Agent-Based Service Composition in Cloud Computing

J. Octavio Gutierrez-Garcia and Kwang-Mong Sim

Gwangju Institute of Science and Technology,
Gwangju 500-712 Republic of Korea
joseogg@gmail.com, kmsim@gist.ac.kr

Abstract. In a Cloud-computing environment, consumers, brokers, and service providers interact to achieve their individual purposes. In this regard, service providers offer a pool of resources wrapped as web services, which should be composed by broker agents to provide a single virtualized service to Cloud consumers. In this study, an agent-based test bed for simulating Cloud-computing environments is developed. Each Cloud participant is represented by an agent, whose behavior is defined by means of colored Petri nets. The relationship between web services and service providers is modeled using object Petri nets. Both Petri net formalisms are combined to support a design methodology for defining concurrent and parallel service choreographies. This results in the creation of a dynamic agent-based service composition algorithm. The simulation results indicate that service composition is achieved with a linear time complexity despite dealing with interleaving choreographies and synchronization of heterogeneous services.

Keywords: web service composition; Cloud computing; multi-agent systems.

1 Introduction

Cloud computing service composition must support dynamic reconfiguration and automatic reaction to new requirements as well as dealing with distributed, self-interested, and autonomous parties such as service providers, brokers, and Cloud consumers; these parties should interact and coordinate among themselves to achieve a proper service composition. This accentuates the need for an agent-based solution. Agents are autonomous problem solvers that can act flexibly (e.g., interacting with other agents through negotiation and cooperation) in a dynamic environment (e.g., a Cloud-computing environment).

Service composition can scale Cloud computing in two dimensions [3]: horizontal and vertical. Horizontal service composition refers to the composition of usually heterogeneous services that may be located in several Clouds. Vertical service composition deals with the composition of homogenous services to increase the capacity of a given Cloud node.

This work proposes an agent-based algorithm that supports both horizontal and vertical service compositions with linear time complexity (as demonstrated in section 4). In addition, each agent is only aware of the requirement it fulfills and possible requirements it may need from external agents. Furthermore, the algorithm handles the composition of atomic and complex web services. Moreover, a formal methodology for defining web services' workflows that handles synchronization and coordination aspects is defined through the use of colored Petri nets [2] and object Petri nets [5], which are endowed with a strong mathematical background.

The significance of this paper is that, to the best of the authors' knowledge, this work is the earliest research that adopts a multi-agent approach for supporting Cloud service composition.

This paper is structured as follows: Section 2 presents the agent-based architecture used to compose services. Section 3 presents Petri net models defined for the involved agents. Section 4 presents experimental results in both horizontal and vertical scenarios. Section 5 presents a comparison with related work and the conclusions.

2 An Agent-Based Cloud Service Composition Architecture

The multi-agent system architecture that supports Cloud computing service composition is presented in Fig. 1. The elements of the architecture are as follows:

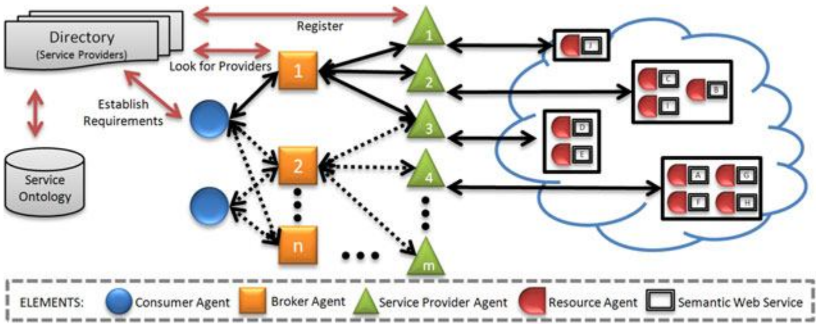
Directory (Service Providers)
Register
Look for Providers
Establish Requirements
Service Ontology
1
2
n
1
2
3
4
m
A
B
C
D
E
F
G
H
I
J
ELEMENTS: Consumer Agent Broker Agent Service Provider Agent Resource Agent Semantic Web Service

Fig. 1. Multi-agent system architecture

A *service ontology* is a formal representation of services and of the atomic requirements that these services can fulfill.

A *directory* is a listing of available service provider agents. Each entry associated to a service provider contains its name, location (e.g., URI address), and capabilities.

A *semantic web service* (SWS) is a web service whose definition is mapped to an ontology, which characterizes the service and its application domain.

A *consumer agent* (CA) submits a consumer's interests to broker agents.

A *resource agent* (RA) orchestrates a semantic web service. Furthermore, a RA contains a reference to an atomic requirement, which can be fulfilled by its SWS.

A *service provider agent* (SPA) handles a set of RAs. Its main function is to coordinate and synchronize RAs.

A *broker agent* (BA) accepts requests from consumer agents and creates a single virtualized service by means of composing services deployed on one or more Clouds. A BA utilizes consumer requirements to search for SPAs; it then starts the composition process by adopting the well-known contract net protocol [4] for selecting services and allocating tasks. However, an ad-hoc mechanism to collect results overrides the one provided by the contract net protocol. This ad-hoc mechanism synchronizes and coordinates interleaving service choreographies. In addition, as a result of the process for satisfying requirements, SPAs may need additional requirements, e.g., a computing provider may need a storage address to store its results. These requirements are handed to the BA. Further details of the composition algorithm are presented in section 3.

3 Petri Net Agent Models

In the context of agent interaction, transitions in colored Petri net models represent either the reception or transmission of messages, and internal actions performed by the agent. In contrast, places represent the interaction states.

3.1 Consumer Agents

The model of a consumer agent (Fig. 2) represents a Cloud consumer, whose actions and events are described as follows: (i) transition t_1 decomposes consumer requirements into a set of atomic requirements. *Input*: An initial mark from p_1. *Output*: A set of requirements (*Req*) in p_2; (ii) transition t_2 requests requirements to a BA. *Input*: A set of requirements (*Req*) from p_2; (iii) transition t_3 receives results from the BA. *Output*: A set of resolved requirements (*Req*, *Output*) in p_3; (iv) transition t_4 composes the atomic outputs into a single virtualized service. *Input*: A set of resolved requirements (*Req*, *Output*) from p_3. *Output*: A single virtualized service represented by an uncolored dot in p_4; (v) transition t_5 restarts the consumer agent's behavior. *Input*: An uncolored dot from p_4. *Output*: An uncolored dot (initial mark) in p_1.

The firing sequence $\sigma = \{t_1, t_2, t_3, t_4, t_5\}$ represents a complete cycle of the model, from providing consumer requirements to the reception of a single virtualized service.

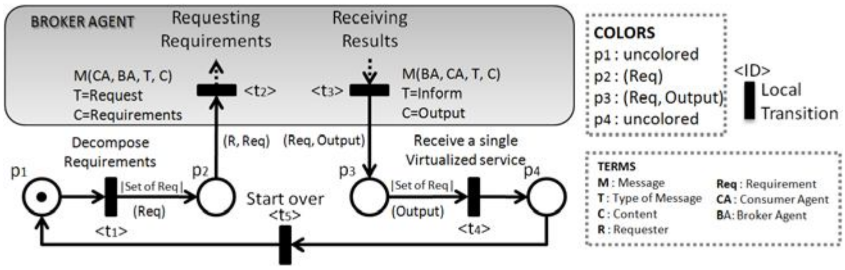

Fig. 2. Consumer agent model

3.2 Resource Agents

Resource agents are token objects, i.e., inner Petri nets contained in a system net. This follows the approach of Petri nets within Petri nets [5]. The system net is represented by the service provider agent model. The definition of resource agent models is limited to a set of design patterns that makes use of synchronized transitions to maintain a consistent behavior with respect to the system net that contains them.

The main structure of the resource agent model has two places and two synchronized transitions st_1 and st_2 (see Fig. 3). Transition st_1 synchronizes the beginning of the workflow with the reception of a request message from a SPA. This transition has a condition denoted by *if [Req = X*$_1$*]* in order to be triggered. The condition consists of accepting requirements that can be handled by the SWS in question. Transition st_2 reports the output to the outer level Petri net (a SPA).

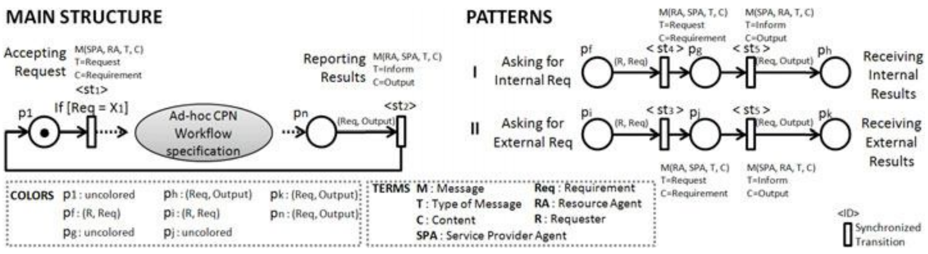
MAIN STRUCTURE
Accepting Request
M(SPA, RA, T, C) T=Request C=Requirement
<st1>
If [Req = X1]
p1
Ad-hoc CPN Workflow specification
pn
(Req, Output)
<st2>
Reporting Results
M(RA, SPA, T, C) T=Inform C=Output
PATTERNS
I Asking for Internal Req
II Asking for External Req
pf pg ph pi pj pk
< st4 > < st5 > < st3 >
(R, Req) (Req, Output)
M(RA, SPA, T, C) T=Request C=Requirement
M(SPA, RA, T, C) T=Inform C=Output
Receiving Internal Results
Receiving External Results
COLORS p1 : uncolored pf : (R, Req) pg : uncolored ph : (Req, Output) pi : (R, Req) pj : uncolored pk : (Req, Output) pn : (Req, Output)
TERMS M : Message T : Type of Message C : Content SPA : Service Provider Agent Req : Requirement RA : Resource Agent R : Requester
<ID> Synchronized Transition

Fig. 3. Resource agent model

In addition to the main structure, a resource agent model has two design patterns (Fig. 3). Design pattern *I* allows a RA to ask for an internal requirement (st_4) and wait until this requirement is resolved by another RA (st_5) belonging to the same SPA. Pattern *II* is used to ask for external requirements (st_3); i.e., these requirements cannot be resolved by any existing token object of the current service provider system net. In this case, the SPA requests the requirement to a BA, which searches for another SPA who can fulfill the requirement. Both patterns *I* and *II* share the synchronized transition st_5 for receiving results and proceeding with the internal workflow.

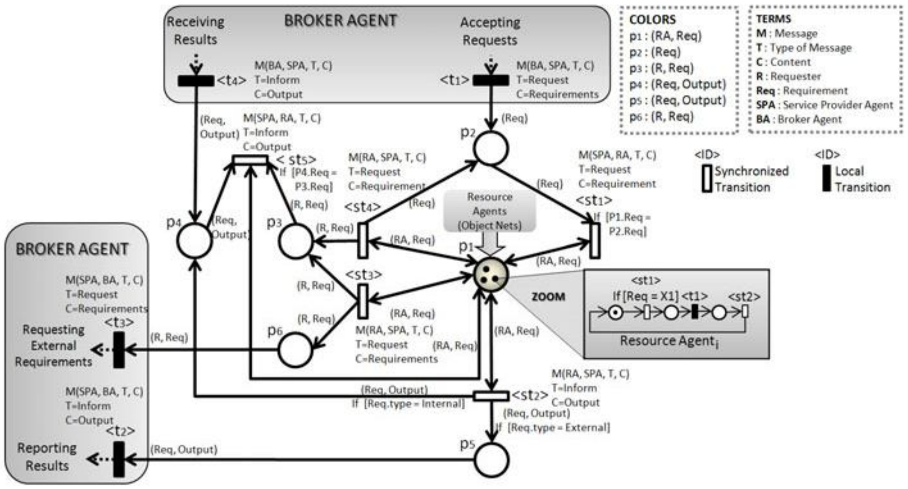
BROKER AGENT
Receiving Results
Accepting Requests
<t4> M(BA, SPA, T, C) T=Inform C=Output
<t1> M(BA, SPA, T, C) T=Request C=Requirements
COLORS p1 : (RA, Req) p2 : (Req) p3 : (R, Req) p4 : (Req, Output) p5 : (Req, Output) p6 : (R, Req)
TERMS M : Message T : Type of Message C : Content R : Requester Req : Requirement SPA : Service Provider Agent BA : Broker Agent
(Req, Output) M(SPA, RA, T, C) T=Inform C=Output
< st5 > If [P4.Req = P3.Req]
M(SPA, RA, T, C) T=Request C=Requirement
<st4> <st3> <st1> If [P1.Req = P2.Req]
Resource Agents (Object Nets)
p1 p2 p3 p4 p5 p6
(R, Req) (RA, Req) (Req)
<ID> Synchronized Transition
<ID> Local Transition
ZOOM
<st1> If [Req = X1] <t1> <st2>
Resource Agent_i
BROKER AGENT
M(SPA, BA, T, C) T=Request C=Requirements
<t3> Requesting External Requirements
M(SPA, BA, T, C) T=Inform C=Output
<t2> Reporting Results
M(RA, SPA, T, C) T=Request C=Requirements
M(RA, SPA, T, C) T=Inform C=Output
<st2>
(Req, Output) If [Req.type = Internal]
(Req, Output) If [Req.type = External]

Fig. 4. Service provider agent model

3.3 Service Provider Agents

The model of a service provider agent (Fig. 4) represents the system net that contains RAs. The SPA coordinates independent and evolving resource agent tokens by means of five synchronized transitions.

Transition st_1 synchronizes the assignment of unfulfilled requirements to available RAs. *Input*: It needs two tokens, one unfulfilled requirement (*Req*) from p_2, and one available resource agent (*RA*) from p_1. The selection of an appropriate *RA* is determined by the condition *if* $[p_1.Req = p_2.Req]$. In addition, a *RA* is available if its transition $RA.st_1$ is enabled. *Output*: An evolved resource agent token in p_1.

Transition st_2 releases a resource agent by receiving its resolved requirement. This sends the *RA* back to its initial state, ready to accept another requirement. *Input*: A *RA* whose synchronized transition $RA.st_2$ is enabled in p_1. *Output*: An available *RA* in p_1 and a resolved requirement (*Req*, *Output*), which is placed in either place p_4 or p_5. This is according to the conditions attached to the output arcs regarding the type of requirement *if [Req.type = Internal/External]*.

Transition st_3 receives request messages from RAs asking for external requirements. *Input*: A resource agent token *RA* whose synchronized transition $RA.st_3$ is enabled in p_1. *Output*: An evolved resource agent token in p_1, and a requirement and its requester (*R*, *Req*) in both place p_3 and p_6; the token in p_3 is used to keep a record, and the token in p_6 is used to send an external request.

Transition st_4 receives request messages from RAs asking for internal requirements. *Input*: A resource agent token *RA* whose synchronized transition $RA.st_4$ is enabled in p_1. *Output*: An evolved resource agent token in p_1; a requirement (*Req*) to be internally solicited in p_2; and a record of the request (*R*, *Req*) in p_3.

Transition st_5 coordinates the delivery of recently arrived and fulfilled requirements to the corresponding resource agents. *Input*: It needs three tokens, one fulfilled requirement (*Req*, *Output*) from p_4; one previous solicited requirement (*R*, *Req*) from p_3; and one *RA* from p_1. The preconditions are $[p_4.Req = p_3.Req]$ and that $RA.st_5$ is enabled in p_1. *Output*: An evolved resource agent token in p_1.

In addition, local transitions are used to interact with the BA:

Transition t_1 accepts requests from BAs. *Output*: A requirement (*Req*) in p_2.

Transition t_2 reports resolved requirements to BAs. *Input*: A resolved requirement (*Req*, *Output*) from p_5.

Transition t_3 requests requirements to BAs. *Input*: A request (*R*, *Req*) from p_6.

Transition t_4 receives outputs from previous requests. *Output*: A resolved requirement (*Req*, *Output*) in p_4.

A firing sequence $\sigma_a = \{t_1, st_1, st_2, t_2\}$ represents the simplest coordination process for a SPA that contains one resource agent. Transition t_1 accepts a request from a BA to fulfill a requirement. Then, transition st_1 assigns the new requirement to an available *RA*. Transition st_2 receives the resultant output from the *RA*, and finally, transition t_2 delivers the output to the *BA*. This firing sequence is assumed to be accompanied by a resource agent's firing sequence $\sigma_b = \{st_1, t_1, st_2\}$, which corresponds to the model of the resource agent contained in Fig. 4. The resultant interleaving firing sequence is $\sigma = \{\sigma_a.t_1, \sigma_a.st_1, \sigma_b.st_1, \sigma_b.t_1, \sigma_b.st_2, \sigma_a.st_2, \sigma_a.t_2\}$.

3.4 Broker Agents

The model of a broker agent (Fig. 5) functions as a mediator among service provider and consumer agents. Its main functionality is contained in the following transitions:

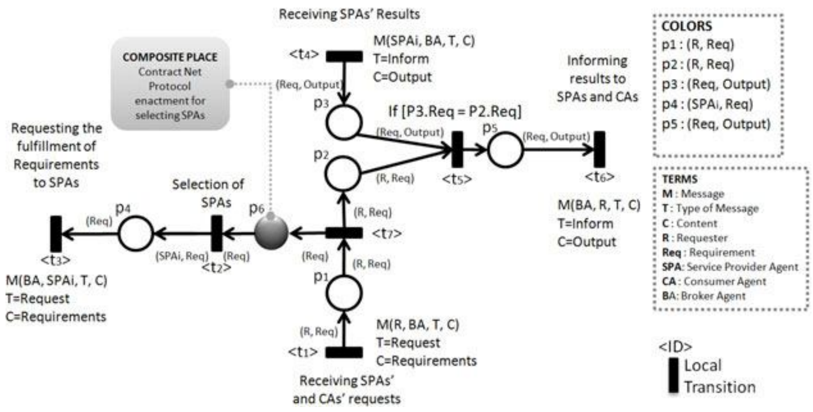

Fig. 5. Broker agent model

Transition t_1 receives requests from Cloud consumers. *Output*: An unresolved requirement (R, Req) in p_1.

Transition t_2 denotes the selection of SPAs based on the contract net protocol. The participants are all available SPAs who may fulfill the requirement(s) in question. *Input*: A requirement (Req) from p_6. *Output*: The selected SPA and the assigned requirement (SPA_i, Req) in p_4.

Transition t_3 represents the transmission of a request message to a SPA; the BA solicits the achievement of atomic requirements. *Input*: A requirement (Req) from p_4.

Transition t_4 receives outputs from previous requests. *Output*: A resolved requirement (Req, $Output$) in p_3.

Transition t_5 matches previous unfulfilled requests with incoming resolved requirements. *Input*: It needs two tokens, one unfulfilled requirement (R, Req) from p_2, and one resolved requirement (Req, $Output$) from p_3. Accompanied by the precondition *if* $[p_3.Req = p_2.Req]$. *Output*: A resolved requirement (Req, $Output$) in p_5.

Transition t_6 informs the results to either SPAs or CAs. *Input*: A resolved requirement (Req, $Output$) from p_5.

A firing sequence $\sigma = \{t_1, t_7, t_2, t_3, t_4, t_5, t_6\}$ represents a service composition solution to a consumer requirement as accepted by a BA (t_1). Subsequently, the requirement is assigned to an appropriate SPA (t_7, t_2, and t_3). Finally, the BA receives some provider's results (t_4), which are handed to the CA (t_5 and t_6).

4 Evaluation

A CA needs to apply several image filters to a huge amount of raw images; it then contacts a BA, which should contract several heterogeneous SPAs to satisfy that requirement. In this scenario, two different kinds of service providers are involved: computing service providers (CSP) and massive storage service providers (MSP).

A CSP has two types of resource agents: (i) an allocator service (AS) (Fig. 6(a)) and (ii) *n* processing services (PS) (Fig. 6(b)). The AS decomposes a computational task into several atomic tasks, which are handled by the PSs. Afterwards, the AS requests its service provider (a CSP) to assign the atomic tasks to the PSs and waits until all tasks are completed to join the outputs. At the same time, the AS divides the task, it computes the storage needed for saving the results, and asks the CSP for a storage address. In turn, the CSP passes the request to its BA, which again initiates the contract net protocol for contracting another SPA, which in this case is a MSP. The MSP only has one resource agent, a storage service (SS) (Fig. 6(c)) that provides a storage address; at the end, this is passed to the AS through the BA and corresponding service provider. Finally, the BA arranges the outputs of service providers and delivers a single virtualized service to the CA.

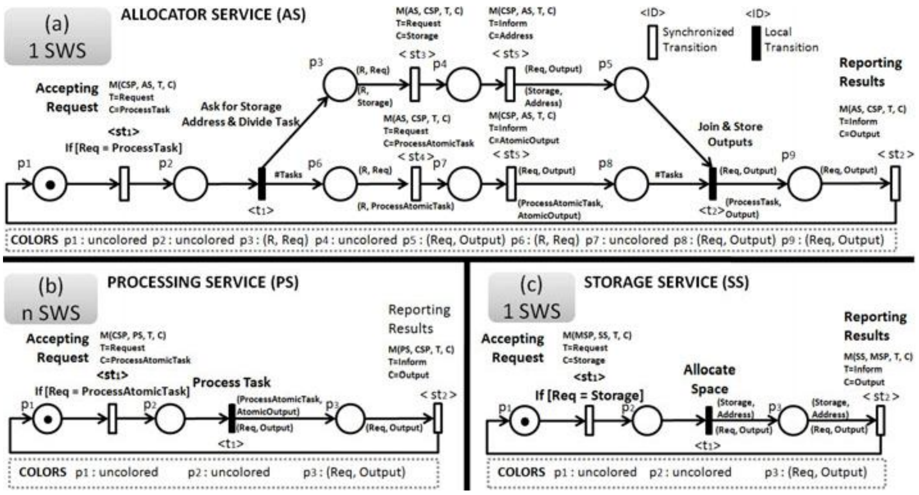
(a) 1 SWS
ALLOCATOR SERVICE (AS)
Synchronized Transition
Local Transition
Accepting Request
Ask for Storage Address & Divide Task
Join & Store Outputs
Reporting Results
COLORS p1: uncolored p2: uncolored p3: (R, Req) p4: uncolored p5: (Req, Output) p6: (R, Req) p7: uncolored p8: (Req, Output) p9: (Req, Output)
(b) n SWS
PROCESSING SERVICE (PS)
Process Task
COLORS p1: uncolored p2: uncolored p3: (Req, Output)
(c) 1 SWS
STORAGE SERVICE (SS)
Allocate Space
COLORS p1: uncolored p2: uncolored p3: (Req, Output)

Fig. 6. Resource agents involved in the example scenario

Three experiments were conducted using the agent-based test bed described in sections 2 and 3. The test bed was implemented using the JADE agent framework [1].

4.1 Vertical Service Composition Experiment (Part A)

Objective. The first experiment was designed to explore the scalability and synchronization capabilities of the composition algorithm in vertical scenarios.

Scenario and experimental settings. A consumer agent was assumed to submit a task that has as an attribute the number of processing resources required to complete it (consumption level). The composition of services is carried out with different consumption levels of the consumer's task starting from 5 to 15, 5 being the exact number of available services per CSP. Thus, when this number is exceeded, the synchronization process of the service provider takes place in order to wait for available services and then assign them to the remaining requirements. In this experiment, the following were involved: one consumer agent, one broker agent, two MSPs, each containing one storage service, and two CSPs, each containing one allocator service and five processing services.

Results and observations. The number of exchanged messages increased at a constant rate of 4 messages per processing resource needed (see Fig. 7(a)). This shows that vertical service composition was achieved with a linear time complexity. In addition, all services were synchronized properly even with an insufficient number of resources. The number of exchanged messages is the result of adding up the required messages for (i) registering service provider agents; (ii) accessing the directory of services; (iii) selecting services by means of adopting the contract net protocol among possible service providers; and (iv) composing the involved services.

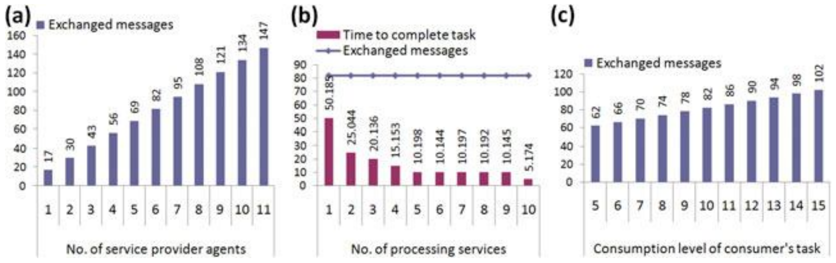

Fig. 7. Experimental results

4.2 Vertical Service Composition Experiment (Part B)

Objective. The second experiment was designed to explore the efficiency of the algorithm for parallelizing activities while dealing with heterogeneous services that may have different capacities (e.g., different processing rates).

Scenario and experimental settings. A consumer agent was assumed to submit a task that has the consumption level fixed to 10; however, in this case, the number of processing services contained in the CSPs was changed, starting from 1 to 10 processing services per CSP. In addition, the processing services were designed to spend 5 s processing each atomic task, while storage and allocator services did not take a considerable amount of time to execute their workflows. In this experiment, the following were involved: one consumer agent, one broker agent, two MSPs, each containing one storage service, and two CSPs, each containing one allocator service and n processing services, where $1 \leq n \leq 10$.

Results and observations. As Fig. 7(b) shows, (i) the number of necessary messages to compose services was constant even in the presence of scarce resources; and (ii) the allocator service efficiently exploited the available processing services by assigning the tasks in parallel. For instance, processing the consumer's task with four processing services, consumed 15 s; in the first 10 seconds, eight atomic tasks were processed by the four processing services, and in the remaining 5 s, just two processing services were used. This left 153 ms for composing and synchronizing. When the number of processing services was from 5 to 9, the consumed time was similar, and the milliseconds of difference were caused by message latencies. With ten resources, the ten atomic tasks were assigned in parallel, leaving 174 ms for the composition and synchronization processes. These results show that the proposed agent-based Cloud service composition algorithm handles the parallelization of tasks in a effective manner without involving additional messages. This parallelization was achieved even with heterogeneous resource agents having dissimilar times/capacities to fulfill the assigned requirements. In this regard, the execution of heterogeneous agents may evolve independently according to their capabilities and constraints.

4.3 Horizontal Service Composition Experiment

Objective. The third experiment was designed to explore the scalability and synchronization capabilities of the composition algorithm in horizontal scenarios.

Scenario and experimental settings. A consumer agent was assumed to require storing a huge amount of data, which must be stored by different service providers. It was first supposed that the data can be stored by one MSP, and then by two MSPs and so on until reaching eleven MSPs. In this experiment, the following were involved: one consumer agent, one broker agent, and n MSPs, each containing one storage service, where $1 \leq n \leq 11$.

Results and observations. The obtained results show that the number of exchanged messages increased at a constant rate of 13 messages per service provider added to the composition (see Fig. 7(c)). With this result, the agent-based service composition algorithm was shown to have a linear time complexity in both vertical and horizontal scenarios. Thus, it is suitable for handling the scalability necessary in Cloud-computing environments.

The specifications of the computer on which the experiments were carried out are as follows: Intel Core 2 Duo E8500 3.16 & 3.17 GHz, 4 GB RAM, with a Windows Vista Enterprise (32 bits) operating system, service pack 1.

5 Conclusions

5.1 Related Work

A previous research effort to achieve service composition in Cloud-computing environments was carried out in [7], which addressed service composition as a combinatorial optimization problem considering multi-Cloud environments, where each Cloud has a directory of available providers and their corresponding services. A search tree is created from these directories that maps the services deployed on the Cloud; artificial intelligence planning techniques are then applied to the tree in order to achieve the composition of atomic services. Another closely related work is [6], which presents a semantic matching algorithm that uses web services' descriptions to match the inputs and outputs of correlated web services.

Both approaches assume complete knowledge of all services deployed in different Clouds. Moreover, the composition process is centralized, and only atomic web services are considered. In contrast to [6] and [7], this work provides decentralized composition for both atomic and complex web services, which may require enacting an interaction protocol in order to fulfill their requirements.

5.2 Concluding Remarks

Throughout this research effort, the advantages of the agent paradigm as an underlying framework for supporting service composition in Cloud-computing environments were demonstrated. In addition, a Petri net–based methodology for defining web services' workflows capable of synchronizing concurrent and parallel execution of atomic and complex web services was developed. This design methodology provides a small set of requirements for assuring proper coordination and synchronization of web services in both horizontal and vertical scenarios.

Moreover, Cloud service composition is supported in a decentralized manner; no agent has dominant control over the others, and each agent knows only what it needs but not how to obtain it. Furthermore, RAs can be added dynamically even when a service composition is taking place due to the independent interfaces, which synchronize the acceptance and request of requirements. This fits well with the constantly changing Cloud infrastructure.

Finally, in the immediate future, work will be focus on deploying the agent-based architecture in a semantic web service framework using RESTful web services.

Acknowledgments. This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MEST) (KRF-2009-220-D00092) and the DASAN International Faculty Fund (project code: 140316).

References

1. Bellifemine, F., Poggi, A., Rimassa, G.: JADE - A FIPA-Compliant Agent Framework. In: 4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, pp. 97–108 (2008)
2. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. J. Softw. Tools Technol. Transf. 9(3), 213–254 (2007)
3. Mei, L., Chan, W.K., Tse, T.H.: A Tale of Clouds: Paradigm Comparisons and Some Thoughts on Research Issues. In: Proc. of the 2008 IEEE Asia-Pacific Services Computing Conference, pp. 464–469. IEEE Computer Society, Washington (2008)
4. Smith, R.G.: The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. IEEE Trans. Comput. 29(12), 1104–1113 (1980)
5. Valk, R.: Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–25. Springer, Heidelberg (1998)
6. Zeng, C., Guo, X., Ou, W., Han, D.: Cloud Computing Service Composition and Search Based on Semantic. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) CloudCom 2009. LNCS, vol. 5931, pp. 290–300. Springer, Heidelberg (2011)
7. Zou, G., Chen, Y., Yang, Y., Huang, R., Xu, Y.: AI Planning and Combinatorial Optimization for Web Service Composition in Cloud Computing. In: Proc. International Conference on Cloud Computing and Virtualization (2012)