A Lightweight Middleware for developing P2P Applications with Component and Service-Based Principles

Ayoub Ait Lahcen, Didier Parigot

A Lightweight Middleware for Developing P2P Applications with Component and Service-Based Principles

Ayoub Ait Lahcen[a,b], Didier Parigot[a]
[a]Zenith Team, Inria Sophia Antipolis, Sophia Antipolis, France
[b]LRIT, Unité associée au CNRST URAC 29, Faculté des Sciences, Rabat, Morocco
Email: {ayoub.ait_lahcen, didier.parigot}@inria.fr

***Abstract*—Developing Peer-to-Peer (P2P) applications became increasingly important in software development. Nowadays, a large number of organizations from many different sectors and sizes depend more and more on collaboration between actors to perform their tasks. Since P2P applications are usually implemented as sets of strongly encapsulated functions, they can benefit from the advantages of component-oriented development. In the literature, there exists a large number of component based approaches. However, most of them are not adapted to P2P applications. In this paper, we present a middleware that combines component-oriented development with well-understood methods and techniques from the field of Service Oriented Computing (SOC) and P2P Computing in order to develop and deploy P2P applications in an effortless and effective way. This middleware is called SON (Shared-data Overlay Network).**

I. Introduction

Developing Peer-to-Peer (P2P) applications became increasingly important in software development. Nowadays, a large number of organizations from many different sectors and sizes depend more and more on collaboration between actors (individuals, groups, communities, etc.) to perform their tasks. P2P architecture is the concept of an entity acting at the same time as a server and as a client in P2P networks [1]. This is completely different to Client/Server networks, within which the participating entities can act as a server or as a client but cannot embrace both capabilities. Therefore, the responsibilities of entities are approximately equal and each entity provides services to each other as peers.

Since P2P applications are usually implemented as sets of strongly encapsulated functions, they can benefit from the advantages of Component-based Software Engineering (CBSE) [2]. The main purpose of CBSE is introducing minimum dependencies between software units (components) in order to promote their reusability and composition [3]. Thus, application developers can respond quickly and at low cost to new business needs. In addition, the reduction of dependencies allows software units to evolve separately during running time. In the literature, there exist a large number of component models. However, most of them are not adapted to P2P applications. One reason for this is that they require a client/server architecture. This implies that all peers have to access to a single provider (or a few central providers) and it thus might cause a bottleneck and central point of failure. Another reason is that they are based on communication protocols which are inadequate in P2P environment. For instance, P2P systems are not forced to operate using a Domain Name Service (DNS) because the peers might not have a permanent IP address.

In this paper, we present a middleware that combines CBSE with well-understood methods and techniques from the field of Service Oriented Computing (SOC) and P2P Computing in order to develop and deploy P2P applications in an effortless and effective way. This middleware is called SON (Shared-data Overlay Network).

SON middleware assists application developers by providing an automatic code generation which handles several requirements (e.g., communication mechanisms, message queue management, broadcasting messages, etc.). In fact, SON's user implements only the business code corresponding to the declared services. Afterwards, a code generation tool generates the corresponding components and their associated containers. The component container embodies all resources needed to adapt the implementation code to the P2P runtime environment.

SON can be considered as a generic lightweight P2P middleware (with the necessary set of operations that must be present to develop component and service-based P2P applications) for the following reason. Since, in most cases, the challenges of P2P systems can be reduced to a single problem: *"How do you find any given data item in a large P2P system in a scalable manner, without any centralized servers or hierarchy?"* [4], SON has been unified the notion of publish/subscribe: it uses a DHT (Distributed Hash Table) [5] not only to publish and subscribe data, but also to enable dynamic service publication, discovery, and deployment.

This paper is organized as follows. In the next section we present background information about two main aspects of the context in which this work has been carried out: SOC and P2P Computing. Section 3 describes the SON middleware. Section 4 presents a summary of a prototypical implementation. It shows how SON middleware can be used to support the development of P2P applications with component and service-based principles through SGT (Simple Georeferencing Tool), an application dedicated to collect, process and display georeferenced data. Section 5, presents related work. Finally, Section 6 concludes.

II. Background and concepts

A. Service oriented computing

SOC [6] is a paradigm that uses services as fundamental elements for developing applications. SOC is based on three actors: i) the Service Provider publishes on a Service Broker the service descriptions which specify both the available service operations and how to invoke them (e.g., network protocol that must be used for the invocation, software components required to establish the connection, etc.); ii) the Service Broker registers the service descriptions and references; and iii) the Service Consumer discovers the services by running a search on the Service Broker. It then establishes a connection with the provider to invoke the service operations.

SOC is an academic initiative that aims at extending service-oriented architecture (SOA) to manage and compose services in a flexible manner and it is organized on three levels:

- The first covers SOA with its minimum functions: publication, discovery and binding services.
- The second is the dynamic services composition. It is responsible for adapting the application at runtime (adding new features; control the execution of the component services and manage dataflow among them; adapting to a new context).
- The third covers the management functions necessary for the overall supervision of applications. It may permit complete visibility into individual business transactions, and deliver application status notifications.

The SOC concepts allow the development of modular and dynamic applications by supporting loose coupling and late binding between the software modules. However, these concepts are generally managed by the programmer and are implemented in the business logic. In Section 3, we show how these SOC concepts (except the third level functionalities) can be integrated into our component model while being separated from the implementation code (business code that implements the services).

B. P2P computing

The idea of P2P is applied in various contexts and P2P systems do not necessarily have several characteristics in common; neither do they have to rely on a fixed set of attributes. There are no major standardization initiatives that look at all aspects of P2P technology and computing. The term P2P is defined by its usage and unique formal definition of P2P computing does not exist [7]. However, there are a number of features many P2P systems share as introduced in the following well-known and academically accepted definitions:

- The Gartner Group [8] defines P2P computing as: *"characterized by direct connections using virtual namespaces, it describes a set of computing nodes that treat each other as equals (peers) and supply processing power, content or applications to other nodes in a distributed manner, with no presumptions about a hierarchy of control"*.

- A brief concise definition of P2P computing is given in [9]: *"a set of technologies that enable the direct exchange of services or data between computers"*.

- A more recent definition is given in [10]: *"The peer-to-peer (P2P) architectural style consists of a network of loosely coupled autonomous peers, each peer acting both as a client and a server. Peers communicate using a network protocol, sometimes specialized for P2P communication such was the case for the original Napster and Gnutella file-sharing applications. Unlike the client-server style where state and logic are centralized on the server, P2P decentralizes both information and control."*

These definitions highlight the following elements that are fundamental to P2P computing and common in describing P2P applications:

- Direct exchange of resources between peers;
- Each peer is independent and equivalent in functions;
- There are no center servers or controllers;
- Peers communicate using a network protocol.

In this paper, and in addition to these elements, we adopt the position proposed in [11]: *"One way to derive a definition of purpose that is more inclusive, flexible, and extensible is this: There are P2P technologies, and there is P2P computing"*. The P2P technologies allow peers to share resources and collaborate on computational tasks. This implies an abundance of supporting technologies, such as discovery, remote resource management, security and more. P2P computing is the use of P2P technologies. A resulting phenomenon is the creation of an overlay community (of peers/components) that collaborates through resource (data, services, ...) sharing. This is the immediate result and operational purpose of P2P computing.

As can be understood from the above definitions, the P2P system we define with SON is formed by establishing an overlay network between peers. Peers are represented by component instances. Each component instance acts both as a server (with its input services) and a client (with its output services). Each component instance is used to store resources (data) which are accessible through services. Each instance is connected to a bounded number of other instances and has a unique identifier, such as an IP address. As the network evolves, instances can continuously seek after new partners by implementing a specific algorithm such as Gossip algorithm [12]. Thus, the final structure of the P2P network depends on the kinds of these searching algorithms.

The underlying layer of SON provides to component instances the necessary storage space (like a DHT; cf. Section V-B) and communication mechanisms (like JXTA; cf. Section V-C). This separation between layers allows us to make only very weak networking issues at the high level description and defers the additional ones to the lowest level where they are needed.

III. The SON Middleware

It is expected that P2P applications would need to run in distributed and ubiquitous environments. In this context, application components must be able to communicate with each other through the network. In addition, they must be able to adapt according to their evolution and execution environment. We say that the application (architecture) is dynamic [13]. To meet these needs, we have developed a lightweight middleware called SON (Shared-data Overlay Network), which combines three powerful paradigms: CBSE, SOC and P2P Computing.

SON is composed of a component model and a connection model (see Figure 1). The component model defines how to create and validate components. The connection model provides not only local and distributed communication mechanisms, but also allows different peers to publish and search resources. In this context, a resource represents a component that provides or requires services, and a peer represents a set of locally interconnected components.

By using SON middleware, the user is able not only to specify applications in component-based service model, but also to perform an effective code generation. In fact, the user defines for each component a set of services (input, internal and output). Then, he only implements the code of the components, i.e., the methods that implement the defined services. Afterwards, a code generation tool, called Component Generator (CG), generates a set of Java source files that implement the container of the component. These Java files (see Figure 2) are compiled together with the implementation code to generate a standalone and ready-to-use component. We note here that the component container embodies all resources needed to adapt the implementation code to the run-time environment. In particular, the generated container embodies:

- mechanisms to instantiate, connect and run the component;
- a local facet for the business code developer who does not need to have a consistent knowledge about the underlying infrastructure;
- a server facet that is connected to the local facet with a facade;
- a facade that transforms the output invocations in the local facet to an output service call emitted by the server facet (and vice versa for the inputs);
- scheduling mechanisms to control the execution of the service invocation queue.

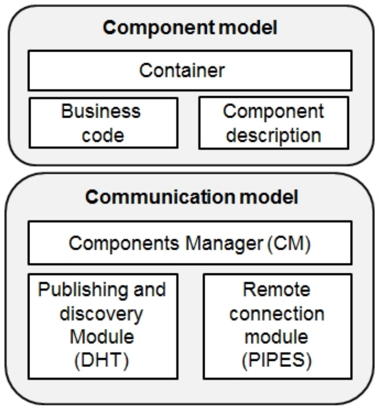
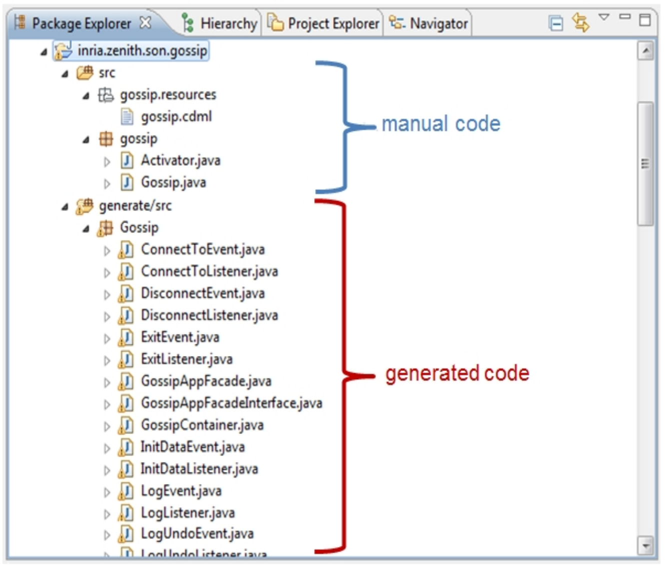


Fig. 1. Overview of SON middleware.



```
inria.zenith.son.gossip
  src
    gossip.resources
      gossip.cdml
    gossip
      Activator.java
      Gossip.java
  generate/src
    Gossip
      ConnectToEvent.java
      ConnectToListener.java
      DisconnectEvent.java
      DisconnectListener.java
      ExitEvent.java
      ExitListener.java
      GossipAppFacade.java
      GossipAppFacadeInterface.java
      GossipContainer.java
      InitDataEvent.java
      InitDataListener.java
      LogEvent.java
      LogListener.java
      LogUndoEvent.java
      LogUndoListener.java
```

Fig. 2. An example of SON's component structure.

Figure 3 shows the process that is followed to generate a component and its associated container.

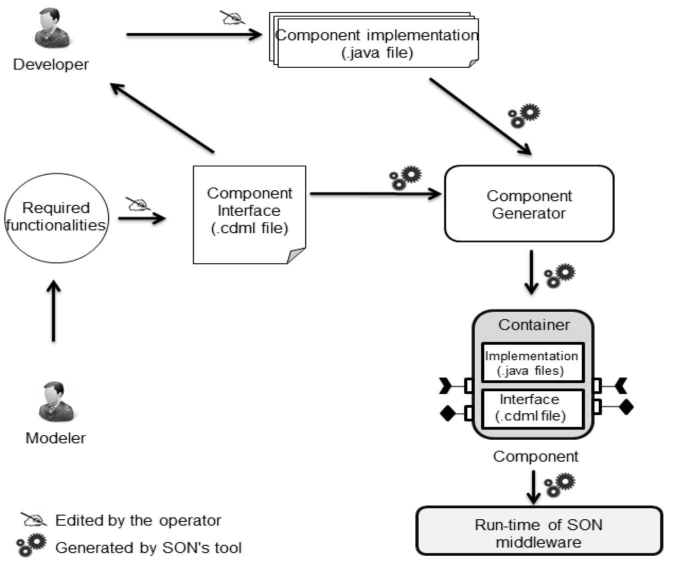


Fig. 3. Overview of the development process.

During the execution, a particular component runs by default. This component, called Component Manager (CM), supports the creation of component instances and establishes connections between them. To make the connection between two component instances, the CM uses their two interface description files to match the required and provided services. This matching works both ways.

There exists two configurations of SON infrastructure. The first configuration (local) can manage the local exchange between the component instances on the same peer. In this case, the CM manages locally a list of component instances. The second configuration allows managing the publishing and discovery of component instances in a P2P network. In this context, the CM delegates the management of remote instances lists to a DHT (Distributed Hash Table) [5]. A DHT is a distributed system that provides mechanisms to collectively manage a mapping from hash values (keys) to some kind of content (data values), without any centralized control or fixed hierarchy, and with a little human assistance. DHTs were introduced in the research community of P2P because, in most cases, the challenges of P2P systems (e.g., storage, connectivity, coordination of resources, etc.) can be reduced to a single problem, called *lookup problem* [4].

After the connection process, two component instances interact with each other directly without going through the CM (cf. Section V-C). The advantage of this environment is its dynamic aspect. In fact, during the execution, the component instances can dynamically join and leave the system over connections established on the fly. The next sections present the different aspects of this infrastructure in more details.

IV. Service-oriented component model

As presented in [14], service-oriented component approach help developers to build SOC applications by separating non-functional requirements from business logic. To implement such applications, one must take into account standards, code distribution, deployment of components and reuse of business logic. To cope with these changes, applications need to be more open, adaptable and capable of evolving. We present in this section a service-oriented component model based on: i) the component interface description, named CDML and ii) the deployment description, named World.

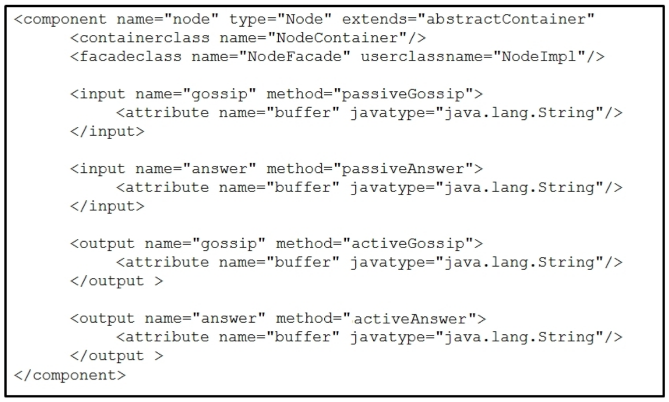
A. The component interface description (CDML)

We have defined an abstract Component Description Meta Language, i.e., independent from any component technology:

- To enable that the runtime environment can be taken into account without any modification to the business code.
- To enable that an interface can dynamically be discovered and adapted.
- To add meta-information to a component. This is a generic approach to record information dealing with several concerns such as deployment management and component behavior.

When these mechanisms are included, The Component Generator can automatically produce the non-functional code. That is to say the container that hides all the communication and interconnection mechanisms like the transformation of a service call by a sending message, the management of a queue of received messages, and the broadcasting of a message toward the connected components. Those runtime operations are totally transparent for the application designer.

As an example, a simple CDML of a component that implements Gossip protocol is given in Figure 4. Gossip protocol [12], also called epidemic protocol, is well-known in the community of P2P. It is mainly used to ensure a reliable information dissemination in a distributed system in a manner closely similar to the spread of epidemics in a biological community. This kind of dissemination is a common behavior of various P2P applications, and according to [12], a large number of distributed protocols can be reduced to Gossip protocol. To model this Gossip protocol, we consider a set of nodes, which get activated in each T time units exactly once and then spread data in a network by exchanging messages. Basically, when a node receives data, it responds to the sender and propagates the data to a subset of nodes selected according to a specific algorithm. In terms of service, a node is a component that has two activities: serving and consuming data. There are two input services for the serving activity (implemented by the methods *passiveGossip* and *passiveAnswer*) and two output services for the consuming activity (implemented by the methods *activeGossip* and *activeAnswer*).

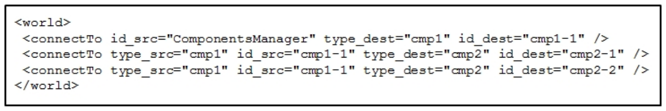


```
<component name="node" type="Node" extends="abstractContainer"
      <containerclass name="NodeContainer"/>
      <facadeclass name="NodeFacade" userclassname="NodeImpl"/>

      <input name="gossip" method="passiveGossip">
           <attribute name="buffer" javatype="java.lang.String"/>
      </input>

      <input name="answer" method="passiveAnswer">
           <attribute name="buffer" javatype="java.lang.String"/>
      </input>

      <output name="gossip" method="activeGossip">
           <attribute name="buffer" javatype="java.lang.String"/>
      </output >

      <output name="answer" method="activeAnswer">
           <attribute name="buffer" javatype="java.lang.String"/>
      </output >
</component>
```

Fig. 4. Simple CDML of a Gossip component.

B. The deployment description (World)

The deployment description file is used to describe the initial state of an application. It contains a description of the components and connections that have to be created by the CM to launch the application. Of course, after that, other components can ask to be connected with each other dynamically as explained in the next section. A component instance is identified by the couple (component name, instance name). For example, in Figure 5 the instance (cmp1, cmp1-1) corresponds to an instance of component cmp1.



```
<world>
 <connectTo id_src="ComponentsManager" type_dest="cmp1" id_dest="cmp1-1" />
 <connectTo type_src="cmp1" id_src="cmp1-1" type_dest="cmp2" id_dest="cmp2-1" />
 <connectTo type_src="cmp1" id_src="cmp1-1" type_dest="cmp2" id_dest="cmp2-2" />
</world>
```

Fig. 5. Example of a deployment description file.

V. P2P COMMUNICATION MODEL

A. The Components Manager (CM)

The Components Manager loads components, creates their instances and maintains a local list of them. To establish connections between two instances, the CM uses their CDMLs to connect output connectors (vs. input) of the first one with input connectors (vs. output) of the second one. When connected, the two component instances interact with each other directly without going through the CM (see Figure 6). Connection management, which includes creation or destruction of connection, occurs when the CM receives notifications announcing changes in the component registry. These mechanisms allow an application to be built as interconnected component instances which can adapt dynamically to their context. Thanks to the CM that monitors the execution context and acts on the components by managing their connections.

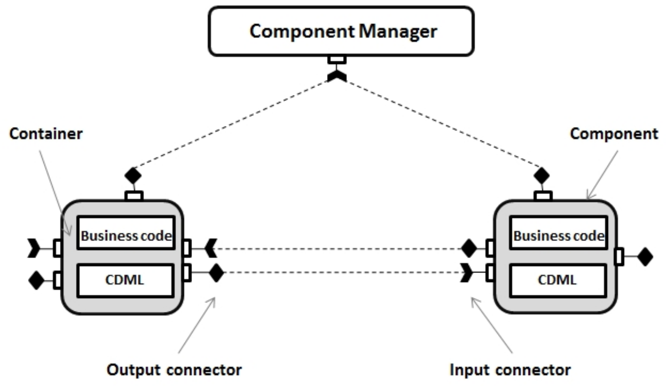


Fig. 6. Connection between instances of components.

In P2P mode, to know whether an instance is already created, the CM should not be limited to a local search. If the instance does not exist locally then the CM should also extend the search to all connected CMs. For better modularity and information management, the CM delegates the management of components and instances tables to the DHT module. The CM has a policy to choose the effective connection. For example, a policy will favor local connections over distributed connections. Moreover, the CM structure allows to instantiate different policies by using the Command design pattern [15]. In fact, The request to connect components is done in two steps. In the first step, the CM interrogates the local list and DHT module on the presence or not of the instance of the destination. Each one responds asynchronously to the CM. When the CM is in possession of all responses (even negative) then in the second step, it selects according to its policy the module that handles the effective connection. If in the first step, there is no positive response, the connection request is put on hold until the CM receives a notification, such as a component has been started or discovered.

To publish, discover and connect components on the network, two modules are proposed (see Figure 7). DHT module publishes and discovers components, and PIPES module connects components that are deployed on remote peers.

B. The DHT module

DHT module manages remote component lists. In the current version, DHT module uses the OpenChord implementation [16], but nothing prevents from using other implementations. For this purpose, an interface was defined with the usual methods (*put (key,value)* and *get(key)*) that can be expected from a DHT module. At each creation of a component instance, the CM publishes into the DHT, the necessary information used by remote PIPES modules to establish connection to this new created component instance.

C. The PIPES module

The PIPES module handles the communication between remote component instances. It opens a TCP connection between peers. It is based on the concept of virtual pipes introduced into the JXTA [17], a communication technology that has been widely used within the Grid community. This concept allows passing through a single TCP connection, several logical communications (virtual pipes) between peers. By using this abstraction, each component may open a virtual pipe to read messages sent to it. A virtual pipe is identified by a Universally Unique Identifier (UUID). This identifier is associated with the component instance name and registered in the DHT as follows:

[Key: component instance Name, Value: UUID of the virtual pipe]

[Key: UUID of the virtual pipe, Value: UUID of the PIPES module]

[Key: UUID of the PIPES module, Value: IP + Port Number]

The second record associates the virtual pipe component with the PIPES module it belongs. The third record associates the PIPES module with its IP address and port number. Thus, two peers can find into the DHT all the information needed to connect their components.

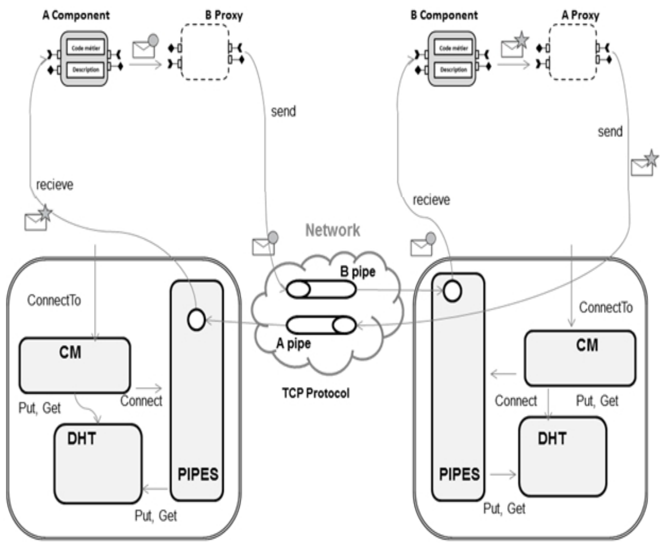


Fig. 7. Run-time architecture of SON middleware.

VI. Implementation

This approach has been fully integrated into the Eclipse environment [18] and implemented on top of OSGi [19]. Eclipse is built around a very small extensible runtime core and its functionality, (including compilers, workbench, and support tools) consists of plug-ins that can be managed separately. That allowed us to integrate the Component Generator (CG) into Eclipse as a plug-in.

The application programmer develops his Java code with Eclipse IDE, in the classic way. Then, after defining the CDMLs, non-functional codes are generated using the CG plug-in to obtain components usable by the SON middleware (see Figure 2). The OSGi service platform provides a computing environment for applications, called bundles, to dynamically deploy services in a centralized environment. It is also a small layer that allows multiple components to efficiently cooperate in a single Java Virtual Machine (JVM) by managing aspects of local service deployment. However, OSGi service platform leaves service dependency management as a task for component developers, thing which is treated automatically in our case by the CM.

At the start of execution, the OSGi platform is launched, and the CM is started by default as a bundle. In this context, two OSGi services are used and published. The first one, called *ContainerService*, allows publishing the CDML when a component is started. The CM then adds that started component to its table of available components. The second one, called, *ContainerProxy*, allows publishing the component instance when it is created. The CM then adds that new instance to its table of created instances. The CM can then manage the execution in an extended environment unlike other classic Java application environments. Moreover, installing a new bundle, registering a new service, or updating an existing component does not need a restart of the JVM because the concerned components are notified of the new state and adapt their connections accordingly through the CM.

VII. Application

In this section we illustrate the practical use of SON middleware with an application called SGT (Simple Georeferencing Tool).

SGT is a simple prototype implemented as an application of SON middleware. It is only composed of three SON's components. SGT is dedicated to collect, process and display georeferenced individual level data. Georeferencing is relating information to geographic location [20] and its scope includes the informal means of referring to locations, which we use in ordinary discourse using placenames, and the formal representations based on longitude and latitude coordinates and other spatial referencing systems.

The application of georeferencing extends to almost all fields of human activity, including medicine, agriculture, petroleum exploration, government administration and historical research.

Georeferencing tools include services to identify a location of a place, object or person, such as discovering the nearest gas station or the whereabouts of a colleague or friend. They include package and vehicle tracking services, location-based games and even marketing services. In our case, we have chosen to explain our simple georeferencing tool SGT by using it as a geo-recommendation application as described in the following scenario.

A. *Using SGT for Geo-recommendation*

In cities all over the world, people search to discover new places, to describe their impressions and to share their discoveries with their colleagues, family, and friends. SGT is used to create and display a combined view of surrounding addresses along with recommendations based on the experiences and tastes of other persons. Thus, when SGT users are far from home and need information about new places (restaurants, movie theaters, museums, gyms, etc.), SGT's search engine can helps them with the recommendations of locals in the surrounding area.

In this experimental scenario, SGT implementation consists in three SON's components: *Provider*, *Consumer* and *Super-node*. *Provider* component instances are used to expose the georeferenced services to the network, while *Consumer* component instances are used by service consumers. Each *Super-node* instance is responsible for serving a certain number of *Provider* and *Consumer* instances by publishing georeferenced services and the associated recommendation, answering queries, and creating notifications. *Super-node* component embodies the functionalities of SON's communication model (see Section V). Thus, and instead of using a central server as the case of most georeferencing tools, *Super-node* instances form an overlay network based on a DHT that offers a reliable, robust and scalable mechanism to store and manage data using P2P principles. As indicated in Section V-B, we use OpenChord as a DHT implementation.

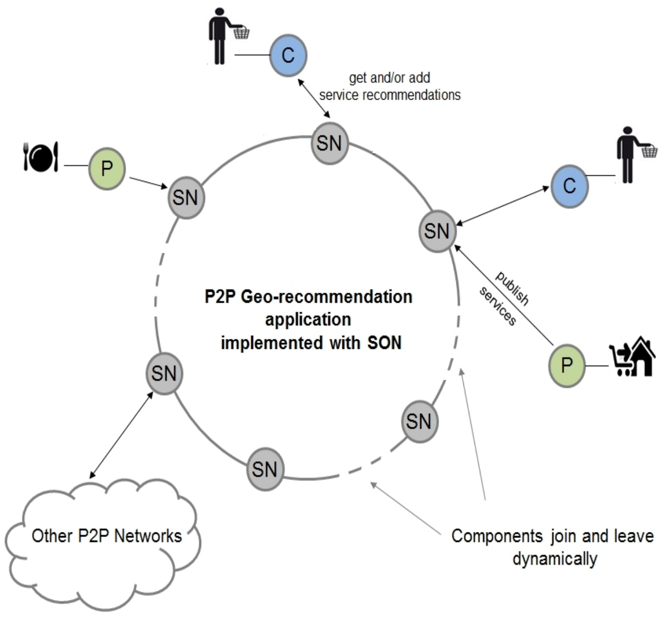

C
get and/or add service recommendations
SN
P
SN
C
SN
publish services
P2P Geo-recommendation application implemented with SON
P
SN
SN
SN
Other P2P Networks
Components join and leave dynamically

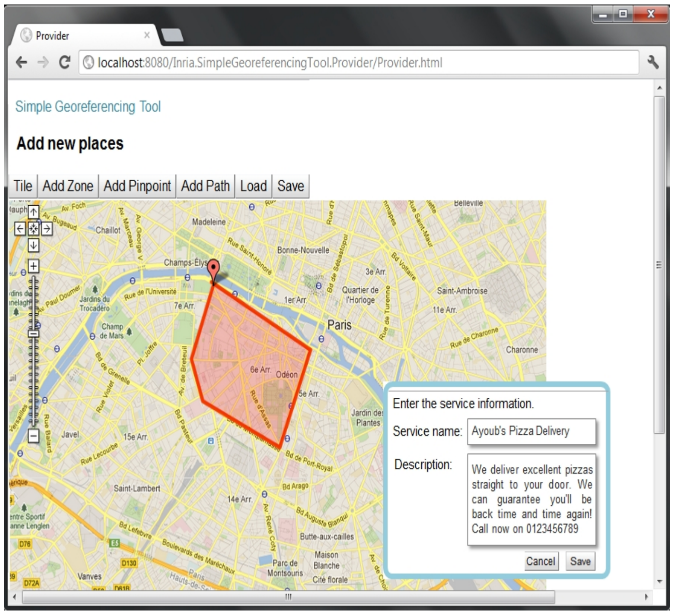
Fig. 8. Using SON to implement a geo-recommendation application.

Figure 8 gives a simple use case where provider users (a restaurant and a shop) use *Provider* component instance (denoted by p) to publish their georeferenced services, while consumer users use *Consumer* component instance (denoted by c) to get and add recommendations about those services. *Provider* and *Consumer* instances connect to the network through *Super-node* instances which are their access points.

Provider users are required firstly to add (through *Provider GUI*, see Figure 9) new places on the map and submit some information about the services available in those places. Places on the map can be a local, work zone, district, path, department, etc. The service information contains a name and a brief description.

After that, for each georeferenced service, a *key-value* pair is stored in the DHT. The *key* is calculated depending on the longitude and latitude coordinates of the region where the service place is located. The *value* of a *key* has the following form: *serviceInfo, point, point, point, ... point*, where each *point* corresponds to the longitude and latitude coordinates of the corners of the polygon representing the service place.

Thus, a consumer user can discover the available services around him by running queries in the DHT through the *Consumer GUI* (see Figure 10). Afterwards, the consumer user can add his own recommendations about a service. He can also subscribe to a desired service and receive notifications about new recommendations added by other people.


Provider
localhost:8080/Inria.SimpleGeoreferencingTool.Provider/Provider.html
Simple Georeferencing Tool
Add new places
Tile
Add Zone
Add Pinpoint
Add Path
Load
Save
Paris
Cancel
Save

Enter the service information.

Service name: Ayoub's Pizza Delivery

Description: We deliver excellent pizzas straight to your door. We can guarantee you'll be back time and time again! Call now on 0123456789

Fig. 9. Screenshot of *Provider GUI*.

We close by pointing out that the front-end part (*Provider* and *Consumer GUI*) has been generated from a Java Servlet using Google Web Toolkit (GWT) [21]. Java Servlet is a server-side web technology that serves user requests and receives responses from the business code of the component. GWT is a development toolkit for building complex browser-based applications without the developer having to be an

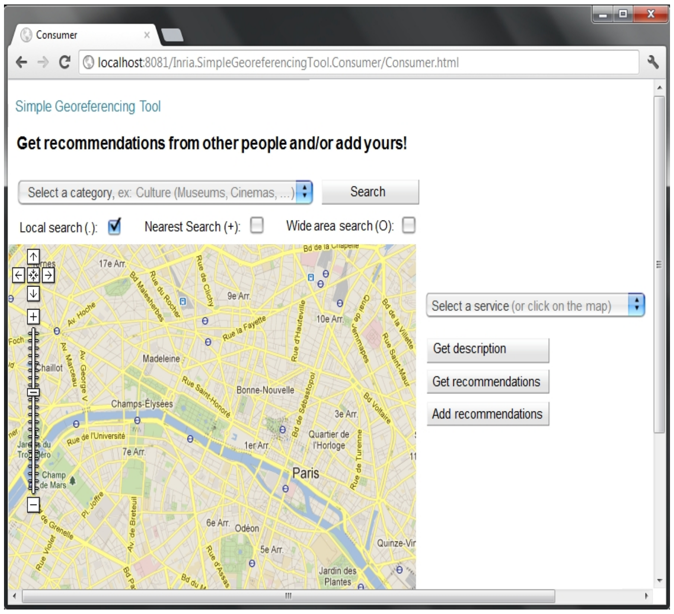

Consumer
localhost:8081/Inria.SimpleGeoreferencingTool.Consumer/Consumer.html
Simple Georeferencing Tool
Get recommendations from other people and/or add yours!
Select a category, ex: Culture (Museums, Cinemas, ...)
Search
Local search (.):
Nearest Search (+):
Wide area search (O):
Select a service (or click on the map)
Get description
Get recommendations
Add recommendations
Paris

Fig. 10. Screenshot of *Consumer GUI*.

expert in browser technologies (e.g., JavaScript, AJAX and XMLHttpRequest). GWT cross-compiler translates the Java source code to standalone JavaScript files that are deeply optimized. These allow SON's components to easily provide a web user interface that runs across all browsers, including those for mobiles.

VIII. Related Work

There has been a large body of related work carried out to develop P2P middlewares. This has proposed increasingly novel approaches addressing application from many different domains such as distributed sharing of data, video streaming and gossip communications. For example, JavaPorts framework [22] aims to provide a set of tools that will enable developing parallel applications on a network of heterogeneous workstations. A JavaPorts application can be defined as a collection of interacting tasks using a Task Graph abstraction. In this graph the nodes correspond to application Tasks. Tasks communicate using point-to-point connections between peer ports. Expeerience [23] is a middleware providing support for mobile application developers exploiting P2P technology over ad hoc networks. It has been developed in Java and is based on JXTA. It manages the discovery service, multiple interfaces, intermittent connectivity and code mobility. SpiderNet [24] is a P2P service composition framework. It achieves service composition by supporting directed acyclic graph composition topologies and considering exchangeable composition orders. SpiderNet provides failure recovery scheme that maintains a small number of dynamically selected backup compositions to achieve quick failure recovery for realtime streaming applications. Juno [25] is a networking middleware dedicated to multimedia content distribution (e.g., file sharing, video on demand and live streaming). It is designed in a component-based manner and has been implemented using the OpenCOM [26] component model. Juno provides a configurable framework, allowing the middleware to be specialised and adapted to a variety of environments. Kompics [27] is a message-passing component model that can be used for building P2P systems. Kompics provides a framework to compose protocol layers in a similar way to Mace [28] and Wids [29]. Mace is a language support for building distributed systems as C++ components. It allows describing each layer of the distributed system as a reactive state transition model. This state transition model enables model checking of the system implementation to find both safety and liveness bugs. WiDS is a toolkit that provides several run-times to run P2P protocols in different modes. In particular, in its simulation engine that helps to evaluate and debug P2P protocols in a controllable environment.

The main characteristics that distinguish SON from the approaches outlined above can be summarized as follows: i) SON's user implements only the code corresponding to the declared services. Afterwards, a code generation tool generates the containers of the components. The component container embodies all resources needed to adapt the implementation code to the P2P run-time environment. ii) SON can be considered as a generic lightweight middleware for the following reason. Since, in most cases, the challenges of P2P systems can be reduced to a lookup problem, SON has been unified the notion of publish/subscribe by using a DHT not only to publish and subscribe data, but also to enable dynamic service publication, discovery, and deployment.

IX. Conclusion

This paper presents a P2P middleware called SON. SON enables to perform an effective code generation. Thus, software developers are assisted and have greater ease in developing component and service-based P2P applications. These facilities allow them to focus more on the business logic and defer to SON the management of the runtime requirements (e.g., communication mechanisms, instantiation and connection of components, service discovery, etc.). In our research team, SON is already used to support complex application development, as P2Prec, a social-based P2P recommendation system for large-scale data sharing. Although we have shown SON to be highly useful, interesting areas of future work exist. In particular, we consider providing support for non-functional concerns such as fault-tolerance, QoS, and resilience.

References

[1] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," *Peer-to-Peer Computing, IEEE International Conference on*, vol. 0, p. 0101, 2001.

[2] C. Szyperski, *Component Software : Beyond Object-Oriented Programming*. New York: ACM Press and Addison-Wesley, 1998.

[3] G. T. Heineman and H. M. Ohlenbusch, "An evaluation of component adaptation techniques," in *In 2nd ICSE Workshop on Component-Based Software Engineering*, 1999.

[4] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in p2p systems," *Commun. ACM*, vol. 46, pp. 43–48, February 2003.

[5] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a dht," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[6] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis, "Analysis of recursive state machines," *ACM Trans. Program. Lang. Syst.*, vol. 27, pp. 786–818, July 2005.

[7] A. Mauthe and D. Hutchison, "Peer-to-peer computing: Systems, concepts and characteristics." *Praxis der Informationsverarbeitung und Kommunikation*, vol. 26, no. 2, pp. 60–64, 2003.

[8] Gartner Research Group, "The emergence of distributed content management and peer-to-peer content networks," 2001.

[9] M. Hofmann and L. R. Beaumont, *Content Networking: Architecture, Protocols, and Practice (The Morgan Kaufmann Series in Networking)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[10] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[11] D. Barkai, *Peer-To-Peer Computing: Technologies for Sharing and Collaborating on the Net*, ser. Engineer-To-Engineer. Intel Press, 2002.

[12] M. Jelasity, "Gossip," in *Self-organising Software*, ser. Natural Computing Series, G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos, Eds. Springer Berlin Heidelberg, 2011, pp. 139–162.

[13] P. K. McKinley, S. S. Masoud, E. P. Kasten., and B. H. C. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[14] J. Liu, J. He, and Z. Liu, "A strategy for service realization in service-oriented design," *Science in China Series F: Information Sciences*, vol. 49, pp. 864–884, 2006.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Publishing, 1995.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," pp. 149–160, 2001.

[17] B. J. Wilson, *JXTA*. New Riders, Jun. 2002.

[18] The Eclipse Foundation, *Eclipse Platform Technical Overview*, February 2003.

[19] The OSGi Alliance, "OSGi service platform core specification," http://www.osgi.org/Specifications, May 2007.

[20] L. L. Hill, *Georeferencing : the geographic associations of information*, ser. Digital libraries and electronic publishing. Cambridge, Mass.: MIT Press, 2006.

[21] GWT, "Google Web Toolkit - Build AJAX apps in the Java language," http://code.google.com/webtoolkit/, 2007.

[22] E. S. Manolakos, D. G. Galatopoullos, and A. Funk, "Component-based peer-to-peer distributed processing in heterogeneous networks using java ports," in *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 234–.

[23] M. Bisignano, A. Calvagna, G. Modica, and O. Tomarchio, "Expeerience: a jxta middleware for mobile ad-hoc networks," in *Peer-to-Peer Computing, 2003. (P2P 2003). Proceedings. Third International Conference on*, sept. 2003, pp. 214 – 215.

[24] X. Gu, K. Nahrstedt, and B. Yu, "Spidernet: an integrated peer-to-peer service composition framework," in *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, june 2004, pp. 110 – 119.

[25] G. Tyson, A. Mauthe, T. Plagemann, and Y. El-khatib, "Juno: Reconfigurable middleware for heterogeneous content networking," in *In Proc. 5th Intl. Workshop on Next Generation Networking Middleware (NGNM), Samos Islands, Greece*, 2008.

[26] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, pp. 1:1–1:42, March 2008.

[27] C. Arad and S. Haridi, "Kompics: a message-passing component model for building distributed systems," 2010.

[28] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: language support for building distributed systems," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 179–188.

[29] S. Lin, A. Pan, R. Guo, and Z. Zhang, "Simulating large-scale p2p systems with the wids toolkit," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, sept. 2005, pp. 415 – 424.