Programme Réseaux du Futur et Services (VERSO)

DataRing Project

Title: P2P Data Sharing for Online Communities

D1.3: Revised DataRing Architecture

31 dec. 2010

Partner in charge of deliverable: ATLAS (INRIA)

Contributors: GEMO (INRIA), LIG, LIRMM, Telecom ParisTech

Abstract. In this report, we present the revised DataRing architecture. The major revision is a more detailed definition of the DHT layer, called Shared-data Overlay Network (SON), which provides DHT and gossip communication for DataRing services. We also identified a new advanced service for P2P recommendation, P2Prec.

1. Introduction

In this deliverable, we present our revision of the DataRing architecture and services. The major revision is a more detailed definition of the DHT layer, called Shared-data Overlay Network (SON), which provides DHT and gossip communication for DataRing services. We also identified a new advanced service for P2P recommendation, P2Prec (see Deliverable D5.2: Demo of replication, caching and indexing services).

SON (http://www-sop.inria.fr/teams/zenith/SON) is based on a set of basic concepts for developing and deploying in a simple and effective way multiple services (e.g. directory, query, summary or recommendation). SON is an open source development platform for P2P networks using web services, JXTA and OSGi. SON combines three powerful paradigms: components, SOA and P2P. Components communicate by asynchronous message passing to provide weak coupling between system entities. To scale up and ease deployment, we rely on a decentralized organization based on a DHT for publishing and discovering services or data. In terms of communication, the infrastructure is based on JXTA virtual communication pipes, a technology that has been extensively used within the Grid community.

In this report, we first describe the revised DataRing Architecture, with SON and P2Prec. Then, we describe SON's architecture.

2. Revised DataRing Architecture

DataRing has a layered service-based architecture (see Figure 1). Besides the traditional advantages of using services (encapsulation, reuse, portability, etc.), this enables DataRing to be network-independent so it can be implemented over different structured (*e.g.* DHT). The main reason for this choice is to be able to exploit rapid and continuing progress in such networks. Another reason is that it is unlikely that a single network design will be able to address the specific requirements of many different community applications. In Figure 1, we call "local data" the data managed by the participants, i.e. repository metadata, data, and data sources. Local data is managed by a local DBMS.

Then, the three main layers in DataRing are, bottom up: SON, data services, and DataRing services.

SON

SON consists of an overlay network and a distributed storage layer. It also includes There is the Peer communication (Peercom) service which we add in the first version of the DataRing architecture. It enables peers to exchange messages. It also allows a peer to call a remote service, *e.g.* a Web service using SOAP, which is provided by another peer over the P2P network. We describe the main layers below. The overlay network layer is on top of Internet (TCP/IP) and in charge of routing. It implements the DHT *lookup()* function and manages peers dynamic behavior (joins/leaves of peers). The distributed storage layer provides key-based data searching and data distribution by implementing the put() and get() functions. This DHT layer can be supported by different DHTs, such as Chord or Free Pastry. Typically, a DHT maps a key k to a peer P called *responsible for k with respect to a hash function h*. Peers maintain information about $O(logN)$ other peers in a *finger table* and resolve lookups via $O(logN)$ messages to other peers. In addition, SON provides support for gossiping messages and call web services (WS).

Data services

The data services are the following. The DS access service provides all other DataRing services within a participant the ability to access the data source, typically through a wrapper interface. As in data integration systems, we must have such wrapper interface implemented for each different kind of data source. However, we will use the same(s) DBMS for the data sources.

The data privacy service provides purpose-based disclosure and trust control over shared data. All data accesses from the search and query service which need to respect data privacy use this service.

The replica and cache management service enables participants to replicate and cache DataRing data (or DS data which have been integrated) and propagate updates to replicas with consistency guarantees.

DataRing services

The DS discovery and integration service helps discover data sources in a given network (e.g. a corporate network) and their relationships in terms of semantics and schema mappings. Such service is very important to ease the activity of administrating DataRing and automate schema integration using approximate mappings. This service is useful to the search and query service to help reformulate queries using schema mappings.

The search and query service provides basic keyword search and more structured query processing functionality. This service also performs translation of keyword search queries into more structured queries when schemas are available for the DS discovery service.

P2Prec is a recommendation service for P2P content sharing systems that exploits users' social data. To manage users' social data, we rely on Friend-Of-A-Friend (FOAF) descriptions. It combines efficient DHT indexing to manage the users' FOAF files with gossip robustness to disseminate the topics of expertise between friends

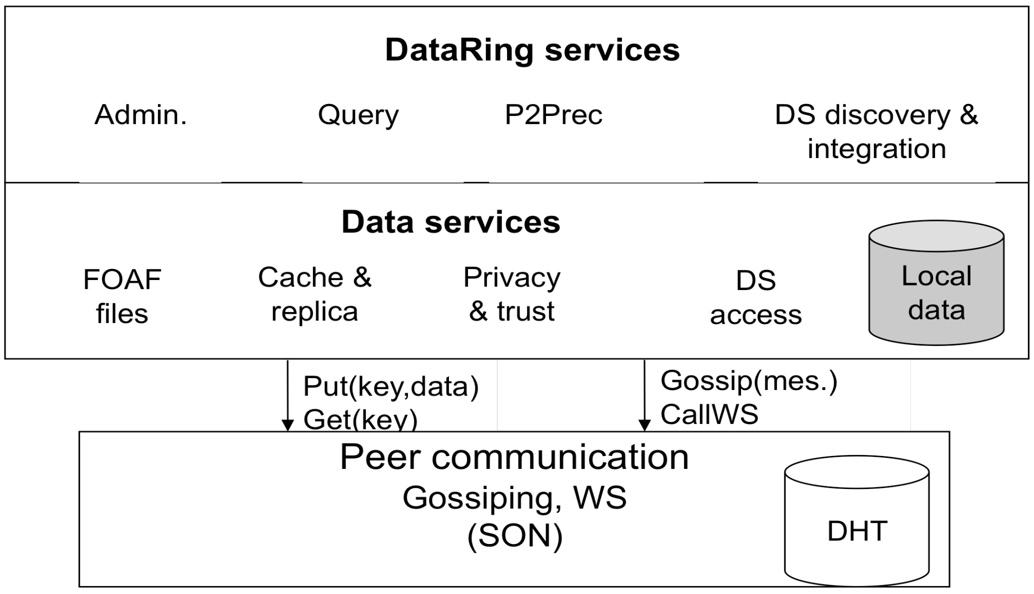

Figure 1. Revised DataRing Layered Architecture

3. Shared-data Overlay Network (SON)

Using SON, the development of a P2P application is done through the design and implementation of a set of components. Each component includes a non-functional code that provides the component services and a code component that provides the component logic (business code). The complex aspects of asynchronous distributed programming (non-functional code) are separated from code components. The Component Generator (CG) automatically generates this non-functional code from a description of services (provided or required services) for each component. This CG component is not present at the execution of the SON infrastructure.

The SON infrastructure (see Figure 2) is composed of a Component Manager (CM), a publishing and discovery module of component (DHT), and a connection module (PIPES). The Component Manager defines how to load and runtime components, the publishing and discovery module allows to publish or discovery components on different peers. The connection module provides connection between remote components on peers.

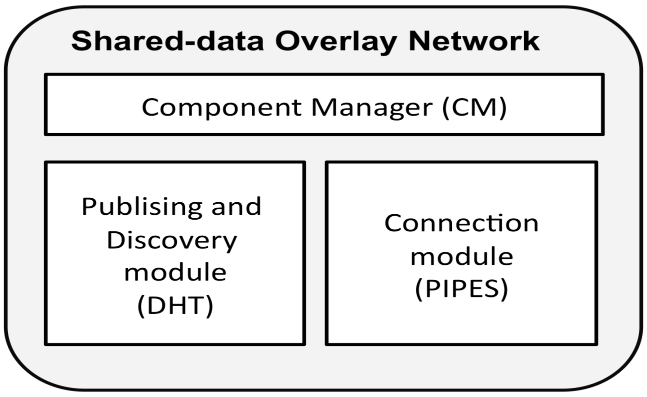
Shared-data Overlay Network
Component Manager (CM)
Publising and Discovery module (DHT)
Connection module (PIPES)

Figure 2. Overview of Shared-data Overlay Network Architecture

At run-time the Component Manager (CM), run by default performs the creation of new component instance and the connections between them. To establish a connection between two components, the CM uses the services description of each component.

They exist two configurations of this SON infrastructure. The first configuration (local) can manage the local exchange between the components, on the same peer. The CM manages locally a list of components. The second configuration allows managing the publishing and discovery of components in a network. In this context, CM delegates the management of lists of remote components to a Distributed Hash Table (DHT). The connection module is used for open the remote connections between remote components. These two configurations offer a great flexibility at our infrastructure. In fact, these configurations can be called dynamically at run-time.

SON is implemented in Java on top of OSGi technology (http://www.osgi.org/) which provides all basic services for the lifecycle of our components, in particular, the deployment services. SON uses the dynamic loading of bundles (components) of OSGi. It is composed of a set of bundles and thus launches as an OSGi configuration. In addition, many OSGI developments are directly usable by our platform. Concerning the Component Generator, it has been integrated into Eclipse environment (http://www.eclipse.org/) as a plugins. The programmer develops his Java code with the IDE Eclipse, in classic way. Then, after defining the services description, non-functional code can be generated using this CG plugins in order to obtain a component usable by the SON infrastructure.

The component model

This component model is based on a description of services required and provided of component. The Services description of a basic Gossip component is given in Figure 3. The `input` keyword corresponds to a provided service definition, and `output` keyword to a required service definition. The Component Generator (CG) automatically generates an equivalent description in Web Services format (WSDL) when generating the non-functional code from the services description of the component.

```
<component name="gossip" type="component" extends="AbstractContainer" ns="gossip" >
<containerclass name="GossipContainer"/>
<facadeclass name="GossipFacade" userclassname="Gossip"/>

<input name="gossip" method="passiveGossip">
   <attribute name="info1" javatype="java.lang.String"/>
</input>

<input name="answer" method="passiveAnswer">
   <attribute name="info1" javatype="java.lang.String"/>
</input>

<output name="gossip" method="activeGossip">
  <attribute name="info1" javatype="java.lang.String"/>
</output>

<output name="answer" method="sendAnswer">
  <attribute name="info1" javatype="java.lang.String"/>
</output>

</component>
```

Figure 3. Service description of the Gossip component

The component model is based on the generation of a non-functional code (container) according to the services description. This generation extends the component code written by user. This generation offers free to users, a programming style as multi-agent programming. This programming style is usually recognized to facilitate distributed application development. At each time, only one message (message received) is treated by the component. This run-time model avoids the common problems of concurrent programming.

Component Manager

At run-time, when a component (A) wants to connect with another component (B), then A component uses the service `ConnectTo(A,B)` provided by the CM. As is the CM that created the component, the components are by default connected to the CM. To establish a connection between two components, the CM uses the services description for associate the services provided of first component at services required of second component, and vice versa. After the connection process, the two components communicate directly with each other without going through the CM. A basic example of association is given in Figure 4.

The advantages of this association of services is be perform at run-time without that each component knows statically the services of other components. In fact, each component can, on the fly, connect to any component. The assembly of components of a given application is not necessarily known statically and can evolve dynamically over time. The components are autonomous and independent.

Through these mechanisms, it is possible to build applications with only the necessary components and simplify interconnection with local or remote components. This generation extends the user code and hide all the mechanisms of communication by asynchronous messages such as: 1) transformation a method call by a message sending; 2) managing the queue of messages received; and 3) broadcasting a message to a set of components. These mechanisms are completely transparent to the designer of the application. Indeed, to access at all SON infrastructure functionalities, the user uses only this `ConnectTo` service/method in his business code.

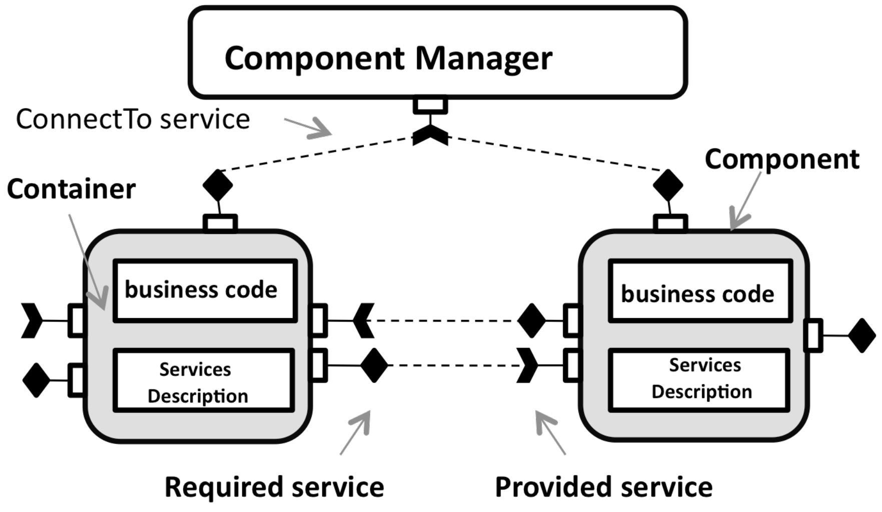

Figure 4. Model of a component

Deployment descriptor

The deployment description file is used to describe the initial state of the application. It describes instances of components and connections to be created by the Component Manager to launch the application. Of course, after this, other components can ask to be connected to each other dynamically by the `ConnectTo` service. A component instance is identified by the pair (`type_[src,dist]`=component name, `id_[src,dist]`=instance name), see an example in Figure 5.

```
<world>
 <connectTo id_src="ComponentsManager" type_dest="gossip" id_dest= "XXX"/>
 <connectTo id_src="ComponentsManager" type_dest="gossip" id_dest= "YYY"/>
 <connectTo id_src="XXX"  id_dest= "YYY"/>
</world>
```

Figure 5. An example of deployment descriptor

The connection model

The Component Manager load components and creates instances of components from the deployment descriptor or when it receives `ConnectTo` service. CM maintains a list of all components loaded and instances created in locally. For instance when the CM receive the `ConnectTo(A,B)` service, if B component instance is not exist then the CM create this B component instance. To make the connection between two components, the CM uses the services descriptors to retrieve components instance names, their types and services provided and required. After the CM connects two components, these components can communicate with each other directly without going through the CM.

In distributed mode, two solutions are proposed for the publication and discovery of the components available on the network. For the first solution, broadcast mode, each peer (each CM) provides access to all components created of connected peers. This solution is usable only for small network with few peers connected. For remote connections, two implementations are proposed; one built on the UDP transport protocol and the other on the TCP protocol. For the second solution (see Figure 6), Peer-to-Peer mode, each CM publishes components created in a Distributed Hash Table (DHT). This DHT is accessible to all registered peers on the network. This P2P solution is composed of two modules: DHT module for publication and discovery of components and PIPES module for communication between remote peers.

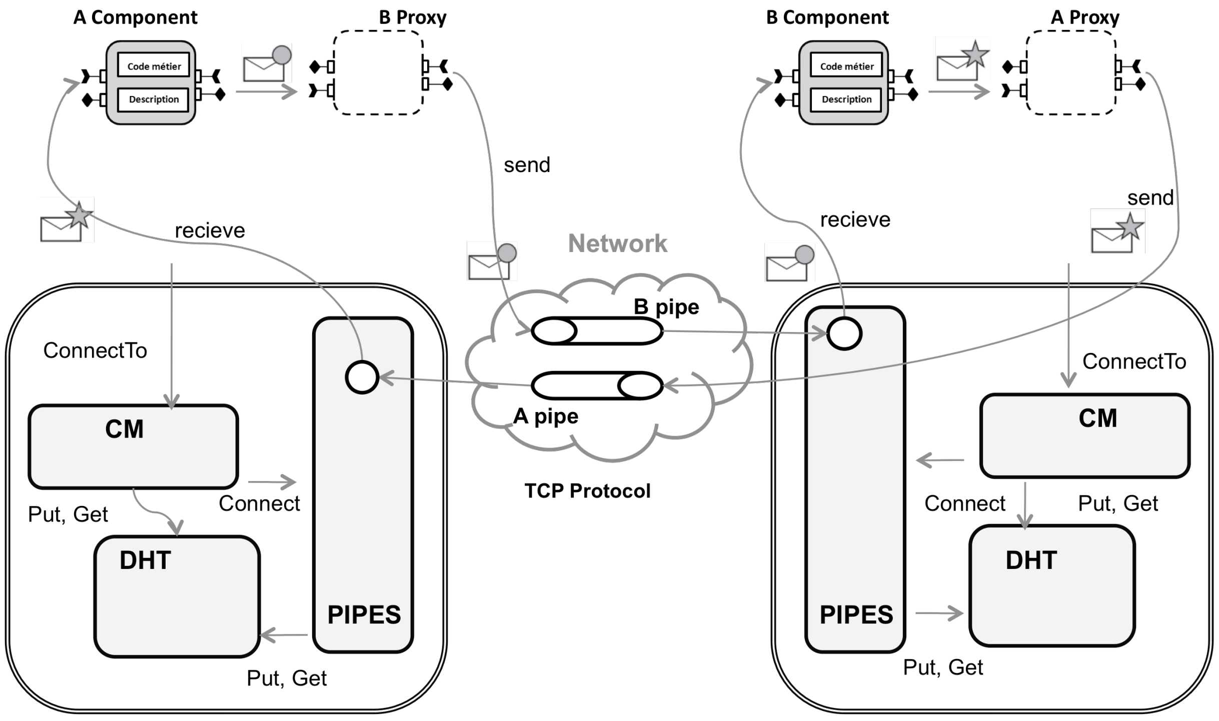

Figure 6. Architecture of SON at run-time

DHT module

CM delegates the management of lists of remote components to this DHT module. In the current version, we use the *OpenChord* implementation for the DHT module, 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, the CM publishes into this DHT, the useful information's for that a remote component can connect to this component.

PIPES module

The PIPES module handles the communication between remote components. It opens the TCP connection between peers. It is based on the concept of virtual pipes introduced into the JXTA technology. This concept allows passing through a single TCP connection, several logical communications (virtual pipes) between peers. Using this abstraction allows each component to open a virtual pipe to read messages sent to it. A virtual pipe is identified by a universally unique identifier (UUID for Universally Unique Identifier).

This identifier is associated with the component instance name and is 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 components.