International Journal of E-Business Research (IJEBR), Volume 5 Number 4, 2009

An Approach to Engineer Communities of Web Services
- concepts, architecture, operation, and deployment –

Zakaria Maamar[1]
Zayed University, Dubai, UAE
Sattanathan Subramanian
INRIA Saclay- Île-de-France, Paris, France
Jamal Bentahar
Concordia University, Montreal, Canada
Philippe Thiran
University of Namur, Namur, Belgium
Djamal Bensilamane
Claude Bernard Lyon 1 University, Lyon, France

Abstract. *This paper presents an approach that provides the necessary assistance to those who are in charge of engineering communities of Web services. Current practices indicate that Web services providing the same functionality are gathered into one community, independently of their origins and the way they carry out this functionality. The provided assistance manifests itself with the concepts to use, the architecture to select, the operations to script, and the deployment to track. Two protocols frame the interactions in an environment of communities of Web services namely the Web Services Community Development Protocol and the Contract-Net Protocol. The former manages a community in terms of Web services attraction/registration/withdrawal to/with/from this community. The latter satisfies users' needs in terms of Web services selection/contracting/triggering. Finally, the paper presents a prototype illustrating the engineering approach with focus on Web services attraction.*
Keywords. Community, Engineering, Web service.

1. Introduction

For the World Wide Web Consortium, a Web service ``*is a software application identified by a URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts and supports direct interactions with other software applications using XML-based messages via Internet-based applications*''. For the last few years, the development pace of Web services has been spectacular (Benslimane, 2007, DPD; Daniel, 2005; Dustdar, 2005). On the one hand, several standards have been developed to deal with for example Web services definition, discovery, and security (Andrews, 2003; Curbera, 2002). On the other hand, several projects have been initiated such as Web services composition, personalization, and contextualization (Baresi, 2007; Medjahed, 2007). These standards and projects have usually a common concern: Web services composition. Composition addresses the situation of a user's request that cannot be satisfied by any single, available Web service, whereas a composite Web service obtained by combining available Web services may be used.

Nowadays, competition between businesses does not stop at goods, services, or software products, but includes as well systems that offer the most recent and accurate information. For example, Google and

[1] Contact author: zakaria.maamar@zu.ac.ae

Yahoo are both search engines. The common practice is to bind to one of the engines according to various factors like reliability, efficiency, previous experiences, financial charges, etc. Web services are definitely not excluded from this competition. Independent providers develop several Web services that could offer the same functionality such as currency exchange. It is reported in (Bui, 2005) that although Web services are heterogeneous, the functionalities these Web services offer are sufficiently well defined and homogeneous enough to allow for market competition to happen. To ease and improve the process of Web services discovery in an open environment like the Internet, we suggested in (Benslimane, 2007; Maamar, 2007; Subramanian, 2007) along with other researchers in (Benatallah, 2003; Medjahed, 2007; Medjahed, 2005) to gather similar Web services[2] into groups known as communities. The notion of group/community/cluster highlights the importance of developing guidelines that would permit the management of Web services to be now parts of communities. Although Web services are investigated in various research projects (Anderson, 2006; Foster, 2006; Mrissa, 2008; Younas, 2006) these guidelines still lack and hence, examining the following elements would be deemed appropriate: (1) how to initiate, set up, and specify a community, (2) how to specify and manage the Web services in a community, and (3) how to reconcile conflicts within a community and between communities?

A community of Web services is dynamic by nature: new Web services join, other Web services leave, some Web services become temporarily unavailable, some Web services resume operation after suspension, just to name some. All these events need to be closely monitored and followed up, otherwise conflicts arise. For example, if a Web service left a community without prior notice, its peers would continue to assume it is still in this community. Moreover, Web services do not always exhibit a cooperative attitude when they become members of a community. First, they can compete on common computing resources, which may affect their performance scheduling. Second, they can announce misleading information (e.g., non-functional details) to enhance their participation opportunities in composite Web services. Last but not least, they can become malicious when they try to alter other Web services' data or behaviors.

Designing, developing, and managing communities of Web services seem to be a cumbersome process on designers/developers, who would definitely benefit from an approach that would assist them engineer such communities. For this purpose, this assistance needs to shed the light on 4 elements: concepts to use, architecture to select, operation to script, and deployment to track. The rest of this paper proceeds as follows. Section 2 consists of three parts dedicated to concept definition, architecture of a community environment, and functioning of this architecture, respectively. Section 3 details the internal structure of the two types of Web services that populate a community. A prototype

[2] Similar Web services and Web services with similar functionality are interchangeably used.

simulating community functioning is presented in Section 4. Sections 5 and 6 are about related and future work, respectively. Conclusions are drawn in Section 7.

2. Engineering approach for Web services communities

2.1 Definitions

The term community means different things to different people. In Longman Dictionary, community is ``a group of people living together and/or united by shared interests, religion, nationality, etc.''. In the field of knowledge management, communities of practice constitute groups within (or sometimes across) organizations who share a common set of information needs or problems (Davies, 2003). Communities are not a formal organizational unit but an informal network with common interests and concerns.

When it comes to Web services, Benatallah et al. define community as a collection of Web services with a common functionality although these Web services have distinct non-functional properties (Benatallah, 2003). Medjahed and Bouguettaya use community to provide an ontological organization of Web services sharing the same domain of interest (Medjahed, 2005). Medjahed and Atif use community to implement rule-based techniques for comparing context policies of Web services (Medjahed, 2007). Finally, Maamar et al. define community as a means to provide a description of a desired functionality without explicitly referring to any concrete Web service (already known) that will implement this functionality at run-time (Maamar, 2007).

2.2 Architecture

Fig. 1 represents a proposed architecture of multiple communities of Web services. Additional components in this architecture are providers of Web services and UDDI registries (or any type of registry like ebXML). Communities are established and dismantled according to specific scenarios and protocols that are detailed in Section 2.3. UDDI registries receive advertisements of Web services from providers for posting purposes. Several UDDI registries could be made available across the Internet because competitor might not want to have their Web services registered in the same UDDI registry (Arpinar, 2004). Several UDDI registries mean balancing the load of handling advertisements and user-search requests of Web services over these UDDI registries, but at the same time raise some questions like content consistency. To keep the focus of this paper on community engineering, discussions on UDDI management are excluded.

Fig. 1 offers some characteristics that need to be stressed out. First, the common way to describing, announcing, and invoking Web services is still the same although Web services are now associated

with communities. Second, the regular facilities that UDDI registries offer are still the same; no extra facilities are required to accommodate communities' needs. Finally, the selection of Web services out of communities is transparent to users and independent of the way they are gathered into communities. Two communities of Web services are shown in Fig. 1. They could for example have airfare quotation and hotel booking as functionalities, respectively. A master component always leads a community.
This master could itself be implemented as a Web service (like shown in Fig. 1) for compatibility purposes with the rest of Web services in a community, which are now denoted as slaves. Master-Slave Web services relationship in a community is regulated using the well-known Contract Net protocol (Smith, 1980) (*CNProtocol*). Needless to say that a single master Web service constitutes a bottleneck in a community operation. An immediate solution would be the use of duplicate masters to intervene upon request, but this is outside this paper's scope.

One of the responsibilities of the master Web service is to attract Web services to be part of its community using rewards (Bentahar, 2007, IS; Bentahar, 2007, WAMIS). As a result, the master Web service regularly checks out UDDI registries so that it is kept updated about the latest changes like new advertisements in their respective contents. More responsibilities of the master Web service are (i) nominate the slave Web service out of several peers to participate in a composite Web service, and (ii) run the *CNProtocol* for the needs of nominating this Web service.

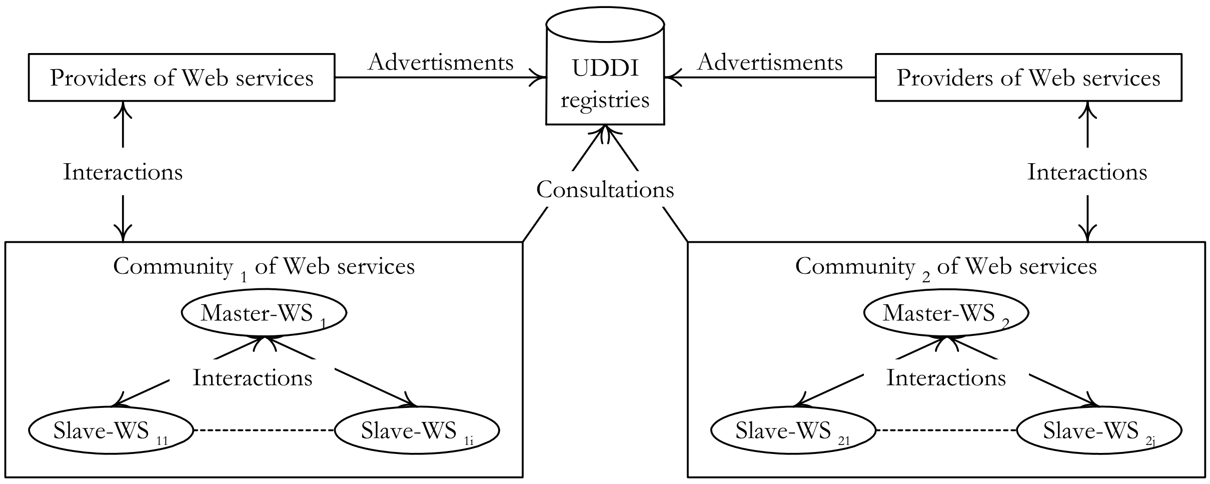


Figure 1 Architecture of Web services communities

In a community, the master Web service is designated in two different ways. The first way is to have a dedicated Web service play the role of master for the time being of a community. This Web service is independently developed (e.g., application designer) from other Web services that are advertised in UDDI registries. It should be noted that the Web service that leads a community never participates in any composition. Therefore, this Web service is only loaded with mechanisms related to community management like Web services attraction and retention. The second way is to identify a Web service from the list of Web services that already populate a community to act as a master. This identification could happen on a voluntary basis or by running an election process among the Web services. Because of the temporary no-participation restriction of a master Web service in compositions, the nominated Web service will be compensated (Bentahar, 2007, IS). The call for elections in a community regularly takes place, so that the burden on the same Web services to lead a community is either minimized or avoided. In this paper, the first way is adopted, i.e., having an independent Web service play the role of master.

2.3 Operation

The operation of the approach to engineer a community of Web services addresses the following questions: (1) how to establish a new community, (2) how to dismantle an existing community, (3) how to attract new Web services to a community, (4) how to retain existing Web services in a community, and (5) how to select slave Web services from a community to take part in a composition scenario?

2.3.1 Community development

A community is initially developed to gather Web services with similar functionalities. This gathering is a designer-driven activity that includes two steps. The first step is to define the functionality, e.g., flight booking, of the community by binding to a specific ontology (Medjahed, 2005). This binding is crucial since providers use different terminologies to describe the functionality of their respective Web services. For example, flight booking, flight reservation, and air-ticket booking are all about the same functionality. To keep the paper focused on the engineering aspect of communities of Web services, the use of ontologies is no further discussed.

The second step is to deploy the master Web service to lead the community and take over multiple responsibilities. One of them is to invite and convince Web services to sign up in its community. The survivability of a community, i.e., to avoid its dismantlement, depends to a certain extent on the status of the existing Web services in this community. Another responsibility is to check the credentials (e.g., announced QoS, adopted protection mechanisms) of Web services before they are admitted into a community. This checking has a dual advantage: boost the security level among the peers in a community and enhance the trustworthiness level of a master Web service towards the slave Web services it manages. The first advantage avoids dealing with malicious Web services that could attempt to alter other peers' data and behaviors. The second advantage shows how much the master Web service relies on the slave Web services in completing the prescribed operations. Enhancing the security of a community is an important factor that contributes towards its reputation. Such a reputation is fundamental to attract both new Web services to sign up and users to request Web services (Elnaffar, 2008). It should be noted that slave Web services could turn out to be "lazy"[3] after joining a community, which calls for their immediate ejection from this community.

Dismantling a community is a designer-driven activity as well and happens upon request from the master Web service. This one oversees the events in a community such as arrival of new Web services, departure of some Web services, identification of Web service to be part of composite Web services, and sanctions on Web services because of misbehavior. When a master Web service notices first, that the number of Web services in a community is less than a certain threshold and second, that the number of participation requests in composite Web services that arrive from users over a certain period of time is also less than another threshold, the community could be dismantled. Both thresholds are set by the designer. Web services to eject from a community can join other communities that are interested in these Web services subject to assessing their functionality similarity with the functionalities of these communities.

Table 1 shows the role of both thresholds (number of Web services in a community and number of Web services in compositions) in the decision of keeping or dismantling a community. Four cases are illustrated along with some comments on the recommended actions to take per case. For instance, when the number of Web services in a community is "high" but the number of participation of these Web services in compositions is "low", this means that the community has a poor configuration. To remedy that configuration, some Web services with a low level of participation in compositions are ejected from the community and other Web services are invited to join the community. A "low" level of participation could be explained by the poor competitiveness (e.g., QoS) of a Web service against other Web services in the same community.

Table 1 Community management

Number of Web services in		**Comment**	**Recommended Action**
Community	**Compositions**		
Low	High	Efficient configuration	Keep inviting Web services
High	Low	Poor configuration	Eject Web services with low participation and invite new ones
Low	Low	Very poor configuration	Dismantle community
High	High	Desired configuration	Maintain same strategy

As part of the approach to engineer communities of Web services, the Web Services Community Development Protocol (*WSCDProtocol*) frames the operations that lead to community development (Fig. 2). These operations are grouped into three categories: WS-Attraction with operations 1 to 4,

[3] Web services that do not satisfy the QoS that a master Web service advertises and guarantees to potential users.

WS-Registration with operations 5 and 6, and WS-Withdrawal with operations 7 and 8. A slave Web service could voluntarily decide to leave a community for various reasons like lack of business opportunities in a community. In addition, this slave Web service could receive a departure notice from the master Web service due to poor performance.

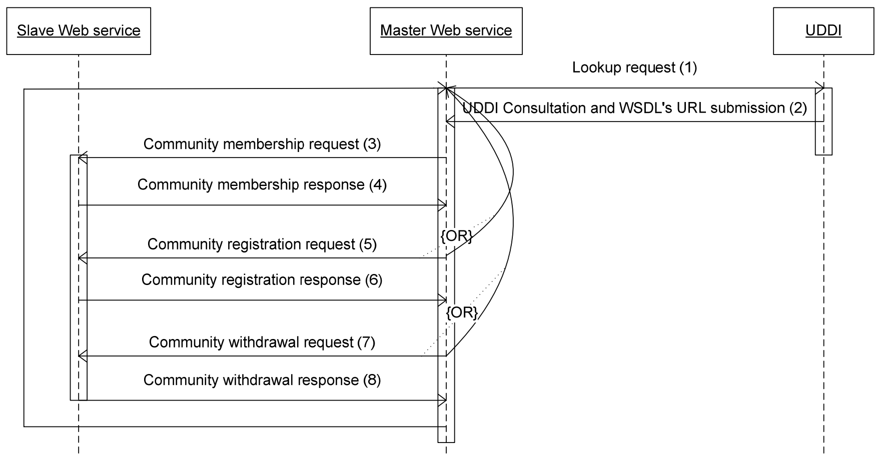

Slave Web service
Master Web service
UDDI
Lookup request (1)
UDDI Consultation and WSDL's URL submission (2)
Community membership request (3)
Community membership response (4)
{OR}
Community registration request (5)
Community registration response (6)
{OR}
Community withdrawal request (7)
Community withdrawal response (8)

Figure 2 Chronology of operations in the *WSCDProtocol*

2.3.2 Web services attraction and retention

Attracting new Web services to and retaining existing Web services in a community fall into the responsibilities of the master Web service. A community could vanish if the number of Web services running in it drops below a certain threshold (Table 1).

Web services attraction makes the master Web service consult regularly the different UDDI registries looking for new Web services[4]. These latter could have recently been posted on UDDI registries or have seen the description of their functionality changed. Changes in a Web service's functionality pose challenges as a Web service may no longer be suitable for a community. As a result this Web service is invited to leave the community. When searching for candidates to join a community, a mapping of the ontology used in the community with other ontologies that can be used by different Web services takes place. This mapping is essential to deal with the problem of using different terminologies to describe Web services' functionalities. Different algorithms and approaches for mapping and merging ontologies have been proposed (Arpinar, 2004) (Noy, 1997). To keep the paper focused on engineering aspects of communities of Web services, these ontological issues are not considered further in the paper. When a candidate Web service is identified based on the functionality it offers, the master Web service interacts with its provider (Fig. 1). The purpose is to ask the provider to register its Web service with the community of this master Web service. Some arguments that are used to convince the provider include high participation-rate of the existing Web services in composite Web

[4] Expressing interests in some Web services to UDDI registries through subscription could be used to keep a master Web service updated.

services (this is a good indicator of the visibility of a community of Web services to the external environment and the reputation of Web services (Maximilien, 2002)), short response-time when handling user requests, and efficiency of the security mechanisms against malicious Web services.

Retaining Web services to remain committed to a community for a long period of time is a good indicator of the following elements:

- Although Web services in a community are in competition, they expose a cooperative attitude. For instance, Web services have not been subject to attacks from peers in the community (because all Web services would like to participate in composition scenarios, some of them could try to make other peers less competitive by illegally altering their execution properties). This backs the security argument that the master Web service uses again to attract Web services and convince their providers.
- A Web service is satisfied with its participation rate in composite Web services. This satisfaction rate is set by its provider. Plus, this is inline with the participation-rate argument that the master Web service uses to attract new Web services.
- Web services are, through the master Web service, aware of some peers in the community that could replace them in case of failure, with less impact on the composite Web services in which they are involved. More details on replacement are provided in Section 6.

Web services attraction and retention shed the light on a third scenario, which is Web services being invited to leave a community as briefly reported earlier. A master Web service could issue such a request upon assessment of the following criteria:

- The Web service has a new description of the functionality it provides. The description does not match the functionality of the community.
- The Web service is unreliable. On different occasions the Web service failed to participate in composite Web services due to recurrent operation problems.
- The credentials of the Web service were “beefed up” to enhance its participation opportunities in composite Web services. Large differences between a Web services’ advertised QoS and delivered QoS indicate performance degradation (Ouzzani, 2004).

2.3.3 Web services selection

In a community, interactions to select Web services for the needs of composition rely on the intrinsic concepts of the contract-net protocol, namely job contracting and subcontracting between two types of agents known as initiator (master Web service) and participant (slave Web service). At any time an agent can be initiator, participant, or both. The sequence of steps in the contract-net protocol, which we slightly extend, is as follows: (1) initiator sends participants a call for proposals with respect to a certain job to carry out; (2) each participant reviews the call for proposals and bids if interested (i.e., feasible job); (3) initiator chooses the best bid and awards a contract to that participant; and (4) initiator rejects other bids.

Mapping the contract-net protocol onto the operation of a community occurs as follows. When a user (through some assistance (Schiaffino, 2004)) selects a community based on its functionality, the master Web service of this community is contacted in order to identify a specific slave Web service that will implement this functionality at run-time. The master Web service sends all slave Web services a call for bids (CNStep 1). Prior to getting back to the master Web service, the slave Web services assess their status by checking their ongoing commitments in other compositions and their forthcoming maintenance periods (Maamar, 2006) (CNStep 2). Only the slave Web services that are interested in bidding inform the master Web service. This latter screens all the bids before choosing the best one (CNStep 3)[5]. The winning slave Web service is notified so that it can get itself ready for execution when requested (CNStep 3). The rest of the slave Web services that expressed interest but were not selected, are notified as well (CNStep 4).

As part of the approach to engineer communities of Web services, the *CNProtocol* frames the operations that lead to Web services selection for composition (Fig. 3). These operations are grouped into two categories: ContractAgreement with operations 1 to 5, and ContractCompletion with operations 6 and 7.

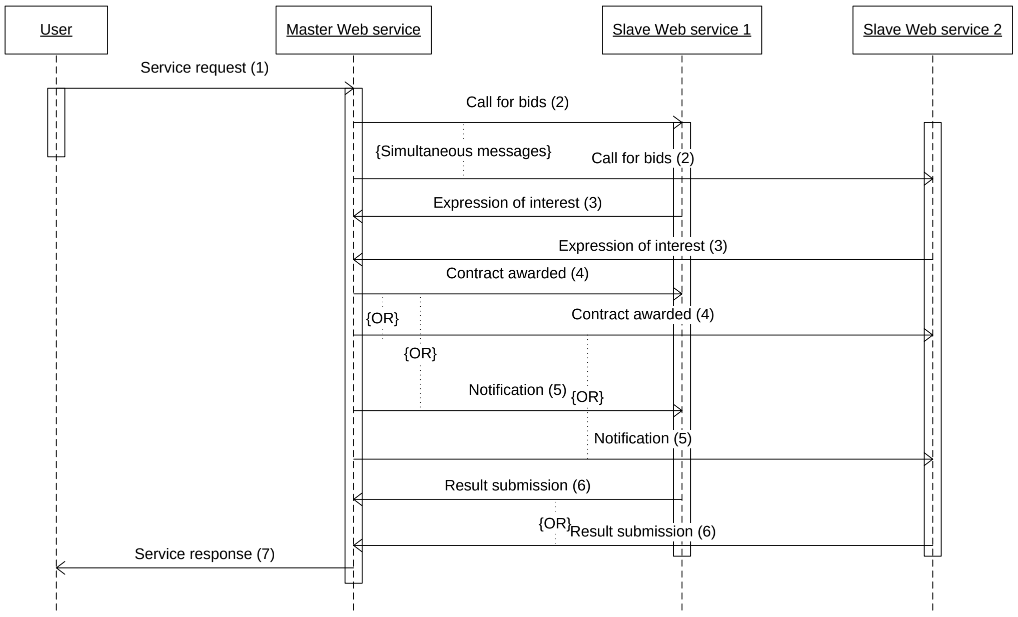


Figure 3 Chronology of operations in the *CNProtocol*

[5] In case there are several tied bids, different selection opportunities are offered to the masterWeb service like randomly, firstly received, etc.

3. Master/Slave Web services: internal structure

The main functions that embody master and slave Web services are presented in this section. This is deemed relevant from an engineering perspective as this would facilitate the implementation work. *WSCDProtocol* and the *CNProtocol* trigger these functions at run-time. In the following, M/SWS stands for Master/Slave Web Service. In this section, the following notation is adopted: *OutputResult* ← *NameOfTheFunction(InputParameters)* where ← is the assignment operator.

3.1 Master Web service

Fig. 4 presents the main functions of a master Web service. They are grouped into three modules: (i) **MWS-Development** consists of *MWS-Attraction*, *MWS-Registration*, and *MWS-Withdrawal* functions; they are devoted to the *WSCDProtocol*, (ii) **MWS-RequestHandler** consists of *MWS-Request*, *MWS-Response*, *MWS-ContractEstablishment*, *MWS-ContractResult*, and *MWS-DataMediation* functions; they are devoted to the *CN-Protocol*, and (iii) **MWS-Monitoring** consists of *MWS-Liveness*, *MWS-QoS*, and *MWS-Trust* support functions; they are devoted to both protocols. In Fig. 4, two interfaces exist. *MWS-Community-Interface* provides an external interface to the slave Web services for the following functions: *MWS-Registration*, *MWS-Withdrawal*, and *MWS-ContractResult*. *MWS-Abstract-Interface* provides an external interface to a user to trigger Web services.

1. **MWS-Development module.** In this module, *MWS-Attraction* function implements Operation 1 through Operation 4 of the *WSCDProtocol* (Fig. 2). Initially, this function submits to a UDDI registry the name of the functionality that labels the community of the master Web service (Operation 1). Upon reception, the UDDI-registry returns details like WSDL files on some Web services that could be potential candidates to join this community because of the matching between their respective functionalities and this community's functionality (Operation 2). In addition, these Web services are still not bound yet to any specific community. Afterwards, *MWS-Attraction* function extracts the URLs from these WSDL files and makes an explicit call to the appropriate Web services. The objective of this call is to invite the candidate Web services to be part of the community of the master Web service using arguments like reputation and benefits (Operation 3). The Web services that show interest get in touch with the master Web service (Operation 4). In addition, if they accept the invitation, *MWS-Registration* function gets triggered (Operation 5 and Operation 6). *RegisteredInfo* ← *MWS-Registration(NewWSDetails)* illustrates the performance of the above operations where:
    - *MWS-Registration* is the name of the function that permits Web services to express their interest in being part of a community.
    - *NewWSDetails* is the incoming message from a Web service to a master Web service.
    - *RegisteredInfo* is the outgoing message from a master Web service to a Web service.

NewWSDetails and *RegisteredInfo* are complex data types. It is the responsibility of application designers to identify the complete structure of these messages. However the following minimal details are recommended for *NewWSDetails*: WSDL and QoS non-functional parameters. For *RegisteredInfo*, slave Web service's identifier is the recommended minimal detail.

MWS-Registration function identifies as well the required mappings, i.e., data mediation via *MWS-DataMediation* of *MWS-RequestHandler* module, which needs to be established during master-slave Web services communications. For instance, the functionality of the community of the master Web service is *getZipCode*, but the slave Web service offers the same functionality using *lookupZipCode* to name its functionality. This requires a mediation to establish master-slave Web services communications (Noy, 1997); it is taken care by the *MWS-DataMediation* function. These mappings are later submitted to *MWS-ContractEstablishment* function of *MWS-RequestHandler* module and *MWS-QoS* function of *MWS-Monitoring* module. The objective is to keep them updated about the new Web service that will shortly be considered as a slave.

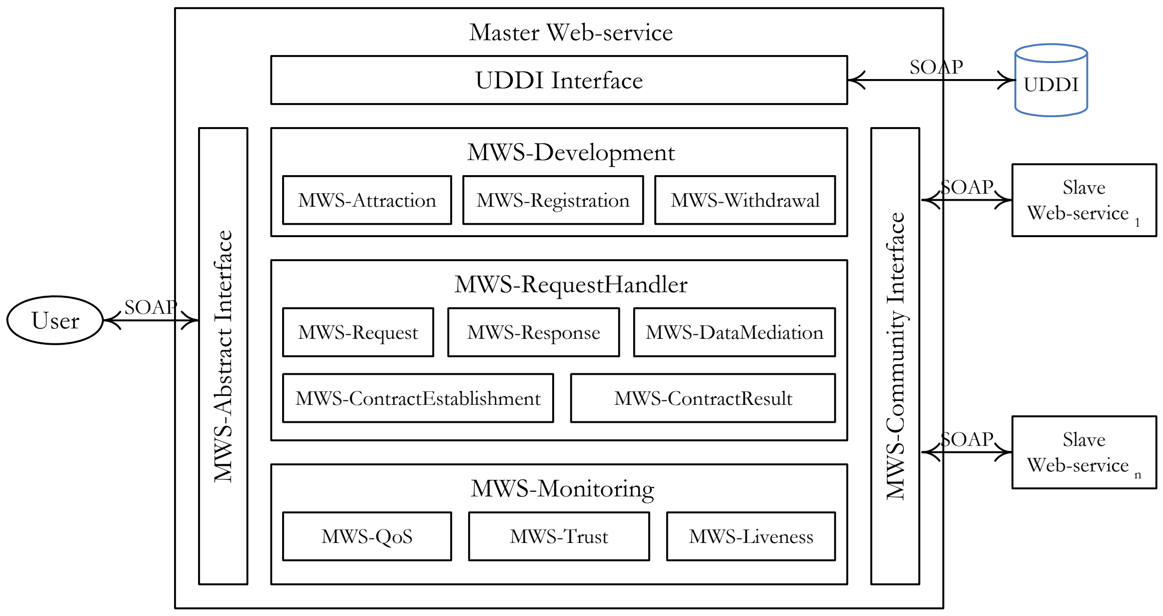


Figure 4 Master Web service architecture

In the **MWS-Development** module, *MWS-Withdrawal* function allows the slave Web service to withdraw itself from a community (Operation 7 and Operation 8 of *WSCD-Protocol*). For instance, if a slave Web service finds out that the current business opportunities are not enough attractive according to its provider's business plan, then it will take the necessary actions to be withdrawn from the community by invoking *MWS-Withdrawal* function. This functions enables a slave Web service to pull out of a community if for instance, the trust level with its master Web service goes below a certain predefined threshold. Trust level is obtained using *MWS-Trust* function of *MWS-Monitoring* module. To withdraw a slave Web service, a master Web service uses *SWS-Withdrawal* function that a slave Web service provides (Section 3.2). *WithdrawInfo* ← *MWS-Withdrawal(RegisteredInfo)* illustrates the supported operations where:

- *MWS-Withdrawal* is the name of the function that allows a slave Web service to express its intention of departure from a community to a master Web service.
- *RegisteredInfo* is the incoming message from a slave Web service to a master Web service.
- *WithdrawInfo* is the outgoing message from a master Web service to a slave Web service.

  RegisteredInfo and *WithdrawInfo* are also complex data types. The following minimal detail is recommended for both: slave Web service's identifier.

2. ***MWS-RequestHandler* module.** Its various functions are in charge of running the *CNProtocol* (Fig. 3). *MWS-Request* function implements Operation 1 by receiving and forwarding the user's request like hotel booking to *MWS-ContractEstablishment* function. This one performs Operation 2 through Operation 4 by submitting a call for bids to all slave Web services (Operation 2) and waiting for responses from interested slave Web services (Operation 3) by calling *SWS-CallForProposal* function that a slave Web service provides (Section 3.2). Following best-bid selection, *MWS-ContractEstablishment* function assigns a contract to the slave Web service of the best bid and informs the rest of slave Web services of this assignment as well (Operation 4) by calling *SWS-AwardWithContract* function that a slave Web service provides (Section 3.2). After a while, the slave Web service finishes processing the user's request and provides results back to *MWS-ContractResult* function (Operation 5 and Operation 6) that forwards these results to *MWS-Response* function. The purpose is to format and modify results using *MWS-DataMediation* function so that the user's requirements (e.g., preferred language) are met. In addition, *MWS-ContractEstablishment* and *MWS-ContractResult* functions provide performance details on the slave Web service that completed the functionality to *MWS-Trust* function of *MWS-Monitoring* module. *MWS-ContractResult(Results)* illustrates how Operation 5 of the *CNProtocol* is called by a slave Web service, where
   1. *MWS-ContractResult* is the name of the function which will be called by a slave Web service to provide the contract results to a master Web service.
   2. *Results* is the message to be sent out to the user. It has a complex data type and is business-driven.

3. ***MWS-Monitoring* module**. It mainly contains functions that support the performance of the

functions of *MWS-Development* and *MWS-RequestHandler* modules. *MWS-QoS* function receives details from *MWS-Registration* and *MWS-Withdrawal* functions of *MWS-Development* module on a slave Web service that is about to enter or leave a community, respectively. *MWS-Trust* function receives details from *MWS-ContractEstablishment* function and *MWS-ContractResult* function of *MWS-RequestHandler* module on the current QoS of a slave Web service after performing a user's request. The objective is to rate the performance of this slave Web service. Interested readers in rating mechanisms are referred to (Maximilien, 2004). Finally, *MWS-Liveness* function is the ping utility that a master Web service uses to check the liveness of a slave Web service by calling *SWS-Liveness* function that a slave Web service provides (Section 3.2). This function provides as well details to *MWS-Trust* function so that it can compare between the agreed QoS of a Web service and the assessed QoS during performance.

3.2 Slave Web service

Fig. 5 presents the main functions of a slave-Web service. They are grouped into three modules: (i) **SWS-Adjournment** consists of *SWS-Withdrawal* function; it is devoted to the *WSCDProtocol*; (ii) **SWS-ContractHandler** consists of *SWS-CallForProposal*, *SWS-AwardWithContract*, and *SWS-ContractResult* functions; they are devoted to the *CN-Protocol*; and (iii) **SWS-Monitoring** with *SWS-Liveness* function. It should be noted that almost each module in a slave Web service has a counterpart module in a master Web service. *SWS-Community-Interface* provides an external interface to the master Web service for the following functions: *SWS-Withdrawal*, *SWS-CallForProposal*, *SWS-AwardWithContract*, and *SWS-Liveness*.

1. **SWS-Adjournment module.** In this module, *SWS-Withdrawal* function supports the master Web service pulls a slave Web service out of a community (Operation 7 and Operation 8 of the *WSCDProtocol*). In addition this function invokes *MWS-Withdrawal* function of a master Web service when a slave Web service decides to quit a community. *WithdrawInfo* ← *SWS-Withdrawal(RegisteredInfo)* illustrates both operations where:
    - *RegisteredInfo* is the incoming message from a master Web service to a slave Web service.
    - *WithdrawInfo* is the outgoing message from a slave Web service to a master Web service.
    - *RegisteredInfo,* and *WithdrawInfo* have a complex data types and are business driven.

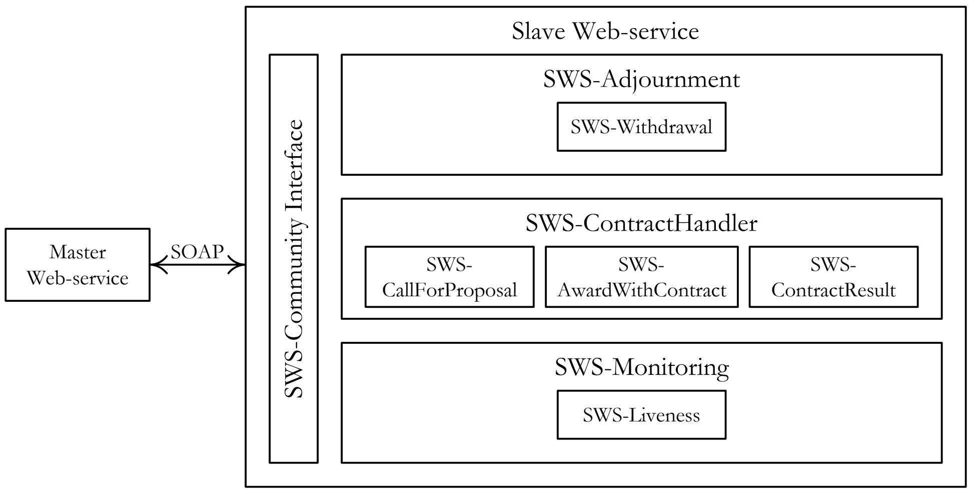


Figure 5 Slave Web service architecture

2. ***SWS-ContractHandler* module.** In this module, *SWS-CallForProposal* function supports a master Web service perform Operation 2 and Operation 3 of the *CNProtocol*. This function is illustrated with *BidDetails SWS-CallForProposal (ContractAdvertisement)* where:
    - *ContractAdvertisement* is the incoming message from a master Web service to a slave Web service about details on the contract that a user submitted to this master Web service.
    - *BidDetails* is the outgoing message from a slave Web service to a master Web service. It contains the bid details with respect to *ContractAdvertisement*.

    ContractAdvertisement and *BidDetails* have a complex data type are and are business driven. Still in the **SWS-ContractHandler** module, *SWS-AwardWithContract* function supports a master Web service perform Operation 4 of the *CNProtocol*. *SWS-AwardWithContract(ContractDecision)* illustrates this operation where:
    - *ContractDecision* is the incoming message sent by a master-Web service to a slave Web service. It contains the result with respect to *BidDetails*. It has a Boolean data type.

    Finally, *SWS-ContractResult* function allows a slave Web service to deliver its performance results back to a master Web service by calling *MWS-ContractResult* function of this master Web service.
3. **SWS-Monitoring module.** In this module, *SWS-Liveness* is a function that *MWS-Liveness* function of a master Web service calls to check the liveness of a slave Web service. *LiveFlag SWS-Liveness(PingInfo)* illustrates this function where:
    - *SWS-Liveness* is the name of the function that a master Web service uses to obtain information about the liveness of a slave Web service.
    - *LiveFlag* is the outgoing message from a slave Web-Service to a master Web service. It has a Boolean data type.

- *PingInfo* is the incoming message that a slave Web service provides to a master Web service. It has a complex data type and is business driven. However, slave Web service's identifier is recommended to be part of this message.

4 Approach implementation

The implementation of the approach to engineer communities of Web services started by identifying how the *WSCDProtocol* and the *CNProtocol* interact with each other. This interaction manifests itself with the different invocation requests that are submitted to master and slave Web services' modules (Section 3). Afterwards, the implementation continued with programming different scenarios like Web services attraction (focus of this section), selection, departure, etc. Overall, the implementation was built around (1) XML for request and response specification between users and Web services and between master Web services and slave Web services; (2) JDK 1.4 for operation processing, and (3) Eclipse 3.2 as an integrated development environment. WSDL files defining master and slave Web service are reported in Appendices 1 and 2, respectively.

4.1 Interactions between protocols

The *WSCDProtocol* and the *CNProtocol* manage communities in terms of attracting Web services to these communities and supporting their engagement in providing facilities related to users' needs. These protocols run in parallel and can be initiated from two points: *Start/WSCDP* is the beginning of a community-management scenario and *Start/ECNP* is the beginning of a user-need-satisfaction scenario. In addition to the transitions that are intra to the *WSCDProtocol* and the *CNProtocol*, respectively, these protocols connect with one another as depicted in Fig. 6. On the one hand, the transition from the *WSCD-Protocol* to the *CNProtocol* shows the Web services that are interesting in bidding to satisfy users' needs (i.e., requested Web service). On the other hand, the transition from the *CNProtocol* to the *WSCDProtocol* shows the Web services that need now to be evaluated following users' needs satisfaction. Inter- and intra-transitions are supported with appropriate mechanisms like *WS-Registration*, *WS-Withdrawal,* and *ContractCompletion* as depicted again in Fig. 6.

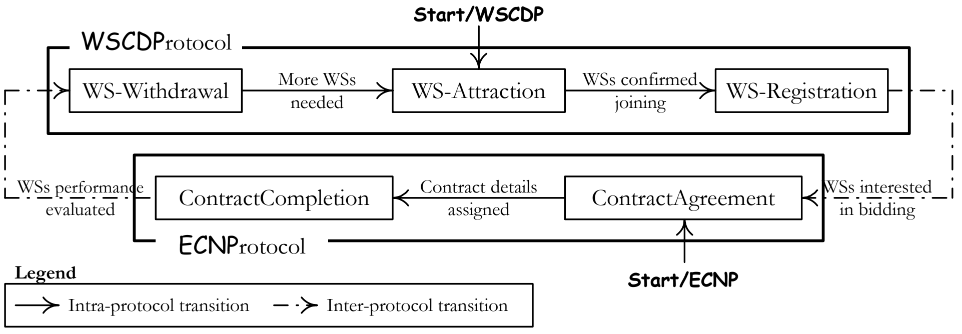
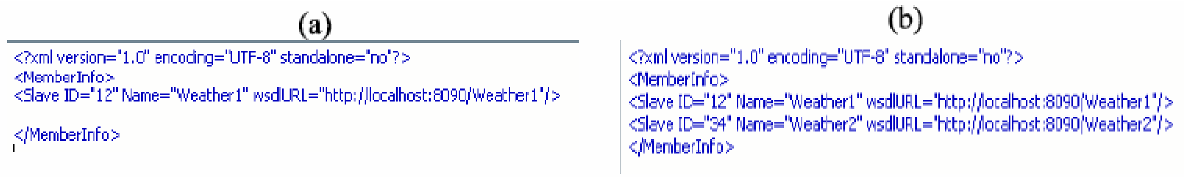


Figure 6 Intra- and inter-protocol interactions

4.2 Web services attraction

Fig. 7 illustrates the use of *MWS-Registration* function between the master Web service of *WeatherCommunity* and a set of Web services. Fig. 7 (a) is about the content of this community, which is currently one i.e., Weather Web service$_1$. As a result, the master Web service triggers *MWS-Attraction* function to interact with a UDDI registry by supplying the nature of functionality, i.e., weather forecast that needs to be attached to Web services. After screening the content of this UDDI registry a set of candidate Web services among them Weather Web service$_2$ are identified. The master Web service contacts Weather Web service$_2$'s provider with details on *WeatherCommunity* as a potential host of this Web service. Some details include the participation rate of Weather Web service$_1$ in previous composition scenarios. Upon acceptance to join *WeatherCommunity*, Weather Web service$_2$ invokes *MWS-Registration* function to get registered in this community. The master Web service checks Weather Web service$_2$'s credentials and assigns an identifier prior to finalizing its entrance (Fig. 7 (b)).


(a)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<MemberInfo>
<Slave ID="12" Name="Weather1" wsdlURL="http://localhost:8090/Weather1"/>

</MemberInfo>
```

(b)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<MemberInfo>
<Slave ID="12" Name="Weather1" wsdlURL="http://localhost:8090/Weather1"/>
<Slave ID="34" Name="Weather2" wsdlURL="http://localhost:8090/Weather2"/>
</MemberInfo>
```

Figure 7 Screen-shots illustrating progress in Web services attraction

5. Related work

In the introduction section, it was reported that the majority of the research initiatives in the field of Web services were mainly concerned with composition with little focus on other issues like how to engineer communities of Web services. The following summarizes some of the initiatives that helped shape the approach to engineer Web services.

In (Foster, 2005), Foster et al. proposed the LTSA-WS tool to verify compositions of Web services implementations. The core idea is to look at Web services engineering task from two perspectives namely process verification and model checking. Properties established from the design specifications and implementation models can then be compared to the expected results from the designer and implementer viewpoints. In [37], Spencer et al. noticed that it is not obvious to adopt a general method including analysis, design, testing, and validation steps that would achieve the reliability of semantic Web services composition. To this purpose, they insisted on the requirements that a framework for this kind of composition needs to satisfy such as (a) handling errors and running recovery actions, (b) updating logs by monitoring data flow while composing Web services, and (c) using logs to ensure a post-runtime analysis. In (Baresi, 2005), Baresi et al. proposed a policy-based approach to monitor Web services' functional (e.g., constraints on exchanged data) and non-functional (e.g., security, reliability) requirements. This approach suggests the use of policies to be defined along the life cycle of a Web service. Policies are of types service, server, supported, and requested. In (Benatallah, 2006), Benatallah et al. propose a model-driven framework, called Service Mosaic, as a CASE tool for developing Web services-based applications. The proposed tool includes much functionality like protocol compatibility and replaceability, a BPEL generator from business protocol specifications, etc.

Regarding Web services communities, Paik et al. present the WS-CatalogNet system, which aims at cataloguing Web services communities and creating peer relationships between them. The different communities can then collaborate during query processing (Paik, 2005). Communities are described in terms of category definitions represented with class description languages. The relationship between communities can be created when their categories are similar to a certain extent. In addition, the WS-CatalogNet system provides monitoring functionality by logging community events and analyzing community interactions. In (Medjahed, 2004), Medjahed et al. proposed the WebBIS system as a generic framework for defining and managing Web services composition in dynamic environments. In this framework, Web services are semantically organized in terms of pull- and push-communities. Both communities establish static and dynamic relationships between Web services. A WebBIS-SDL language is proposed to advertise and monitor Web services.

In (Bianchini, 2005), Bianchini et al. suggested service ontology to help organize Web services in three abstraction layers: concrete Web services, abstract Web services, and subject categories. Concrete Web services are directly invocable. Each cluster (or community) of similar concrete Web services is associated with an abstract Web service. This latter is not invocable. Finally, subject categories organize Web services into standard, available taxonomies and to provide a topic-driven access to the underlying abstract Web services.

The approach proposed in this paper for engineering Web services communities is substantially different from the aforementioned approaches, which are concerned among others with how to monitor Web services composition itself and how different communities collaborate. Communities, here, were looked into from two perspectives namely management and performance along with the following research questions: how to attract Web services, how to eject Web services, how to retain Web services, just to list some.

6. Future work

6.1 Alliance development

In Longman Dictionary, alliance is an arrangement in which two or more countries, groups, etc. agree to work together to try to change or achieve something. One of the scenarios that could affect the internal organization of a community is to set up alliances among Web services. An alliance is like a micro-community whose development would be triggered because of some mutual agreements between providers of Web services as part of their partnership strategies. Providers could join forces by referring to/recommending other peers' Web services and vice-versa. Alliances constitute an attractive solution to exception handling. A Web service could be "easily" substituted by another Web service in the same alliance before looking for another Web service in other alliances in the same community. The search and identification of a new Web service might have a major impact on the specification of the composition. In addition, during the execution of the extended contract-net protocol, a Web service could directly subcontract a composition request to the Web services in its alliance before going through the master Web service. This would help reduce the interactions between the master Web service and all slave Web services. Finally, like a community, an alliance would have a dynamic nature: new alliances could be formed, new members could be admitted to and excluded from alliances, and some alliances could be either discarded or merged.

6.2 Community *versus* society of agents

It would be tempting to look into the similarity between a society of software agents and a community of Web services. In (Narendra, 2001), Narendra defines this kind of society as a group of software agents that come together. The purpose is to collaborate and meet some common goals. At this development stage of the proposed engineering approach, this society definition does seem appropriate for community and further investigation is deemed appropriate. For instance, the Web services in a community do not collaborate. They, however, compete in order to participate in composite Web services since they all offer the same functionality but in a different set-up. The collaboration takes place at the community level where Web services from independent communities work together. Each community contributes one Web service to a composite Web service.

Trust is another important issue that could be looked into. Indeed, trust models developed for agent societies could be adapted with respect to the intrinsic characteristics of communities of Web services. However, unlike agent societies, communities of Web services should address the trust issue from three perspectives: user, slave Web service, and master Web service. Users should be able to identify trustworthy communities when requesting Web services. Slave Web services should be equipped with mechanisms allowing them to assess if a master Web service is trustworthy or not, particularly when the master Web service selects slave Web services out of a community to participate in composition scenarios. Last but not least, a master Web service should be able to distinguish malicious from non malicious Web services before and after joining its community.

7. Conclusion

This paper laid down the foundations upon which the engineering communities of Web services would take place. These foundations primarily revolved around the concepts to use, the architecture to select, the operations to script, and the deployment to track. The role of a community is to gather Web services with similar functionalities (like FlightBooking) independently of who developed these Web services and how these latter carry out their respective functionalities. Web services in a community were specialized into two types known as master and slave. The master Web service led a community and interacted with users and providers of Web services. A slave Web service satisfied users' needs as per the master Web service's request.

Two protocols namely Web Services Community Development (*WSCDProtocol*) and Contract-Net (*CNProtocol*) framed the interactions between master and slaveWeb services in a community. Samples of interactions included attracting Web services to a community, convincing Web services to remain in a community, just to name a few. In addition, these interactions were experimented through a prototype.

Our future work is twofold: alliance development and community agent society comparison. The first part would look into the use of alliances as a means to internally structure a community. An alliance is like a micro-community that would be developed because of some mutual agreements between providers of Web services as part of their partnership strategies. The second part would identify the similarities and differences between a society of agents and a community of Web services. There is a research trend that suggests coupling Web services to software agents (Cavedon, 2005).

Acknowledgments. The authors would like to thank the reviewers for their comments and suggestions of changes.

References

Anderson, A. H. (2006). Web Services Policies. *IEEE Security & Privacy*, 4(3).

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., & Weerawarana, S. (2003). Business Process Execution Language for Web Services, Version 1.1. *Standards proposal by BEA Systems, IBM Corporation and Microsoft Corporation*.

Baresi, L., Guinea, S., & Plebani, P. (2005). WS-Policy for Service Monitoring. *Proceedings of the 6th Workshop on Technologies for E-Services (TES) held in conjunction with The 31st International Conference on Very Large Data Bases (VLDB).*

Baresi, L., Guinea, S., & Plebani, P. (2007). Policies and Aspects for the Supervision of BPEL Processes. *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE).*

Benatallah, B., Casati, F., Toumani, F., Ponge, J., & Nezhad, M. (2006). Service Mosaic: A Model-Driven Framework for Web Services Life-Cycle Management. *IEEE Internet Computing*, 10(4).

Benatallah, B., Sheng, Q. Z., & Dumas, M. (2003). The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1).

Benslimane, D., Maamar, Z., Taher, Y., Lahkim, M., Fauvet, M. C., & Mrissa, M. (2007). A Multi-Layer and

Multi-Perspective Approach to Compose Web Services. *Proceedings of The IEEE 21st International Conference on Advanced Information Networking and Applications (AINA).*

Benslimane, D., & Maamar, Z. (2007). Special Issue on Context-AwareWeb Services. *Distributed and Parallel Databases, Kluwer Academic Publishers*, 21(1).

Bentahar, J., Maamar, Z., Benslimane, D., & Thiran, Ph. (2007). An Argumentation Framework for Communities of Web Services. *IEEE Intelligent Systems*, 22(6), 75–83.

Bentahar, J., Maamar, Z., Benslimane, D., & Thiran, Ph. (2007). Using Argumentative Agents to Manage Communities of Web Services. *Proceedings of The International Workshop on Web and Mobile Information Systems (WAMIS) held in conjunction with The IEEE 21st International Conference on Advanced Information Networking and Applications (AINA).*

Bianchini, D., Antonellis, V. De., & Melchiori, M. (2005). Capability Matching and Similarity Reasoning in Service Discovery. *Proceedings of The Open Interop Workshop on Enterprise Modeling and Ontologies for Interoperability (EMOI-INTEROP) held in conjunction with The Seventh Conference on Advanced Information Systems Engineering (CAiSE).*

Arpinar, I. B., Aleman-Meza, B., Zhang, R., & Maduko, A. (2004). Ontology-DrivenWeb Services Composition Platform. *Proceedings of The IEEE International Conference on E-Commerce Technology (CEC).*

Bui, T., & Gacher, A. (2005). Web Services for Negotiation and Bargaining in Electronic Markets: Design Requirements and Implementation Framework. *Proceedings of The 38th Hawaii International Conference on System Sciences (HICSS).*

Cavedon, L., Maamar, S., Martin, D., & Benatallah, B. (2005). Introduction to The Extending Web Services Technologies: The Use of Multi-Agent Approaches. Multiagent Systems, Artificial Societies, and Simulated Organizations Series, *Kluwer Academic Publishers*, 13.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2).

Daniel, F., & Pernici, B. (2005). Insights into Web Service Orchestration and Choreography. *International Journal of E-Business Research, The Idea Group Inc*, 1(2).

Davies, J., Duke, A., & Sure, Y. (2003). OntoShare - A knowledge Management Environment for Virtual Communities. *Proceedings of The Second International Conference on Knowledge Capture (K-CAP).*

Dustdar, S., & Schreiner, W. (2005). A Survey on Web Services Composition. *International Journal on Web and Grid Services*, 1(1).

Elnaffar, S., Maamar, Z., Yahyaoui, H., Bentahar, J., & Thiran, P. (2008). Reputation of Communities of Web services - Preliminary Investigation. *Proceedings of the International Symposium on Web and Mobile Information Services (WAMIS) held in conjunction of the 22nd International Conference on Advanced Information Networking and Applications (AINA).*

Foster, H., Uchitel, J., Magee, S., & Kramer, J. (2006). LTSA-WS: A Tool forModel-based Verification of Web Service Compositions and Choreography. *Proceedings of The 28th International Conference on Software Engineering (ICSE).*

Foster, H., Uchitel, S., Magee, J., & Kramer, J. (2005). Tool Support for Model-Based Engineering of Web Service Compositions. *Proceedings of the IEEE International Conference on Web Services (ICWS).*

Noy, N. F., & Hafner, C. (1997). The State of the Art in Ontology Design: A Survey and Comparative Review.

AI Magazine, 18(3).

Maamar, Z., Benslimane, D., & Narendra, N. C. (2006). What Can Context do for Web Services? *Communications of the ACM*, 49(12).

Maamar, Z., Lahkim, M., Benslimane, D., Thiran, P., & Subramanian, S. (2007). Web Services Communities - Concepts & Operations -. *Proceedings of The 3rd International Conference on Web Information Systems and Technologies (WEBIST).*

Maximilien, M., & Singh, M. (2002). Concept Model of Web Service Reputation. *SIGMOD Record*, 31(4).

Maximilien, M., & Singh, M. (2004). Toward Autonomic Web Services Trust and Selection. *Proceedings of The 2nd International Conference on Service-Oriented Computing (ICSOC).*

Medjahed, B., & Atif, Y. (2007). Context-based Matching for Web Service Composition. *Distributed and Parallel Databases, Springer,* 21(1).

Medjahed, B., & Bouguettaya, A. (2005). A Dynamic Foundational Architecture for Semantic Web Services. *Distributed and Parallel Databases, Kluwer Academic Publishers*, 17(2).

Medjahed, M., Benatallah, B., Bouguettaya, B., & Elmagarmid, A. (2004). Webbis: An Infrastructure For Agile Integration of Web Services. *International Journal of Cooperative Information Systems*, 13(2).

Mrissa, M., Ghedira, C., Benslimane, D., Maamar, Z., Rosenberg, F., & Dustdar, S. (2008). A Context-based Mediation Approach to Compose Semantic Web Services. *ACM Transactions on Internet Technology*, 8(1).

Narendra, N. C. (2001). Flexible Agent Societies: Flexible Workflow Support for Agent Societies. *Proceedings of The International Conference on Intelligent Agents Web Technologies and Internet Commerce (IAWTIC).*

Ouzzani, M., & Bouguettaya, A. (2004). Efficient Access to Web Services. *IEEE Internet Computing,* 8(2).

Paik, H. Y., Benatallah, B., & Toumani, F. (2005). Toward self-organizing service communities. *IEEE Transactions on Systems, Man, and Cybernetics, Part A,* 35(3).

Subramanian, S., Thiran, P., Maamar, Z., & Benslimane, D. (2007). Engineering Communities of Web Services. *Proceedings of the 9th International Conference on Information Integration and Web-based Applications & Services (iiWAS).*

Schiaffino, S., & Amandi, A. (2004). User - Interface Agent Interaction: Personalization Issues. *International Journal of Human Computer Studies, Elsevier Sciences Publisher,* 60(1).

Smith, R. (1980). The Contract Net Protocol: High Level Communication and Control in Distributed Problem Solver. *IEEE Transactions on Computers*, 29.

Spencer, B., & Liu, S. (2004). What Does Software Engineering Practice Offer to Semantic Web Service Compostion. *Proceedings of The Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications held in conjunction with The Third International Semantic Web Conference (ISWC).*

Younas, M., Awan, I., & Duce, D. (2006). An Efficient Composition of Web Services with Active Network Support. *Expert Systems with Applications, Elsevier Science Publisher,* 31(4).

Appendix 1

The following illustrates the WSDL file of a master Web service with focus on mandatory ports and messages.

```
<definitions name="master-WS" ... >
  <message name="NewWSDetails">
    <part name="WSDL" element="xsd:String"/>
    <!--Others depend on the business requirement -->
  </message>
  <message name="RegisteredInfo">
    <part name="communityMembershipID" element="xsd:integer"/>
    <!--Others are depends on the business requirement -->
  </message>
  <message name="WithdrawInfo">
    <part name="WithdrawalAcceptence" element="xsd:boolean"/>
    <part name="WithdrawalAcceptenceRefID" element="xsd:integer"/>
    <!--Others are depends on the business requirement -->
  </message>
  <message name="Results">
    <!-- Others are depends on the business requirement -->
  </message>
  <portType name="MWS-ContractResults">
    <operation name="MWS-ContractResults">
      <input message="Results"/>
      <fault message="MWS-ContractResultsFault"/>
    </operation>
  </portType>
  <portType name="MWS-Registration">
    <operation name="MWS-Registration">
      <input message="NewWSDetails"/>
      <output message="RegisteredInfo"/>
      <fault message="MWS-RegistrationFault"/>
    </operation>
  </portType>
<portType name="MWS-Withdrawal">
    <operation name="MWS-Withdrawal">
      <input message="RegisteredInfo"/>
      <output message="WithdrawInfo"/>
      <fault message="MWS-WithdrawalFault"/>
    </operation>
  </portType>
  <binding> ... </binding>
  <service> ... </service>
</definitions>
```

Appendix 2

The following illustrates the WSDL file of a slave Web service with focus on mandatory ports and messages.

```
<definitions name="slave-WS" ... >
  <message name="RegisteredInfo">
    <part name="communityMembershipID" element=
    "xsd:integer"/>
    <!--Others are depends on the business requirement -->
  </message>
  <message name="WithdrawInfo">
    <part name="WithdrawalAcceptence" element="xsd:boolean"/>
    <part name="WithdrawalAcceptenceRefID" element="xsd:integer"/>
    <!--Others are depends on the business requirement -->
  </message>
  <message name="BidDetails">
    <part name="communityMembershipID" element="xsd:integer"/>
    <part name="serviceCost" element="xsd:decimal"/>
    <part name="QoS" element="xsd:string"/>
    <!--Others are depends on the business requirement -->
  </message>
  <message name="ContractAdvertisement">
    <part name="requiredService" element="xsd:string"/>
    <part name="dateAndTime" element="xsd:date"/>
    <part name="QoS" element="xsd:date"/>
    <!--Others are depends on the business requirement -->
  </message>
  <message name="ContractDecision">
    <part name="ContractFlag" element="xsd:boolean"/>
    <part name="contractID" element="xsd:integer"/>
    <!--Others are depends on the business requirement -->
  </message>
  <message name="PingInfo">
    <part name="communityMembershipID" element="xsd:integer"/>
    <part name="communityMasterID" element="xsd:integer"/>
    <!--Others are depends on the business requirement -->
  </message>
  <message name="LiveFlag">
    <part name="state" element="xsd:boolean"/>
    <!--Others are depends on the business requirement -->
  </message>
  <portType name="SWS-Withdrawal">
    <operation name="SWS-Withdrawal">
      <input message="RegisteredInfo"/>
      <output message="WithdrawInfo"/>
      <fault message="SWS-WithdrawalFault"/>
    </operation>
  </portType>
  <portType name="SWS-CallForProposal">
    <operation name="SWS-CallForProposal">
      <input message="ContractAdvertisement"/>
      <output message="BidDetails"/>
      <fault message="SWS-CallForProposalFault"/>
    </operation>
  </portType>
  <portType name="SWS-AwardWithContract">
    <operation name="SWS-AwardWithContract">
      <input message="ContractDecision"/>
      <fault message="SWS-AwardWithContractFault"/>
    </operation>
  </portType>
  <portType name="SWS-Liveness">
    <operation name="SWS-Liveness">
      <input message="PingInfo"/>
      <output message="LiveFlag"/>
      <fault message="SWS-LivenessFault"/>
    </operation>
  </portType>
  <binding> ... </binding>
  <service> ... </service>
</definitions>
```