   Description "...";
   Requires PTemperature, PSmoke;
   Optional PMonitor;
   Groups TempGroup, SmokeGroup, MonitorGroup;
}
```

Finally, the implementations of temperature service and fire detection service are the following:

```
Service implementation STemperature {
   Components Temperature temp, ...;
   Port_bind
   {
      PTemperature.GetTemp = temp.TempData;
      PTemperature.TempHigh = temp.TempHigh;
   }
   ... Component_composition ...
}

Service implementation SFireDetection {
   Components Control control, Alarm alarm, Sprinkler
              sprinkler;
   Port_bind
   {
      PFireDetection.CheckFire = control.CheckFire;
      PFireDetection.Fire = control.Fire;
      PTemperature.GetTemp = control.TempData;
      PTemperature.TempHigh = control.TempHigh;
      PSmoke.GetSmoke = control.SmokeData;
      PSmoke.SmokeHigh = control.SmokeHigh;
      PMonitor.DisplayData = control.DisplayData;
   }
   ... Component_composition ...
}
```

It is only necessary to include on the one hand, the list of components that are going to be bound with the service ports and, on the other hand, the component composition using the UM-RTCOM model as we described previously.

It is important to point out that it is possible to define different components to provide the same service.

C. Implementation

In order to test the proposed service model, a first prototype has been implemented. The model allows developers to define systems independently of platform and programming language. We decided to implement our application example using Windows P2P Networking, but it can also be realized on the JXTA P2P framework with the same design.

The application can provide the four services described. The Fig. 7 shows the fire detector screen with three temperature peers and three smoke peers connected. On the other hand, the Fig. 8 are screenshots of the other three peers; the temperature sensor and smoke sensor only show their sensor values, and the monitor peer contains a list with fire detector information (connection and disconnection of sensors, high values of temperature or smoke, fire detector actions, etc.).

The implementation has been tested on top of an ad-hoc network using a laptop computer for each peer.

Fig. 7. Fire Detector Screen.

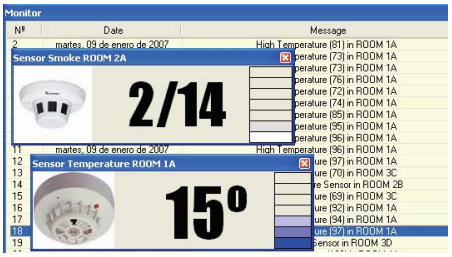


Fig. 8. Temperature Sensor, Smoke Sensor and Monitor Screens.

V. Conclusions and Future Work

In this paper we have presented CoSP2P, a novel component-based service model for P2P systems. We propose a syntax to describe this kind of systems independently of the programming languages or operative systems used. The model provides the possibility of including real-time constraints in order to form systems with timing requirements in P2P environments. Furthermore, with the description of a fire detector system, we have deployed an application using Windows P2P Networking in order to assess the model expressiveness.

As future work, we are currently extending the model syntax, as well as port and service definitions, including declarations of the peers and groups that can take part in a system. On the other hand, we intend to include an exception handler, to provide semantics to group formation and develop, using the proxyless version of JXME, a system described with the proposed model.

References

[1] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, Zhichen XU. Peer-to-Peer Computing. *Technical report, Hewlett-Packard*, 2002.

[2] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks Journal*, 38(4): pp. 393–422, 2002.

[3] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2(5): pp. 483-502, 2002.

[4] Introduction to Windows Peer-to-Peer Networking. URL: www.microsoft.com/technet/prodtechnol/winxppro/deploy/p2pintro.mspx

[5] Project JXTA. URL: http://www.jxta.org.

[6] Diaz M., Garrido D.,Llopis L., Rus F., Troya J.M. Integrating Real-Time Analysis in a Component Model for Embedded Systems. *Proceedings of the 30th IEEE Euromicro Conference*, pp. 14–21, 2004.

[7] Guido Dehlen, Linh Pham. Mobile Web Services for Peer-to-Peer Applications. Consumer Communications and Networking Conference, pp. 427–433, 2005.

[8] Mariano Bisignano, Giuseppe Di Modica, Orazio Tomarchio. JMobiPeer: a middleware for mobile peer-to-peer computing in MANETs. *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'05)*, pp. 785–791, 2005.

[9] Sascha Alda, Armin B. Cremers. Towards Composition Management for Component-based Peer-to-Peer Architectures. *Electronic Notes in Theoretical Computer Science 2005*, 114: pp. 47–64, 2005.

[10] Flávia C. Delicato, Paulo Pires, Jos Ferreira de Rezende, Luiz Rust da Costa Carmo, Luci Pirmez. Service-oriented Middleware for Wireless Sensor Networks. *Proceedings of the 2005 ACM symposium on Applied computing*, pp. 1155–1159, 2005.

[11] L.Q. Zhuang, J.B. Zhang, Y.Z. Zhao, M. Luo, D.H. Zhang, Z.H. Yang. Power-aware Service-oriented Architecture for Wireless Sensor Networks. *Industrial Electronics Society, 2005. IECON 2005. 32nd Annual Conference of IEEE*, pp. 2296–2301, 2005.