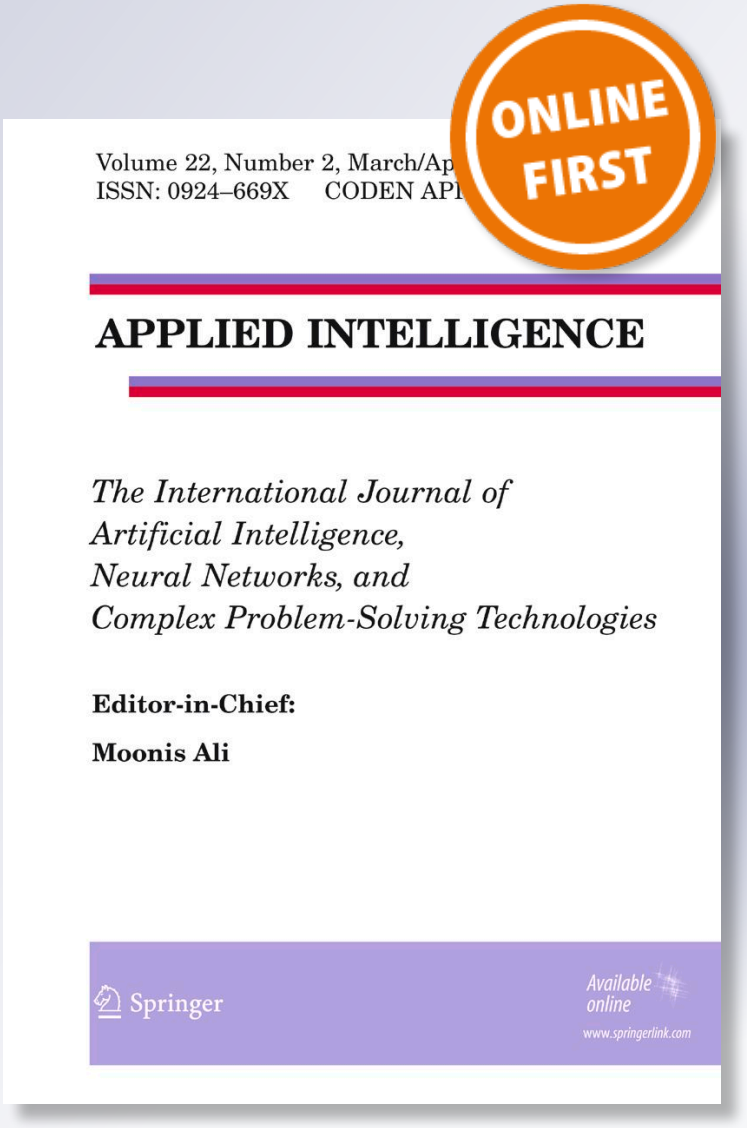
Agent-based Cloud service composition

J. Octavio Gutierrez-Garcia & Kwang Mong Sim

Applied Intelligence
The International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies

ISSN 0924-669X

Appl Intell
DOI 10.1007/s10489-012-0380-x



Springer

Appl Intell
DOI 10.1007/s10489-012-0380-x

Agent-based Cloud service composition

J. Octavio Gutierrez-Garcia · Kwang Mong Sim

Abstract Service composition in multi-Cloud environments must coordinate self-interested participants, automate service selection, (re)configure distributed services, and deal with incomplete information about Cloud providers and their services. This work proposes an agent-based approach to compose services in multi-Cloud environments for different types of Cloud services: one-time virtualized services, e.g., processing a rendering job, persistent virtualized services, e.g., infrastructure-as-a-service scenarios, vertical services, e.g., integrating homogenous services, and horizontal services, e.g., integrating heterogeneous services. Agents are endowed with a semi-recursive contract net protocol and service capability tables (information catalogs about Cloud participants) to compose services based on consumer requirements. Empirical results obtained from an agent-based testbed show that agents in this work can: successfully compose services to satisfy service requirements, autonomously select services based on dynamic fees, effectively cope with constantly changing consumers' service needs that trigger updates, and compose services in multiple Clouds even with incomplete information about Cloud participants.

Keywords Agent-based Cloud computing · Autonomous agents · Cloud computing · Cloud resource management · Cloud service composition · Multiagent systems

J.O. Gutierrez-Garcia
Computer Science Department, Instituto Tecnológico Autónomo de México, 1 Río Hondo, Progreso Tizapán, Mexico City, D.F. 01080, Mexico

K.M. Sim (✉)
School of Computing, The University of Kent, Chatham Maritime, Kent ME4 4AG, UK
e-mail: prof_sim_2002@yahoo.com

Published online: 05 September 2012

1 Introduction

Cloud computing is a collection of web-accessible resources, provisioned under service-level agreements established via negotiation, that should be dynamically composed and virtualized based on consumers' needs [11] on an on-demand basis [50]. In addition, there is an increasing number of Cloud providers (e.g., GoGrid [14], Amazon [3], and Google [18]) and the services offered by Cloud providers (e.g., Software-as-a-Service applications [13] and computing resources [49]) have also increased. There are also increasing demands for Cloud services from consumers. Hence there is a need for dynamic and automated Cloud service composition that can support an everything-as-a-service model [31, 38] capable of satisfying complex consumer requirements as they emerge. Due to this, Cloud service composition in single and multiple Cloud-computing environments must support: (i) coordination of independent and self-interested parties, e.g., Cloud consumers and service providers, (ii) service selection based on dynamic market-driven fees associated to Cloud services (e.g., Amazon EC2 spot instances [2]), (iii) efficient reconfiguration of existent and permanent service compositions, given constantly changing consumer requirements, (iv) dealing with incomplete knowledge about the existence of Cloud participants, and the services they provide, due to the distributed nature of Cloud-computing environments, and (v) dynamic and automated composition of distributed and parallel services.

The new challenges that Cloud computing poses to service composition, emphasizes the need for the agent paradigm [15, 48, 52, 53]. Agents are independent problem solvers (e.g., Cloud participants) that may collaborate to achieve global objectives (e.g., service composition) while simultaneously considering both individual goals and con-

straints (e.g., utility maximization) [52]. A multiagent system is a collection of autonomous, interacting, and cooperative agents [15, 53] that react to events (e.g., changes in consumer requirements) and may self-organize by means of interaction, negotiation (see [30]), coordination (see [34, 37]), cooperation (see [16]), and collaboration (see [27]). In addition, in a multiagent system, an agent may have incomplete information (e.g., information about Cloud providers and their services) as well as an incomplete list of capabilities (e.g., mapping consumer requirements to available Cloud resources or offering for leasing Cloud resources) for solving a given problem (e.g., Cloud service composition) [48]. Moreover, agent-based approaches have proved to be effective and efficient in a comprehensive range of application areas, from assisting humans in daily activities [9, 26, 33] and processing natural language [36] to supporting Grid and Cloud resource management [27, 41–45].

It was noted in [42–45] that agents are appropriate tools for automating Cloud resource management such as autonomous resource mapping and dealing with changing requests of consumers. Agent-based Cloud computing—the idea of adopting autonomous agents for managing Cloud resources was first introduced and proposed in [42–45]. Whereas [44] presented the challenging problems in Cloud resource management and some agent-based approaches for solving these problems, [42] presented the idea of agent-based Cloud commerce, and [43, 45] proposed a Cloud negotiation model, including the negotiation protocols and strategies of agents to facilitate the establishment of service-level agreements among Cloud participants. This research is among the earliest efforts, to the best of the authors' knowledge, in adopting an agent-based distributed problem solving approach for supporting Cloud service compositions.

Cloud service composition may be augmented in two dimensions [32]: Horizontal and vertical. Horizontal service composition deals with the combination and integration of heterogeneous services, e.g., storage, compute, cryptography services, etc. Vertical service composition involves the integration of homogenous services, e.g., augmenting storage capacity by adding new storage data centers.

Cloud service composition may be carried out in two modalities: One-time and persistent. One-time service compositions consider Cloud resources as functions, which receive consumer requirements as input parameters, and return the corresponding output. Once the output is generated, no link between consumers and providers remains, for instance, a consumer submitting a rendering job. Persistent service compositions create a virtualized service that is assumed to be accessed/used by consumers for a (long) predefined time, for instance, infrastructure-as-a-service scenarios. Both one-time and persistent service compositions may be augmented in horizontal and vertical dimensions.

Self-organizing systems are composed of entities or agents that interact [25] to evolve and adapt the structure and objective of the system according to incoming events [17]. Through interaction, agents receive feedback (e.g., service fees) that determines the entities to be connected with (e.g., providers to be contracted). System events (e.g., submission of consumer requirements) are handled by surrounding agents (e.g., Cloud consumers and broker agents) that replicate their effects to interconnected and close agents (e.g., service providers). In this manner, the evolution of the system is based on local rules that decentralize the management of the design objective (e.g., service composition).

In this present research, a self-organizing agent-based service composition method is proposed. Cloud participants are represented by agents, and Cloud resources are wrapped and controlled by agents. In addition, a set of agent behaviors to handle the coordination of self-interested parties are defined. Agents are endowed with service capability tables (SCTs) to register and consult information about Cloud services, Cloud participants, and their current status, e.g., unreachable, failed, busy, etc. In addition, a semi-recursive contract net protocol (SR-CNP) enhanced with SCTs is devised. The SR-CNP is based on the contract net protocol [46], which is a distributed problem solving technique used for (i) task assignment and (ii) establishing service contracts. In the contract net protocol, the task assignment takes place among managers with tasks to be executed and contractors capable of executing such tasks. Contractors may charge (differently) for performing tasks. So, a negotiation between managers and contractors is carried out by adopting a bidding scheme where managers announce tasks and contractors submit bids on the announced tasks. Then, the managers select the best bids and notify the decisions to all the contractors, see [40] for an application example of the contract net protocol. The SR-CNP not only combines the features of SCTs and the contract net protocol, it also considerably augments and extends the contract net protocol by (i) selecting contractor agents based on SCTs to focus the interaction with feasible contractors, i.e., service providers capable of carrying out a given task, (ii) allowing agents playing contractor roles to play multiple manager roles derived from their involvement as contractors, (iii) propagating and integrating results obtained from the multiple instantiations of manager roles in the context of the same protocol run, and (iv) reacting to failures by updating the status of agents in the SCTs (e.g., agents sending failure messages are labeled as failed agents) and re-instantiating manager roles with the remaining feasible participants. SR-CNP is used to (i) cope with incomplete information about the structure of distributed Cloud-computing environments, and (ii) handling dynamic service fees. Self-organization capabilities are integrated into agent behaviors to efficiently support constantly changing consumer requirements in persistent service compositions. Horizontal and vertical Cloud service compositions in both one-time and persistent modalities are achieved in an autonomous and dynamic manner.

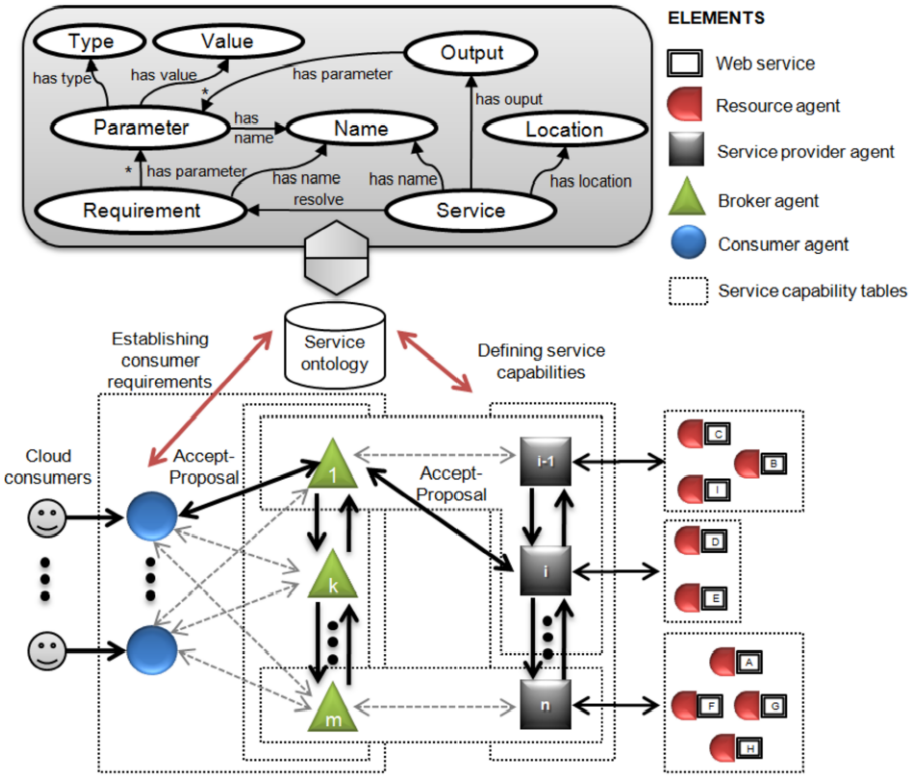


Fig. 1 Agent-based Cloud service composition architecture

The novelty and significance of this research is that, to the best of the authors' knowledge, it is the earliest effort in providing an agent-based approach for dealing with one-time, persistent, vertical, and horizontal Cloud service compositions. The novel features included in this paper are:

(1) Designing and implementing an agent-based testbed that supports vertical, horizontal, one-time and persistent service composition (Sect. 2).
(2) Deploying agent-based distributed problem solving techniques in single and multiple Cloud-computing environments, which are integrated into the SR-CNP (Sect. 2.2) enhanced with SCTs (Sect. 2.1) for (i) coping with incomplete information and (ii) dynamic service selection based on market-driven service fees.
(3) Integrating self-organization capabilities into agent behaviors (Sect. 3) to efficiently create and update Cloud service compositions.
(4) Providing experimental evidence (Sect. 4) to demonstrate the effectiveness of agent-based techniques in the creation of horizontal, vertical, one-time, and persistent Cloud service compositions by conducting experiments: (i) To evaluate the self-organization capabilities of agents. (ii) To compare the proposed agent-based service composition approach using SCTs, where agents have incomplete information about Cloud participants, and using a central directory with complete knowledge. (iii) To evaluate the reaction of agents to constantly changing consumer requirements in persistent service compositions.

In addition, Sect. 5 presents a comparison with related work, and Sect. 6 includes some concluding remarks and describes some future works.

2 Agent-based Cloud service composition architecture

The agent-based architecture (Fig. 1) is composed of six elements: Service ontology, web services, resource agents (RAs), service provider agents (SPAs), broker agents (BAs), and consumer agents (CAs).

(1) *Web services* are interfaces to remote-accessible software or (Cloud) resources.
(2) *The service ontology* (Fig. 1) provides the service specification that describes the functionality, input and output of services. A web service is described by the requirement it resolves, and the parameters of the requirement correspond to the input of the service. The service output is a set of parameters that results from resolving the requirement. The locations of web services are expressed as URI addresses. The following is an example of a web service definition:

```
<service>
        <name> AllocatorService </name>
        <location> http://api.server.com/allocatorService </location>
        <requirement>
                <name> allocateCPU </name>
                <parameter> <type> I/O performance </type> <value> Level </value></parameter>
                <parameter> <type> PlatformType </type> <value> Bits </value> </parameter>
        </requirement>
        <output>
                <parameter><type>URIlocation</type><value>http://api.server.com/cpu32</value></parameter>
        </output>
</service>
```

(3) *Resource agents* orchestrate web services and control the access to them. RAs receive requests to resolve requirements from service providers. Then, RAs handle the requests via their associated web service, returning the output to the service provider. In addition, RAs are used to orchestrate web services and control the access to them to adopt W3C's standpoint that in [54] states that web services should be implemented by agents.
(4) *Service provider agents* manage Cloud providers' resources by controlling and organizing RAs. This function is divided into: (i) offering for leasing Cloud resources to brokers, (ii) allocating/releasing Cloud resources whenever transactions are agreed, (iii) directing and delegating brokers' requirements to appropriate RAs, (iv) keeping track of available resources, (v) synchronizing the execution of concurrent and parallel RAs, and (vi) establishing service contracts with brokers. In addition, SPAs' functions are designed to endow SPAs with capabilities to act on behalf of Cloud providers.
(5) *Broker agents* compose and provide a single virtualized service to Cloud consumers. This is achieved through: (i) receiving consumer requirements, (ii) selecting and contacting a set of possibly heterogeneous service providers, (iii) managing parallel agent conversation contexts that have effect on one or more service contracts (service-level agreements), and (iv) handling consumers' update requests of persistent service compositions. In addition, since Cloud service composition can be carried out in multi-Cloud environments, BAs act as an intermediary between Cloud consumers and SPAs to compose and provide a single virtualized service to Cloud consumers from multiple Cloud providers.
(6) *Consumer agents*' functions are: (i) receiving and mapping consumer requirements to available Cloud resource types, (ii) submitting service composition requests to BAs, (iii) selecting the best (cheapest) BA, (iv) receiving and handling the single virtualized service provided by BAs to Cloud consumers, and (v) submitting update requests of persistent service compositions to contracted BAs. In addition, CAs' functions are designed to endow CAs with capabilities to act on behalf of Cloud consumers.

2.1 Service capability tables (SCTs)

SCTs are used by agents to register and consult information about Cloud participants, Cloud services, and their status, e.g., (i) whether a service is able to perform one of its capabilities at a given moment for the context of a Cloud service composition and (ii) whether the service is busy, etc. Whereas using SCTs to record Cloud service capabilities relates to the idea of using acquaintance networks [24], to record a list of acquainted agents and their functionalities, SCTs and acquaintance networks differ in (i) the information stored, SCTs extends acquaintance networks by keeping records of the status of agents, which are modified as a result of agent interaction, (ii) SCTs are more volatile, i.e., the status of agents is constantly updated as agent interaction takes place, unlike acquaintance networks that may be updated only when new agents enter to a system.

Agents may be included in SCTs because of previous encounters, recommendation lists, or merely because they can see each other, e.g., agents in the same Cloud. SCTs are dynamic, exact and incomplete: dynamic, because agents can be removed or added; exact, because the agents' addresses and functionalities are correct; and incomplete, because agents may be unaware of the full list of existent agents.

The functionalities of agents are represented by the consumer requirements they resolve. Thus, the records of SCTs are composed of: (i) agents' addresses, (ii) the requirements that agents can resolve, and (iii) the last known status of the service.

Each CA has one SCT that records a set of BAs' addresses and their status. However, the functionalities of BAs are left unspecified given that BAs are designed to resolve any set of Cloud requirements through collaboration with other agents. The possible status types for BAs are: *available*, *unreachable*, and *failed*. A BA is (i) *available* when it

responds to CAs' requests, (ii) *unreachable* when it doesn't respond to CAs' requests, and (iii) *failed* when it is unable to satisfy CA's requests even though it has the capabilities to do so.

Each BA has two SCTs that record information about SPAs and other BAs. The SCT of service providers records: (i) the address of providers, (ii) the list of requirements that the providers can resolve, and (iii) the status of providers. The SCT of BAs only records other BAs' addresses and their status, similar to the CAs' case. When a BA receives a request to carry out a Cloud service composition, if it is possible, the BA contracts its SPAs, otherwise the BA subcontracts services to other BAs for fulfilling the unresolved requirements (details are given in Sect. 3.2). The possible status types for both BAs and SPAs are: *available*, *unreachable*, and *failed*.

Each SPA has two SCTs that record information about RAs and other SPAs. The SCT of RAs is complete (unlike the previous SCTs) given that SPAs are aware of the Cloud resources they provide. The SCT of RAs is used to delegate unresolved requirements to appropriate RAs, and it can be updated as providers acquire or remove Cloud resources. The SCT with information about SPAs is used to subcontracting services to other SPAs. A SPA may subcontract services to other SPAs when (i) its RAs fail, and (ii) when its RAs, as the normal process of resolving a given requirement, request from its SPA the fulfillment of an external requirement. For example, RAs contained in storage service providers may ask for cryptographic services. This was adopted from the object-oriented approach, to allow the abstraction and encapsulation of SPAs' functions, and whenever uncommon, but previously considered requirements come out, SPAs can collaborate with each other to resolve interrelated requirements. The possible status types for RAs are: *available*, *unreachable*, *failed*, and *busy*. An RA is *busy* when it is executing a task previously assigned by its SPA.

2.2 The semi-recursive contract net protocol (SR-CNP)

A problem whose solution can be obtained from the solution to smaller instances of the same problem is a recursive problem [19]. An agent interaction protocol is a communication pattern established among agents with potentially different roles to attain a design objective [6]. Then, a semi-recursive (i.e., to some extent recursive) agent interaction protocol is a protocol that attains its design objective (e.g., composing a set of Cloud consumer requirements) by re-instantiating its communication pattern within itself to solve smaller instances of its design objective (e.g., composing a subset of the Cloud consumer requirements).

The underlying structure of SR-CNP (as a semi-recursive agent interaction protocol) is based on (recursive calls of) the contract net protocol [46], which is a distributed problem solving technique used for establishing service contracts among consumers and contractors. The contract net protocol was selected as the underlying structure of SR-CNP because it allows a cost-based Cloud resource selection from a heterogeneous set of Cloud providers.

Agents in the contract net protocol have two roles: initiator and participant. A consumer adopting the initiator role broadcasts a *call-for-proposals* to achieve a task (e.g., service composition) to n participants (contractors). The participants may reply with: (i) a proposal (quotation) to carry out the task, or (ii) a *refuse* message. From the received m proposals, the initiator selects the best (cheapest) proposal, and sends: (i) an *accept-proposal* message to the best participant, and (ii) *reject-proposal* messages to the remaining $m - 1$ contractors. After carrying out the task, the selected participant sends either: (i) an *inform-result* message or (ii) a *failure* message in case of unsuccessful results.

The SR-CNP extends the contract net protocol by (i) focusing the contracting process, interacting only with feasible contractors, e.g., service providers capable of carrying out a given task, (ii) allowing agents playing participant roles to play multiple initiator roles derived from their involvement as participants (contractors), e.g., accepting to perform a set of jobs (Cloud requirements) as a participant, creates multiple and concurrent instantiations of initiator roles to subcontract as many contractors as needed to execute each job (Cloud requirement), (iii) propagating and integrating results obtained from the multiple instantiations of initiator roles in the context of the same protocol run (Cloud service composition), and (iv) reacting to failures by updating the status of agents in the SCTs (e.g., agents sending failure messages are labeled as failed agents) and re-instantiating initiator roles with the remaining feasible participants.

The SR-CNP follows a divide-and-conquer strategy that allows agents (with incomplete but complementary information about the Cloud-computing environment) to work together to achieve a complex objective (e.g., Cloud service composition). In addition, agents adopting the SR-CNP can select and contract services based on dynamic fees.

In a given context, CAs, BAs, and SPAs can adopt the SR-CNP initiator role, but only BAs and SPAs can take the role of participant (details are given in Sect. 3).

3 Agent behaviors

The tasks that agents can perform are defined as behaviors. In implementation terms, an agent is defined as a single execution thread in which a set of *behavior* objects are instantiated. Each agent behavior handles an agent task that may

Table 1 Summary of agent behaviors

Agent	Behavior identifier	Main function
CA	*SR-CNPinitiatorCA*	To submit service composition call-for-proposals to BAs
	ServiceAugmenterCA	To submit requests for incremental updates
	ServiceRevokerCA	To submit requests for subtractive updates
	ContractChangeMonitor	To receive expired contracts' notifications
BA	*SR-CNPparticipantBA*	To handle consumers' service composition requests either from CAs or other BAs
	SR-CNPinitiatorBA	To submit call-for-proposals to resolve requirements to SPAs and service composition requests to other BAs
	ResultHandlerBA	To virtualize the service composition
	ServiceRevokerBA	To submit requests for subtractive updates
	ContractChangeMonitor	To receive expired contracts' notifications
SPA	*SR-CNPparticipantSPA*	To handle BAs' and/or other SPAs' service requests
	ReqAssignerSPA	To assign requirements to RAs
	IntermediarySPA	To handle requests to resolve requirements from RAs
	SR-CNPinitiatorSPA	To submit call-for-proposals to resolve requirements to SPAs
	ResultHandlerSPA	To receive results from both RAs and contracted SPAs
	ServiceRevokerSPA	To carry out subtractive updates
	ContractExpirationMonitor	To detect expired contracts
RA	*MainStructureRA*	To resolve requirements by orchestrating a web service
	RequesterRA	To request external requirements to SPAs
	ReleaserRA	To release Cloud resources

be activated in response to different events (e.g., reception of a *call-for-proposals* message). To handle multiple concurrent actions, a set of behaviors can be added, stopped or removed from a pool of agent behaviors. A set of agent behaviors is defined for each type of agent: CAs, BAs, SPAs, and RAs. See Table 1 for a summary of agent behaviors and their functions.

CAs, BAs, SPAs, and RAs interact among each other to compose and manage persistent, one-time, vertical, and horizontal Cloud services by adopting diverse agent behaviors (Fig. 2).

CAs interact with BAs (Fig. 2) by adopting (i) a *SR-CNPinitiatorCA* behavior (the initiator role of SR-CNP) to submit service composition requests, (ii) a *ServiceAugmenterCA* behavior to submit requests for incremental updates, (iii) a *ServiceRevokerCA* behavior to submit requests for subtractive updates, and (iv) a *ContractChangeMonitor* behavior to receive expired contracts' notifications.

BAs interact with CAs, other BAs, and SPAs (Fig. 2). BAs interact with CAs (Fig. 2) by adopting (i) a *SR-CNPparticipantBA* behavior (the participant role of SR-CNP) to handle and bid for CAs' service composition requests, and (ii) a *ResultHandlerBA* behavior to deliver single virtualized Cloud services. BAs interact with other BAs (Fig. 2) by adopting (i) a *SR-CNPinitiatorBA* behavior (the initiator role of SR-CNP) to submit service composition requests when subcontracting services is required, (ii) a *SR-CNPparticipantBA* behavior (the participant role of SR-CNP) to handle and bid for other BAs' service composition requests, (iii) a *ResultHandlerBA* behavior to receive outcomes from other BAs fulfilling requirements, and (iv) a *ContractChangeMonitor* behavior to send expired contracts' notifications to other BAs. BAs interact with SPAs (Fig. 2) by adopting (i) a *SR-CNPinitiatorBA* behavior to submit *call-for-proposals* to resolve requirements, (ii) a *ResultHandlerBA* behavior to receive outcomes from SPAs fulfilling requirements, (iii) a *ServiceRevokerBA* behavior to submit requests for subtractive updates, and (iv) a *ContractChangeMonitor* behavior to receive expired contracts' notifications.

SPAs interact with BAs, other SPAs, and RAs (Fig. 2). SPAs interact with BAs (Fig. 2) by adopting (i) a *SR-CNPparticipantSPA* behavior (the participant role of SR-CNP) to handle and bid for BAs' service composition requests, (ii) a *ResultHandlerSPA* behavior to deliver outcomes from RAs fulfilling requirements, (iii) a *ServiceRevokerSPA* behavior to handle requests to release Cloud resources, and (iv) a *ContractChangeMonitor* behavior to send expired contracts' notifications. SPAs interact with other SPAs (Fig. 2) by adopting (i) a *SR-CNPinitiatorSPA* behavior (the initiator role of SR-CNP) behavior to submit *call-for-proposals* to resolve requirements to SPAs

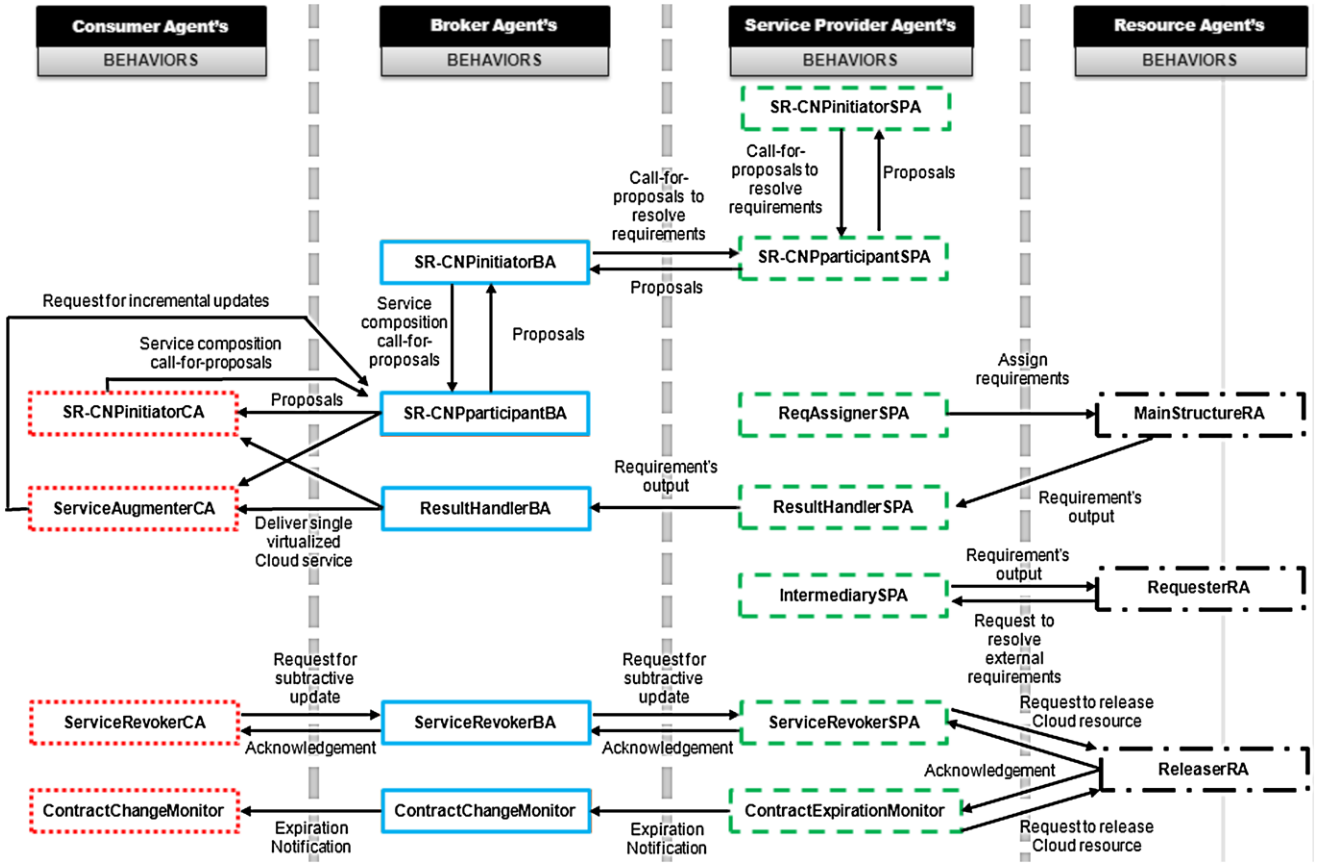


Fig. 2 Interaction diagram of agent behaviors

when subcontracting services are required, (ii) a *SR-CNPparticipantSPA* behavior (the participant role of SR-CNP) behavior to handle and bid for other SPAs' requests to resolve requirements, (iii) a *ResultHandlerSPA* behavior to receive outcomes from other SPAs fulfilling requirements, (iv) a *ServiceRevokerSPA* behavior to handle requests to release Cloud resources, and (v) a *ContractChangeMonitor* behavior to send expired contracts' notifications to other SPAs. SPAs interact with RAs (Fig. 2) by adopting (i) a *ReqAssignerSPA* behavior to assign requirements to RAs, (ii) a *ResultHandlerSPA* behavior to receive outcomes from RAs fulfilling requirements, (iii) an *IntermediarySPA* behavior to handle requests to resolve requirements from RAs, (iv) a *ServiceRevokerSPA* behavior to submit requests to release Cloud resources, and (v) a *ContractExpirationMonitor* behavior to submit requests to release Cloud resources.

RAs interact with SPAs (Fig. 2) by adopting a (i) *MainStructureRA* behavior to resolve Cloud requirements and deliver the outcome to SPAs, (ii) a *RequesterRA* behavior to request requirements to SPAs, and (iii) a *ReleaserRA* behavior to receive requests to release Cloud resources.

3.1 Consumer agent behaviors

CAs are endowed with four behaviors: *SR-CNPinitiatorCA*, *ServiceAugmenterCA*, *ServiceRevokerCA*, and *ContractChangeMonitor*.

The *SR-CNPinitiatorCA* behavior (Behavior 1) is for decomposing and mapping consumer requirements to atomic requirements (line 1). In addition to submitting service composition requests to BAs (line 2), which are obtained from the SCT, the selection of BAs is based on the best (cheapest) proposal (line 4). Once a BA is selected, a contract (service-level agreement) is established between both parties (line 7), and the CA waits for the single virtualized service to be delivered (line 8). Finally, the virtualized service is ready to be consumed (line 9).

Two important timeouts are involved in the SR-CNP: *timeout1* and *timeout2*. The *timeout1* (line 3) refers to the proposal submission deadline, while *timeout2* (line 8) refers to the deadline to deliver the single virtualized service composition. If any of the two deadlines are missed, exceptions are thrown (lines 12 & 15), the contract is removed and the status of the BA is changed to *failed* (line 11) or unreachable (line 14), and the behavior is halted.

Behavior 1 SR-CNPinitiatorCA

Input: (i) Consumer requirements (ii) contractors (BAs)
Output: (i) A single virtualized service
1: decomposing requirements into atomic requirements
2: CA sends *call-for-proposals* to resolve requirements to *m* feasible BAs recorded in the SCT
3: **if** (BlockReceive(Proposals, timeout1)) **then**
4: evaluate proposals
5: CA sends *reject-proposal* to (nProposals-1) BAs
6: CA sends *accept-proposal* to 1 BA
7: create contract between CA and BA
8: **if** (BlockReceive(virtualized service, timeout2)) **then**
9: consume virtualized service
10: **else**
11: remove contract and Update status of the BA to *failed* in the SCT.
12: throw exception
13: **Else**
14: update status of the BAs to *unreachable* in the SCT.
15: throw exception

Behavior 2 ServiceAugmenterCA

Input: (i) New consumer requirements (ii) contract ID
Output: (i) An augmented single virtualized service
1: decomposing new requirements into atomic requirements
2: BA ← get previously contracted BA based on contract ID
3: CA sends *call-for-proposals* to resolve new requirements to the previously contracted BA
4: **if** (BlockReceive(Proposal, timeout1)) **then**
5: evaluate proposal
6: **if** (proposal is accepted) **then**
7: CA sends *accept-proposal* to BA
8: update the contract with the newly contracted requirements
9: **if** (BlockReceive(virtualized service, timeout2)) **then**
10: consume virtualized service
11: **else**
12: revoke changes to the contract and Update status of the BA to *failed* in the SCT
13: throw exception
14: **else**
15: CA sends *reject-proposal* to BA
16: **else**
17: update status of the BA to *unreachable* in the SCT
18: throw exception

CAs can update requirements for persistent service compositions. There are two types of updates: incremental and subtractive.

Incremental updates are carried out using the *ServiceAugmenterCA* behavior (Behavior 2). To update the requirements for a service composition, its corresponding contract identifier should be provided as well as the new requirements to be added (line 1). The *ServiceAugmenterCA* behavior follows a similar structure to the *SR-CNPinitiatorCA* behavior, because CAs should receive a quotation that indicates the cost derived from updating requirements of service compositions. The modifications in the behavior are: (i) the *call-for-proposals* message is sent only to the contracted BA (lines 2 & 3), because contracting a different BA implies moving the existent persistent composition, and it could be computationally expensive and impractical, and (ii) the new clauses are added to the service contract (line 8).

Subtractive updates are carried out using the *ServiceRevokerCA* behavior (Behavior 3). To remove requirements from a service composition, its corresponding contract iden-

Behavior 3 ServiceRevokerCA

Input: (i) Consumer requirements to be removed R (ii) contract ID
Output: (i) A reduced single virtualized service
1: BA ← get previously contracted BA based on contract ID
2: CA sends *request* to remove R requirements to the BA
3: **if** (BlockReceive(Acknowledgement, timeout)) **then**
4: update contract by removing the clauses linked to the requirements
5: **else**
6: update status of the BA to *unreachable* in the SCT
7: throw exception

Behavior 4 SR-CNPparticipantBA

Input: (i) *call-for-proposals* from CAs or other BAs to resolve R
Output: (i) Instantiation of **SR-CNPinitiatorBA** behaviors or (ii) *refuse* message
1: **if** (BlockReceive(*call-for-proposals*(R))) **then**
2: reqsSPAs ← get requirements that can be fulfilled by feasible SPAs from R
3: reqsBAs ← get requirements that can be fulfilled by feasible BAs from R
4: **if** (all the requirements can be fulfilled by either BAs or SPAs recorded in SCTs) **then**
5: BA sends Proposal to *initiator*
6: **if** (BlockReceive(reply, timeout)) **then**
7: **if** (proposal is accepted) **then**
8: create contract between the BA and either CAs or other BAs
9: SPAcontractors ← getFeasibleAgts(SPAs, reqsSPAs)
10: contract SPAs by adopting **SR-CNPinitiatorBA**(SPAcontractors, reqsSPAs)
11: **if** (it is necessary to subcontract services to other BAs) **then**
12: BAcontractors ← getFeasibleAgts(BAs, reqsBAs)
13: contract BAs by adopting **SR-CNPinitiatorBA**(BAs, reqsBAs)
14: **else**
15: start over
16: **else**
17: start over
18: **else**
19: BA sends *refuse* message to *initiator*
20: start over

tifier should be provided as well as the requirements to be removed. The CA sends a request to reduce the virtualized service (line 2) to the contracted BA (line 1). Then, the CA waits for an *acknowledgement* message (line 3), and if it is received, the CA updates its contract (line 4), otherwise it updates the status of the BA in the SCT to *unreachable* and throws an exception (lines 6 & 7).

In persistent service compositions, Cloud resources are reserved for long periods of time, e.g., infrastructure-as-a-service scenarios. If expired contracts are detected, a notification is sent to the corresponding consumer. Then, to receive contracts' expiration notifications a *ContractChangeMonitor* behavior is integrated into CAs. When a notification derived from expired contracts is received, the associated contract is updated. An explicit definition of *ContractChangeMonitor* behavior is omitted due to the fact that its description is straightforward enough to be implemented.

3.2 Broker agent behaviors

BAs are endowed with four main behaviors: *SR-CNPparticipantBA*, *SR-CNPinitiatorBA*, *ResultHandlerBA*, *ServiceRevokerBA*, and *ContractChangeMonitor*.

The *SR-CNPparticipantBA* behavior (Behavior 4) is based on the participant role of the SR-CNP. Its main function is to handle consumers' service composition requests either from CAs or other BAs (as explained in Sect. 2.1). Service composition requests consist of a set of consumer requirements. When a BA receives a *call-for-proposals* to carry out a service composition (line 1), the BA determines whether the requirements can be resolved by its SPAs

Behavior 5 SR-CNPinitiatorBA

Input: (i) Contractors (BAs or SPAs) and (ii) set of requirements R
Output: (i) Instantiation of a **ResultHandlerBA** behavior or (ii) failure propagation
1: The BA marks the set of requirements R as handled by it.
2: BA sends *call-for-proposals* to resolve R requirements to m Contractors recorded in SCTs
3: **if** (BlockReceive(Proposals, timeout1)) **then**
4: evaluate proposals
5: BA sends *reject-proposal* to (nProposals-1) Contractors
6: BA sends *accept-proposal* to 1 Contractor
7: create a **ResultHandlerBA**(R) behavior to virtualize the service composition
8: create a **ServiceRevokerBA**(R) behavior to allow subtractive updates
9: **else**
10: remove contract and update status of the Contractors to *unreachable* in the SCT
11: BA sends *failure(R)* message to *requester*

recorded in the SCT (line 2) and/or whether some requirements should be submitted to other BAs (line 3) when subcontracting is required. If all the requirements can be resolved by the current set of agents recorded in the SCTs (line 4), a *proposal* is sent to the *initiator* (line 5), otherwise the BA sends a *refuse* message (line 19). If the *proposal* is accepted (line 7), the BA establishes a contract with the consumer and instantiates parallel *SR-CNPinitiatorBA* behaviors for delegating the requirements (lines 8–13). As indicated in lines 9–11, the delegation of requirements to SPAs has priority over the delegation to BAs, because subcontracting is assumed to be computationally expensive.

The *SR-CNPinitiatorBA* behavior (Behavior 5) is based on the initiator role of the SR-CNP. Its main function is to contract service providers (either SPAs or BAs). The inputs (contractors and requirements) are previously determined in the *SR-CNPparticipantBA* behavior that instantiated the behavior. As the first step of the *SR-CNPinitiatorBA* behavior, the requirements to be delegated, are marked by the BA (line 1) to prevent cyclic nested contracts, e.g., agent a contracting agent b, b contracting c, and c contracting a. In addition, the marking of requirements, results in the reduction of messages exchanged in some situations, as analyzed in the empirical evaluation of the testbed in Sect. 4.1, Observation 5. In difference to the initiator role of the SR-CNP, the *ResultHandlerBA* (line 7) and *ServiceRevokerBA* (line 8) behaviors are instantiated when a proposal is selected (line 4).

The main function of the *ResultHandlerBA* behavior (Behavior 6) is to virtualize the service composition. The input corresponds to results derived from resolving previously delegated requirements. If the requirement was successfully resolved (line 2), its corresponding contract is updated (line 3) to record the progress of the service composition. If the service composition is complete (line 4), the service is virtualized and delivered (line 5), otherwise the BA waits for the remaining unresolved requirements with the remaining associated *ResultHandlerBA* behaviors. If the fulfillment of some requirements fail (line 6), the agent that failed is marked as *failed* in its corresponding SCT (line 7). Then, the unresolved requirements are reassigned to remaining feasible providers (lines 8–16) instantiating *SR-CNPinitiatorBA* behaviors. If there is no feasible provider, the contract is cancelled, and the failure is propagated (lines 19 & 20).

The *ServiceRevokerBA* behavior (Behavior 7) is created whenever a service composition is started. Its function is to allow subtractive updates. To update requirements of a service composition, its corresponding contract identifier should be provided as well as the requirements to be removed. A service composition can be reduced or simply removed (line 7), if all its requirements are provided. The subtractive process starts when a request to remove requirements is received (line 1). Then, the BA sends an *acknowledgement* message to the requester. Given that BAs may use several providers (SPAs or other BAs) to complete a service composition, removal requests are sent to each associated provider (lines 4 & 5). Finally, if an *acknowledgement* message is received, the contract associated to the service composition is updated (line 7).

In persistent service compositions, Cloud resources are reserved for long periods of time, e.g., infrastructure-as-a-service scenarios. So, similarly to CAs, a *ContractChangeMonitor* behavior is integrated into BAs to receive contracts' expiration notifications from either other BAs or SPAs, if an expiration notification is received, the notification is propagated to the corresponding consumer either other BAs or CAs. An explicit definition of *ContractChangeMonitor* behavior is omitted due to the fact that its description is straightforward to be implemented.

3.3 Service provider agent behaviors

SPAs are endowed with eight main behaviors: *SR-CNPparticipantSPA*, *ReqAssignerSPA*, *IntermediarySPA*, *SR-CNPinitiatorSPA*, *ServiceRevokerSPA*, *ResultHandler SPA*, and *ContractExpirationMonitor*.

Behavior 6 ResultHandlerBA

Input: (i) Outputs of requirements
Output: (i) Result propagation or (ii) failure propagation

```
1:  if (BlockReceive(result, timeout2)) then
2:      if (it is an outcome of a fulfilled requirement) then
3:          fulfill the contract clause of the corresponding requirement's contract
4:          if (all the clauses of the contract are fulfilled) then
5:              deliver virtualized service to its requester
6:      else-if (it is a failure) then
7:          update status of the agent to failed in the SCT
8:          failedRequirements ← unpack the failed Requirements from result message
9:          reqsSPAs ← get requirements that can be fulfilled by SPAs from failedRequirements
10:         reqsBAs ← get requirements that can be fulfilled by BAs from failedRequirements
11:         if (all the requirements can be fulfilled by either BAs or SPAs recorded in SCTs) then
12:             SPAcontractors ← getFeasibleAgts(SPAs, reqsSPAs)
13:             contract SPAs by adopting SR-CNPinitiatorBA(SPAcontractors, reqsSPAs)
14:             if (it is necessary to subcontract services to other BAs ) then
15:                 BAcontractors ← getFeasibleAgts(BAs, reqsBAs)
16:                 contract BAs by adopting SR-CNPinitiatorBA(BAs, reqsBAs)
17:             start over
18:         else
19:             remove contract
20:             BA sends failure(R) message to requester
21: else
22:     remove contract and update status of the agent to failed in the SCT
23:     BA sends failure(R) message to requester
```

Behavior 7 ServiceRevokerBA

Input: (i) Consumer requirements to be removed R and (ii) contract ID
Output: (i) A reduced single virtualized service

```
1:  if (BlockReceive(request to remove R requirements)) then
2:      BA sends acknowledgement message to requester
3:      for (all requirements R_i to be removed) do
4:          provider ← get previously contracted provider based on contract ID and R_i
5:          BA sends request to remove requirement R_i to provider
6:          if (BlockReceive(Acknowledgement, timeout)) then
7:              update contract by removing requirement R_i
8:          else
9:              update status of provider to failed in the SCT
10:             throw exception
11:     start over
```

The *SR-CNPparticipantSPA* behavior (Behavior 8) is based on the participant role of the SR-CNP. The main difference with previous participant roles is that when a proposal is accepted (line 4), a *ReqAssignerSPA* behavior is instantiated (line 6) to delegate the acquired requirement to appropriate RAs.

The *ReqAssignerSPA* behavior (Behavior 9) assigns requirements to available and appropriate RAs. First, it is verified whether feasible (working) RAs exist to handle the given requirement (line 1). Then, if RAs are available, the requirement is delegated (lines 3 & 4) and the status of the RA is changed to busy in the SCT of RAs (line 5). In addition, a *ResultHandlerSPA* behavior is instantiated to catch the output (line 6). If there is no available RA, the behavior holds until an RA is available (line 8). This handles the use of resources that are concurrently accessed as well as allowing service providers with scarce resources handling heavy loads of service requests.

Behavior 8 SR-CNPparticipantSPA

Input: (i) *call-for-proposals* from BAs or other SPAs to resolve *r*
Output: (i) Instantiation of a **ReqAssignerSPA** behavior
1: **if** (BlockReceive(*call-for-proposals*(*r*))) **then**
2: SPA sends Proposal to *initiator*
3: **if** (BlockReceive(reply, timeout)) **then**
4: **if** (proposal is accepted) **then**
5: create contract between the SPA and either BAs or other SPAs
6: delegate requirement *r* to RAs by adopting a **ReqAssignerSPA**(*r*) behavior
7: create a **ServiceRevokerSPA**(*r*) behavior to allow subtractive updates
8: start over
9: **else**
10: start over

Behavior 9 ReqAssignerSPA

Input: (i) Requirement *r* to be resolved (ii) contract ID
Output: (i) Instantiation of a **ResultHandlerSPA** behavior or (ii) failure propagation
1: **if** (there are RAs that can fulfill requirement *r* in the SCT) **then**
2: **if** (there is an available RA that can handle *r*) **then**
3: RA ← get an available RA that can handle requirement *r*
4: SPA sends *request* to resolve requirement *r* to the RA
5: update status of the RA to *busy* in the SCT.
6: create a **ResultHandlerSPA**(r) to receive the result
7: **else**
8: BlockUntil(there is an available RA that can handle *r*)
9: start over
10: **else**
11: update contract by removing the clause linked to requirement *r*
12: SPA sends *failure(r)* message to *requester*

Behavior 10 IntermediarySPA

Input: (i) Request to resolve an internal/external requirement *r*
Output: (i) Instantiation of **ReqAssignerSPA** behavior or (ii) Instantiation of **SR-CNPinitiatorSPA** behavior
1: **if** (BlockReceive(request to delegate requirement *r*)) **then**
2: **if** (there are RAs that can fulfill requirement *r*) **then**
3: create a **ReqAssignerSPA**(*r*) behavior to delegate the requirement internally
4: **else-if** (there are SPAs recorded in the SCT that can fulfill requirement *r*) **then**
5: create contract
6: SPAcontractors ← getFeasibleAgts(SPAs, *r*)
7: contract SPAs by adopting a **SR-CNPinitiatorSPA**(SPAcontractors, *r*) behavior
8: **else**
9: SPA sends *failure(r)* message to *requester* (RA)
10: start over

The *IntermediarySPA* behavior (Behavior 10) handles requests to resolve internal and external requirements from RAs. A requirement is internal when there is a sibling RA (an RA belonging to the same SPA) that can resolve the requirement (lines 2 & 3). A requirement is external when no sibling RA can resolve the requirement, and another SPA should be contracted (lines 4–7). This is to favor encapsulation and abstraction as explained in Sect. 2.1. The *SR-CNPinitiatorSPA* behavior (line 7) is similar to Behavior 5 of BAs, hence its description is omitted.

The coordination of RAs in such a manner allows simpler RA definitions by indicating what requirements are needed,

Behavior 11 ServiceRevokerSPA

Input: (i) Requirements to be removed *R* (ii) contract ID
Output: (i) Cancellation of services
1: **if** (BlockReceive(request to remove *R* requirements)) **then**
2: SPA sends *acknowledgement* message to *requester*
3: **for** (all requirements R_i to be removed) **do**
4: RA ← get RA that is fulfilling R_i based on the contract ID
5: SPA sends *request* to remove requirement R_i to the RA
6: **if** (BlockReceive(Acknowledgement, timeout)) **then**
7: update contract by removing the clause linked to requirement R_i
8: update status of the RA to *available* in the SCT.
9: **else**
10: update status of the RA to *failed* in the SCT
11: throw exception
12: start over

Behavior 12 ResultHandlerSPA

Input: (i) Outputs of requirements
Output: (i) Result propagation or (ii) failure propagation
1: **if** (BlockReceive(result, timeout2)) **then**
2: **if** (it is an outcome of a fulfilled requirement) **then**
3: fulfill the contract clause of the corresponding requirement's contract
4: SPA sends *inform*(result) message to its requester (BA/SPA/RA)
5: **if** (the fulfiller is an RA) and (it is not a persistent service composition) **then**
6: update status of the RA to *available* in the SCT.
7: **else-if** (it is a failure) **then**
8: update status of either the SPA or RA to *failed* in its corresponding SCT.
9: *r* ← unpack the failed requirement from *result* message
10: **if** (there is an RA that can fulfill *r*) **then**
11: delegate the requirement by adopting a **ReqAssignerSPA**(*r*) behavior
12: start over
13: **else-if** (there is a SPA recorded in the SCT that can fulfill *r*) **then**
14: SPAcontractors ← getFeasibleAgts(SPAs, *r*)
15: contract SPAs by adopting a **SR-CNPinitiatorSPA**(SPAcontractors, *r*) behavior
16: **else**
17: update contract by removing the clause linked to requirement *r*
18: SPA sends *failure(r)* message to *requester*
19: **else**
20: update contract by removing the clause linked to requirement *r*
21: update status of the agent to *failed* in its corresponding SCT
22: SPA sends *failure(r)* message to *requester*

instead of how to get them (details of RA definitions are given in Sect. 3.4).

The *ServiceRevokerSPA* behavior (Behavior 11) is similar to Behavior 7 of BAs. Its function is to carry out subtractive updates or cancel provided services. In addition, when a service is cancelled by its consumer, the RAs that were assigned to the service composition are released by updating their statuses in the SCT (line 8).

The *ResultHandlerSPA* behavior (Behavior 12) receives results from both RAs and contracted SPAs (line 1). As requirements are being resolved, the contract clauses are fulfilled (line 3). If the result was provided by an internal RA, and the service composition is not persistent, the RA is released (lines 5 & 6). Determining whether a requirement correspond to persistent compositions is based on its parameters stated in the contract clauses. If the fulfillment of a requirement fails (line 7), the agent that failed (either RA

Behavior 13 MainStructureRA

```
Input: (i) Request to resolve requirement r from its SPA
Output: (i) Inform results to SPA
1:    try
2:        switch (step)
3:            case (0): if (BlockReceive(request to resolve requirement r)) then
4:                        ... // resolving requirement
5:                        step++
6:                        break
7:            case (1): ... // resolving requirement
8:                        step++
9:                        break
10:           ...
11:           case (n): RA sends inform(output) message to SPA
12:                     step ← 0
13:                     break
14:   catch(exception)
15:       RA sends failure(r) message to SPA
16:   start over
```

Behavior 14 RequesterRA

```
Input: (i) Requirement r to be delegated to SPA
Output: (i) Reception of resolved requirement r
1:    RA sends request to delegate requirement r to SPA
2:    if (BlockReceive(result, timeout)) then
3:        if (result is received) then
4:            continue with the workflow specification using r.output
5:        else-if (there is a failure) then
6:            throw exception
7:    else
8:        throw exception
```

or SPA) is marked as *failed* in the corresponding SCT (line 8), and the unresolved requirement is reassigned to the next available RAs (lines 10 & 11), or if it is not possible, assigned to other SPA (lines 13–15). Finally, if no feasible agent is found, the contract is cancelled and the failure is propagated (lines 17 & 18).

In persistent service compositions, Cloud resources are reserved for long periods of time, e.g., infrastructure-as-a-service scenarios. To manage the contracts and Cloud resources associated to persistent compositions, the *ContractExpirationMonitor* behavior is provided. This behavior should be executed on a daily basis to detect expired contracts. If expired contracts are detected, a notification is sent to the corresponding consumers and the RAs involved in the contract are released. An explicit definition of *ContractExpirationMonitor* behavior is omitted due to the fact that its description is straightforward to be implemented.

3.4 Resource agent behaviors

The behaviors of RAs are pattern behaviors given that resources agents orchestrate web services, and the behaviors depend on the web service workflow. Two customizable patterns are given: *MainStructureRA* and *RequesterRA*.

The *MainStructureRA* behavior (Behavior 13) is mapped to a general-purpose agent behavior. The main structure of control is provided by a switch selector (lines 2–13), where the first instruction is the reception of a request to resolve a requirement from a SPA (line 3). In complex web services (web services that require exchanging more than one message to provide their output), ad-hoc agent behaviors should be created or added (lines 7–10), based on the web service workflow. In the case of atomic web services (one-time web services that produce outputs in one step), only the first and last switch cases (lines 3 & 11) are necessary to receive SPAs' requests (line 3), resolve the requirement (lines 4–

10), and inform the result (line 11). In case of an exception, a *failure* message is sent (line 15) to RA's SPA.

The *RequesterRA* behavior (Behavior 14) is defined for complex web services that need to interact either with sibling RAs or with SPAs for requesting external requirements (as explained in Sect. 3.3). This pattern behavior should be included into the *MainStructureRA* behavior as needed. In addition, to prevent hard wired agent interaction among RAs and SPAs, RAs delegate requirements to their SPAs (line 1), which is in charge of resolving the requirement (see *IntermediarySPA* behavior). Then, RAs wait for the result (line 2), and once received, the ad-hoc workflow continues (line 4). In case of exceptions (lines 5–8), these are caught by the *MainStructureRA* behavior.

In the case of RAs that control resources to be used in persistent service compositions, the *ReleaserRA* behavior is used. Its function is to release the Cloud resource, whenever its corresponding service contract is expired or it is cancelled by SPAs. The behavior consists of: (i) receiving SPAs' requests, (ii) sending an *acknowledgement* message, and (iii) resetting the *MainStructureRA* behavior to prepare the RA for new service requests. An explicit definition of *ReleaserRA* behavior is omitted due to the fact that its description is straightforward to be implemented.

3.5 Complexity of the agent behaviors

To measure the complexity of agent behaviors, two performance measures are considered: number of messages exchanged and time complexity of algorithms.

(*1*) *Number of messages*. The number of messages sent by agents relies on the level of connectivity of agents' SCTs, i.e., number of connections (e.g., service providers) recorded in SCTs.

For instance, the number of messages sent by a BA is determined by the nested instantiations of the *SR-CNPinitiatorBA* behavior (see lines 10 and 13 of the *SR-CNPparticipantBA*, and lines 13 and 16 of the *ResultHandlerBA* behaviors). In the worst case scenario, when a CA submits a requirement to a BA. The *SR-CNPparticipantBA behavior* creates p (No. of requirements submitted by the CA) instances of the *SR-CNPinitiatorBA*. For each instance, the BA with q feasible SPAs, sends q *call-for-proposals* messages (assuming that the BA have enough SPAs recorded in its SCTs to handle all possible requirements). After receiving the proposals, the BA sends q responses (1 *accept-proposal* message and $q-1$ *reject-proposal* messages). If the contracted SPAs fail, the *failure* is propagated to the BA. Then, the BA sends $q-1$ *call-for-proposals* messages to the remaining feasible SPAs, and $q-1$ responses, and so on. Thus, in the worst case scenario the BA sends:

$$\begin{aligned} &p\big(2q+2(q-1)+2(q-2)+\cdots+2(2)+2(1)\big)\\ &\quad=2p\big(q+(q-1)+(q-2)+\cdots+(2)+(1)\big)\\ &\quad=2p\big(q(q+1)/2\big)=p^{*}q(q+1)\ \text{messages} \end{aligned}$$

If after contacting all the SPAs, the p requirements are not resolved, the BA may subcontract services to one BA among r BAs by again adopting the *SR-CNPinitiatorBA* behavior. In a similar case to SPAs, if all the subcontracted BAs fail, the BA sends: $p^{*}r(r+1)$ messages.

The remaining broker behaviors send a relatively insignificant number of messages. Thus, the number of messages sent by BAs in the worst case scenario is bounded by: $p^{*}q(q+1)+p^{*}r(r+1)$ messages. When p, q, and r, tend to infinity, the number of messages exchanged in the worst case scenario is bounded by $2n^3$ messages

In the case of SPAs, in the worst case scenario, when a BA submits a requirement to a SPA. The SPA sends a *request* message to a feasible RA (see line 4 of *ReqAssignerSPA* behavior), if the RA fails to resolve the requirement, the SPA sends a message to another feasible RA. If all the RAs fail, the SPA will have sent i *request* messages to i feasible RAs. Now in a similar case to BAs, SPAs adopt the *SR-CNPinitiatorSPA* behavior to delegate the requirement to q agents (other SPAs), and if all of them fail, the SPA will have sent $q(q+1)$ messages. Thus, in the worst case scenario a SPA, which handles r requirements, sends: $r(i+q(q+1))$ messages. When r, i, and q, tend to infinity, the number of messages exchanged in the worst case scenario is bounded by n^2+2n^3 messages.

CAs send $2r$ messages to ask for a service composition, where r is the number of BAs recorded in the SCT. RAs may send 1 out of 4 types of messages: (i) to inform the result of a resolved requirement, (ii) to request the delegation of a requirement, (iii) to acknowledge the cancellation of a service, and (iv) to indicate that a failure occurred.

(*2*) *Time complexity*. All the functions involved in the agent behaviors (e.g., getFeasibleAgts()) take $O(n)$ time to execute. Agent Behaviors 6, 7, 11, and *ContractExpirationMonitor* have quadratic time complexities $O(n^2)$. The remaining behaviors have linear time complexities $O(n)$. This can be determined by simple inspection of the algorithms. The complexity of Behavior 13 (*MainStructureRA*) depends on the workflow associated to the web service that is orchestrated. However, in the case of atomic web services, the complexity is $O(n)$.

4 Evaluation and empirical results

Three groups of experiments were conducted using the agent-based testbed defined in Sects. 2 and 3. The testbed was implemented using the java agent development framework [7] (JADE).

Table 2 Input data source for Experiment 4.1

Input data	Possible values			
Service composition request rate (Requests per second)	*Low*		*High*	
	5		100	
Level of knowledge	1 % to 33 %	34 % to 66 %	67 % to 100 %	100 %
Degree of connectivity	*weak*	*moderate*	*strong*	*full*
Yellow page service provider	service capability tables			central directory
Probability of failure of RAs	{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}			
Cloud resource types	$\{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$			
No. of requirements per service composition request	1 to 20			
SR-CNP's timeouts	*timeout1*		*timeout2*	
	2 s		Undefined	

The experiments were conducted on a computer with the following specifications: Intel Core 2 Duo E8500 3.16 GHz, 4 GB RAM, with a Windows Vista Enterprise (32 bits) operating system, service pack 1.

4.1 Evaluating self-organization in horizontal and vertical service composition

(*a*) *Objective*. The series of experiments were designed: (i) To evaluate the self-organization capabilities of agents in both vertical and horizontal service composition scenarios. (ii) To compare two agent-based approaches: one using SCTs, where agents do not necessarily have complete information about other agents' service capabilities, and the other using a central directory where agents have complete knowledge about other agents' service capabilities.

(*b*) *Experimental settings*. As presented in Table 2, there are six input parameters in the testbed: (i) the service composition request rate, (ii) the level of knowledge of agents regarding information about other agents' service capabilities, (iii) the probability of failure of RAs, (iv) the number of resource types, (v) the number of requirements per service composition request, and (vi) the timeouts involved in the SR-CNP.

The service composition request rate is the number of service composition requests per second. In some commercial web services [4], the request rate is set to 1 request per second. Therefore, the agents must handle request rates higher than the limit of web services to avoid any delay. To ensure efficiency, two different request rates were used: (i) A high request rate (100 requests per second) was selected to explore the efficiency of the testbed in extreme situations, and (ii) a low request rate (5 requests per second) was selected to evaluate the testbed in possible real world settings.

Agents adopting weakly, moderately, and strongly connected SCTs have knowledge of the service capabilities of other agents, ranging from 1 % to 33 %, 34 % to 66 %, and 67 % to 100 %, respectively. The exact level of knowledge within each range, and the agents included in SCTs were determined randomly. Agents using a central directory that contains addresses and service capabilities of all agents in the system are considered to be fully connected with a level of knowledge of 100 %. The central directory is handled by a system agent, to which agents send messages to consult the directory.

To evaluate the self-organization capabilities of agents, RAs were designed to fail with a failure probability in the range of 0.0 to 1.0 (in 0.1 increments). When RAs (as the end-point of the agent-based testbed) fail, they induce the highest need for self-organization to the system. RAs contact their SPAs that may contact other sibling RAs, SPAs or sending result messages to BAs, which may attempt to subcontract services to other BAs, and so on. In addition, SPAs and BAs were designed to provide random quotations to keep a uniform exploration of SCTs.

Cloud resource types were randomly selected from $\{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$, which can be mapped to current available types of general-purpose instances of Amazon EC2 [2]. This creates heterogeneous SPAs with different types and level of Cloud resources. In addition, the web services contained in the RAs are assumed to be atomic.

The number of requirements per service composition request was randomly set from 1 to 20 requirements. In addition, the type for each requirement was randomly assigned based on the available resource types. In doing so, consumer composition requests were randomly generated in both horizontal and vertical scenarios. The maximum level of requirements was defined based on the current consumers' limit of on-demand/reserved instances of Amazon EC2 [1]. An example of a Cloud service composition request is as follows $\{(2, r_1), (3, r_4), (1, r_7), (4, r_8)\}$ where 2, 3, 1, and 4 resources

Table 3 Performance measures for Experiment 4.1

Percentage of successful service compositions	N_{SUC}/N_{ATT}
Average service composition time for successful service compositions	$\sum(T_{SUC})/N_{SUC}$
Average number of messages	$\sum(M_{ATT})/N_{ATT}$
N_{SUC}: Number of successful service compositions	
N_{ATT}: Number of attempted service compositions	
T_{SUC}: Service composition time for successful service compositions	
M_{ATT}: Number of messages exchanged for attempted service compositions	

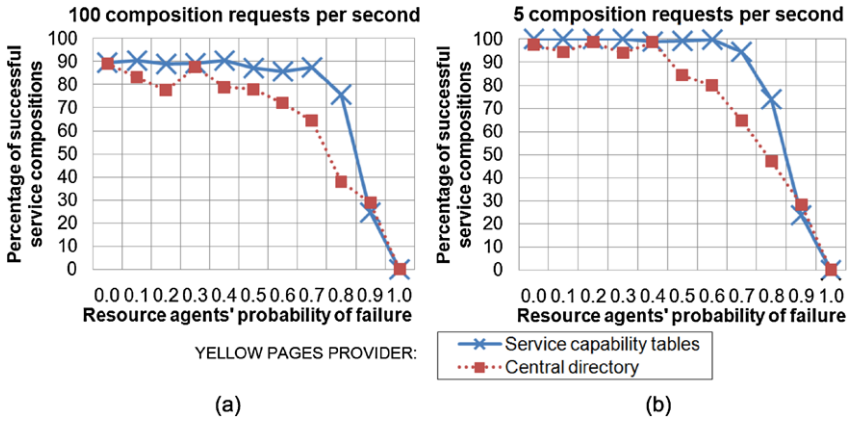


Fig. 3 Overall percentages of successful service compositions

of types r_1, r_4, r_7, and r_8 are requested, respectively, and resource types r_1, r_4, r_7, and r_8 may represent Amazon EC2 instances endowed with specific-purpose software to execute bioinformatics applications, data mining applications, image manipulation applications, etc.

There are two relevant agents' timeouts involved in the agent behaviors (see *SR-CNPinitiatorCA behavior*). The first timeout *timeout1* (line 3) corresponds to the time that agents wait for receiving proposals to carry out the composition. The second timeout *timeout2* (line 8) corresponds to the time that agents wait for receiving virtualized service compositions. These timeouts are also used by BAs and SPAs. The values of the timeouts have influence in the response time of the testbed given that short timeouts may produce quicker compositions, however agents may not receive responses to *call-for-proposals* to carry out service compositions.

The current Cloud service composition approach may be oriented to human users (although not exclusively), and the maximum acceptable response time to any user-oriented processing task (according to [47]) is 10.0 s. An experimental tuning that was carried out showed that the majority of the Cloud service compositions completed within 10.0 s (see Fig. 5). This resulted in a *timeout1* set to 2 s. In the case of *timeout2*, it was left undefined, i.e., a considerable large value was assigned to provoke that agents waited until a virtualized service composition was delivered or a failure message was received. In doing so, it was possible to differentiate between service compositions that took much time to complete and those that were not achieved because of RAs' failures.

The agents involved in the simulations were: 25 CAs, 25 BAs, 25 SPAs, and 3000 RAs. In addition, RAs were randomly allocated among the 25 SPAs to simulate Cloud providers with different levels of resources.

For each configuration of the agent-based testbed, 10 experiment runs were carried out. Each run consisted of 25 service compositions, which were conducted with 11 (from 0.0 to 1.0) probabilities of failure of RAs, then for each level of knowledge (4 levels), and finally, for 2 request rates. This resulted in an overall of 22,000 Cloud service compositions.

(*c*) *Performance measures*. The performance measures are: (i) Percentage of successful service compositions, (ii) average service composition time for successful service compositions, and (iii) average number of messages exchanged for successful service compositions. See Table 3 for details.

(*d*) *Results*. Empirical results are shown in Figs. 3, 4, 5, 6. From these results, five observations are drawn.

Observation 1. In general, agents adopting weakly, moderately, and strongly connected SCTs achieved higher success rates in service compositions than agents using a central directory for both high and low request rates.

Analysis: Figs. 3(a) and 3(b) show that agents adopting SCTs achieved higher success rates in service composition than agents using a central directory. This was because the central directory introduced a bottleneck in the system given that agents requesting for providers that could resolve

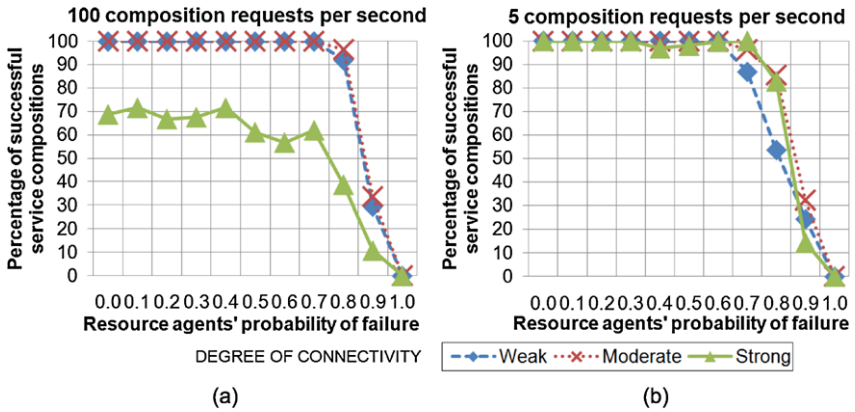


Fig. 4 Successful service composition rates using SCTs

a given requirement overloaded the agent that controls the directory (a *system agent*). This slowed down the response time of the *system agent* and caused a delay that prevented BAs and SPAs from responding to *call-for-proposals* on time. For example, when a BA receives a *call-for-proposals* to carry out a service composition, the BA consults the system agent of the central directory to check whether appropriate SPAs exist before preparing and sending a proposal (see line 4 of *SR-CNPparticipantBA* behavior). Then, if the delay in accessing the directory is longer than the timeout for receiving proposals (*timeout1*) of the initiator agent of the SR-CNP, the initiator agent opts out from the SR-CNP because no proposals were received. This delay was longer for higher request rates, showing the worst performance (Fig. 3a) in comparison to the lower request rates (Fig. 3b).

It can be seen from Figs. 3(a) and 3(b) that, agents adopting weakly, moderately, and strongly connected SCTs generally achieved the best performance. This was because SCTs prevent system bottlenecks. Agents using SCTs, although incomplete tables, were generally able to respond to *call-for-proposals* on time. Even though in some cases, agents rejected *call-for-proposals* (see line 19 of *SR-CNPparticipantBA* behavior) due to the lack of appropriate agents recorded in SCTs to carry out the service compositions, agents that responded to *call-for-proposals* were capable of carrying out the service composition through collaborating with their contacts recorded in SCTs.

These results show that through self-organization and interaction, agents adopting SCTs outperformed agents using a central directory in achieving successful service compositions despite dealing high probabilities of failure and incomplete information.

Observation 2. With high service request rates, agents adopting weakly and moderately connected SCTs achieved significantly higher success rates in service composition than agents adopting strongly connected SCTs. However, for low service request rates, agents adopting weakly, moderately, and strongly connected SCTs generally achieved almost similar success rates in service composition except for high probabilities of failure. With low service request rates, when the probability of failure is higher than 0.7, agents adopting moderately and strongly connected SCTs generally achieved significantly higher success rates in service composition than agents adopting weakly connected SCTs.

Analysis: As it can be seen in Fig. 4(a), agents who adopted weakly and moderately connected SCTs considerably outperformed agents who adopted strongly connected SCTs with high service request rates. This was because agents adopting strongly connected SCTs had more connections, and hence, they contacted more contractors (when executing the SR-CNP) before contracting one. This triggered a message flooding in the system that saturated agents, and thus, agents were unable to send proposals before reaching SR-CNP's timeouts (as it was explained in observation 1 of Experiment 4.1). This caused agents to opt out from the SR-CNP, propagating service composition failures that affected the performance (Fig. 4(a)).

However, as it can be observed in Fig. 4(b), in general, agents adopting weakly, moderately, and strongly connected SCTs achieved similar success rates for low service request rates. With fewer composition requests, all the agents (including agents adopting strongly connected SCTs) were capable of promptly handling the load of messages derived from adopting the SR-CNP. Nonetheless, for high probabilities of failure (Fig. 4(b)), agents adopting moderately and strongly connected SCTs generally outperformed agents adopting weakly connected SCTs. This was due to the high connectivity degree that allowed agents to self-organize with more connections, and thus, improving the service composition success rate. Nonetheless, Fig. 4(b) shows that agents adopting strongly connected SCTs with a probability of failure of 0.9 achieved the worst performance. This was because they had more connections, and thus sent more messages. This triggered a message flooding that saturated agents, similar to the presented with high service request rates (Fig. 4(a)).

These results indicate that there is a tradeoff between the (i) number of contractors taken into account to execute

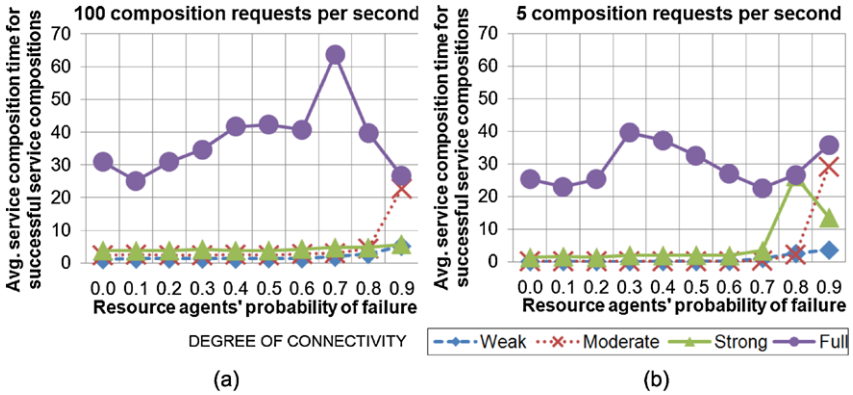


Fig. 5 Average service composition time for successful compositions

the SR-CNP and the service request rates, and (ii) the service composition success rate. For high service request rates, agents should avoid considering many contractors (e.g., not higher than 16, as agents adopting weakly and moderately connected SCTs) for carrying out a service composition. For low service request rates, agents should consider as many contractors as possible to obtain the better (cheapest) services, and preserving high service composition success rates.

Observation 3. On the average, agents adopting weakly, moderately, and strongly connected SCTs took shorter time than agents adopting a central directory for successfully composing services.

Analysis: it can be noted from Figs. 5(a) and 5(b) that agents adopting SCTs took substantially shorter time than agents using a central directory. As previously highlighted in the analysis of observation 1 of Experiment 4.1, accessing the central directory consumes time. Thus, agents receiving highly heterogeneous (random) service composition requests must access the central directory continuously, incrementing the time to complete service compositions. On the other hand, agents adopting SCTs divided service composition requests into several parts, which were delegated to several agents (see lines 2–3 of *SR-CNPparticipantBA* behavior). In doing so, agents with only local knowledge distributed the work, preventing bottlenecks that slow down the service composition process.

These results show that through collaboration, agents adopting SCTs distribute the service composition load over all agents, achieving service compositions in shorter times. On the contrary, agents using a central directory centralized the control under a single component, and this caused some deterioration in performance.

Observation 4. Whereas agents adopting weakly, moderately, and strongly connected SCTs achieved generally constant average time for composing services for different probabilities of failure, the average time for agents using a central directory to compose services largely fluctuated for different probabilities of failure.

Analysis: It can be observed from Figs. 5(a) and 5(b) that, in general, agents adopting SCTs composed services in a constant average time, even when reacting to different failure rates. This was due to the local management that agents followed to handle failures, i.e., when an agent, either BA, SPA or RA, failed to resolve a requirement, the original requester, either BA or SPA, contacted its next feasible agent from the SCTs. BAs could have contacted SPAs or other BAs, and SPAs could have delegated the requirement to other RA or contacted other SPAs. This absorbed the impact caused by failures. In addition, agents adopting SCTs distribute the loads for achieving service compositions (this was pointed out in the analysis of observation 3 of Experiment 4.1).

From Figs. 5(a) and 5(b) it can be noted that the average time to complete service compositions by agents using a central directory were largely different for different failure rates. This was because the central directory was saturated by agents requesting for addresses in a non-deterministic pace, i.e., agents accessed the directory with different rates based on (i) the time when their service composition proposals were accepted, and then, they needed addresses of SPAs or BAs, or (ii) when agents needed to subcontract the service from other agents to handle unresolved requirements derived from RAs' failures. The different overload rates caused the system agent in charge of the central directory to be unable to respond in a constant rate. This caused the agents, either SPAs or BAs, to complete their tasks in largely different amounts of time.

These results indicate that, in general, constant and low average time to compose services can be achieved with agents adopting SCTs given that the work is distributed among all the components of the system.

Observation 5. With lower probabilities of failure (up to 0.6), agents with lesser knowledge of other agents' service capabilities exchanged fewer messages, and agents adopting weakly connected SCTs exchanged the fewest messages, while agents using a central directory exchanged the most messages. However, for higher probabilities of failure

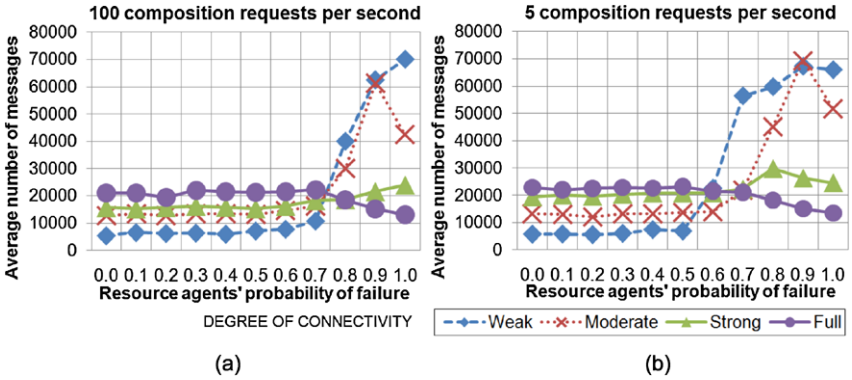


Fig. 6 Average number of messages

(above 0.6), agents with more knowledge of other agents' service capabilities, exchanged fewer messages, and agents using a central directory exchanged the fewest messages, while agents adopting weakly connected SCTs exchanged the most messages.

Analysis: Figs. 6(a) and 6(b) show the average number of messages exchanged. It is observed that, in most cases, for failure rates lower than 0.6, agents with lesser knowledge sent fewer messages. This was because agents with fewer connections contacted fewer service providers when executing the SR-CNP, and because the failures were handled locally, e.g., contacting sibling RAs.

On the other hand, Figs. 6(a) and 6(b) also show that, for failure rates higher than 0.6, the average number of messages exchanged was reduced as agents' knowledge of other agents' capabilities increased. This was because agent behaviors were designed to share information about failed agents (either BAs or SPAs), preventing unnecessary messages. This mechanism works as follows:

(i) When a set of requirements is submitted to an agent, the agent marks the requirements to indicate that the requirements were handled by it (see line 1 of *SR-CNPinitiatorBA* behavior).
(ii) When agents propagate failures, the failed requirements containing the list of previous handlers are sent to the original requester (see line 18 of *Result HandlerSPA* behavior).
(iii) Agents inspect the list of previous failed agents, and omit them for future interactions (see line 2 of *SR-CNPparticipantBA* behavior).

For instance, a BA with full knowledge of all SPAs' capabilities contacts all the available SPAs for composing a service, but subsequently, almost all the SPAs fail (common with high failure rates). Then, the failure is propagated back to the BA. When the BA attempts to subcontract the service to other BAs, they will not carry out the service composition because all their SPAs recorded in the SCTs already have failed. In contrast, for a BAs with lesser knowledge of other agents' capabilities, the sets of SPAs included in the SCTs of BAs tend to be mutually exclusive given that the SCTs were randomly created using a uniform distribution. Therefore, more messages within BAs' layer where needed to contact all SPAs.

Agents are less likely to send out unnecessary messages if they have more knowledge about other agents' capabilities. In addition, information exchange allows agents to adapt and react according to feedback provided by other agents regarding the current state of the Cloud-computing environment.

4.2 Evaluating self-organizing agent layers

(*a*) *Objective*. A series of experiments was carried out to study the performance of different configurations of the testbed with agents adopting SCTs by varying the number of agents in each layer.

(*b*) *Experimental settings*. As presented in Table 4, there are six input parameters in the testbed: (i) the number of CAs, BAs, and SPAs grouped in three categories: small, medium, and large, (ii) the service composition request rate, (iii) the level of knowledge of agents, (iv) the number of resource types, (v) the number of requirements per service composition request, and (vi) the timeouts involved in the SR-CNP.

The number of CAs, BAs, and SPAs was grouped in three size categories: small (S)—5 agents, medium (M)—15 agents, and large (L)—25 agents. This resulted in 3^3 testbed's configurations. Agents' SCTs were randomly defined, considering the whole agent universe, i.e., each agent was randomly connected, from 1 % to 100 % of the agent universe. The service composition request rate was set to 100, to evaluate the performance of the testbed in each configuration, considering extreme situations. The Cloud service composition requests were randomly generated following the constraints defined in Experiment 4.1. In addition, the timeouts of the SR-CNP: *timeout1* was set to 2 seconds and *timeout2* was left undefined, as defined in Experiment 4.1. For each configuration of the testbed, 10 experiment runs were carried out, for an overall of 270 service compositions.

(*c*) *Performance measures*. The performance measures are: (i) Percentage of successful service compositions, (ii) aver-

Table 4 Input data source for Experiment 4.2

Input data	Possible values			
No. of agents per layer	*small (S)*	*medium (M)*	*large (L)*	
	5	15	25	
27 Configurations	{SCAs, MCAs, LCAs} × {SBAs, MBAs, LBAs} × {SSPAs, MSPAs, LSPAs}			
Level of knowledge	From 1 % to 100 % of the agent universe			
Yellow page service provider	service capability tables			
Service composition request rate	*High*: 100 requests per second			
Cloud resource types	$\{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$			
No. of requirements per service composition request	1 to 20			
SR-CNP's timeouts	*timeout1*			*timeout2*
	2 s			undefined

Table 5 Performance measures for Experiment 4.2

Percentage of successful service compositions	N_{SUC}/N_{ATT}
Average service composition time for successful service compositions grouped by size category and agent type	$\sum(T_{SUC})/N_{SUC}$ grouped by {S, M, L} × {CA, BA, SPA} e.g., $\sum(T_{SUC})/N_{SUC}$ grouped by S and CA indicates "Average service composition time for successful service compositions when the number of CAs was small"
Average number of messages grouped by size category and agent type	$\sum(M_{ATT})/N_{ATT}$ grouped by {S, M, L} × {CA, BA, SPA} e.g., $\sum(M_{ATT})/N_{ATT}$ grouped by S and CA indicates "Average number of messages exchanged for attempted service compositions when the number of CAs was small"
N_{SUC}: Number of successful service compositions	
N_{ATT}: Number of attempted service compositions	
T_{SUC}: Service composition time for successful service composition	
S: Small number of agents	
M: Medium number of agents	
L: Large number of agents	
M_{ATT}: Number of messages exchanged for attempted service composition	

age time for successfully composing services, and (iii) average number of messages exchanged. In addition, the performance measures were grouped by agent type and size category. See Table 5 for details.

(*d*) *Results*. Empirical results are shown in Figs. 7 to 9. From these results, two observations are drawn.

Observation 1. Almost all the configurations achieved a 100 % success rate in service composition. Except for LCA-SBA-LSPA, 26 out of the 27 configurations achieved a 100 % success rate, and LCA-SBA-LSPA achieved a 95 % success rate.

Analysis: It can be observed from Fig. 7 that, almost all the testbed's configurations achieved a perfect success rate. This was because at least one path existed between the CAs' connections included in the SCTs and the SPAs that contained the necessary Cloud resources to carry out the service composition requests.

In addition, as it is presented in Fig. 7, the configuration: LCA-SBA-LSPA (large number of CAs, small number of BAs, large number of SPAs) achieved a 95 % success rate. This was because: (i) several service compositions could not be handled completely by a single BA, and thus, the composition was divided increasing the number of messages in the BAs' layer, and (ii) a large number of CAs and SPAs implied that BAs received more service composition requests and considered more providers in the SR-CNP, respectively. This overloaded the system in the BAs' layer, and in some occasions, BAs opted out from the SR-CNP propagating failures

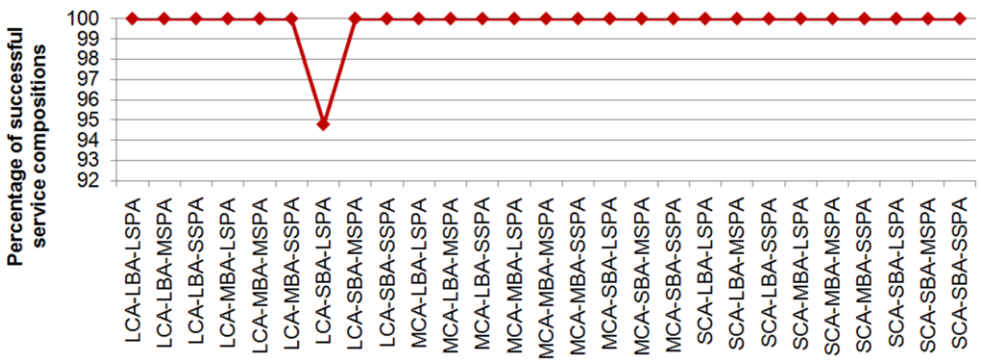


Fig. 7 Percentage of successful compositions of each testbed's configuration

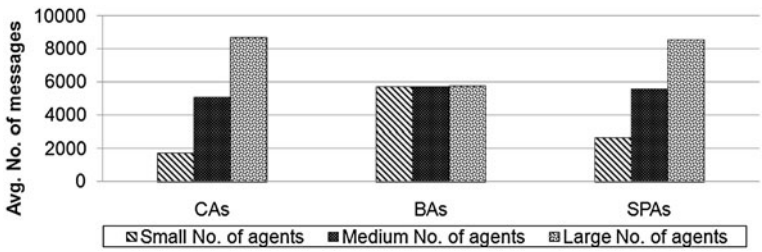


Fig. 8 Average No. of messages exchanged for service compositions group by agent type and size category

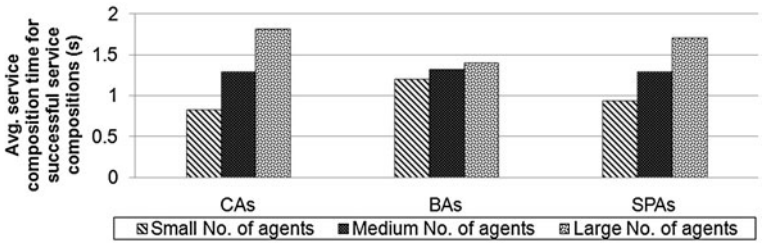


Fig. 9 Average service composition time for successful service compositions group by agent type and size category

(such as the system overloading analyzed in observation 1 of Experiment 4.1).

The proposed agent-based approach performs well for a wide variety of configurations with large, medium, and small numbers of agents in each layer. In addition, the results suggest that, (i) whereas reducing the number of providers contacted in the SR-CNP may leave out the cheapest service, the system overloading may be prevented, and that (ii) the BAs' layer should be endowed with at least a medium number of BAs to handle a large number of service composition requests, considering quotations from all the available service providers.

Observation 2. In general, the average number of messages exchanged for attempted service compositions was generally constant for all the size categories of BAs. The number of messages exchanged remains constant regardless of the number of BAs that have participated in the composition, and as a consequence, the average service composition time for successful compositions is generally similar.

Analysis: As it can be observed in Figs. 8 and 9, the average number of messages exchanged when the number of BAs was small, medium, and large, is almost constant (Fig. 8), and as a result, the average time for compositing services (Fig. 9) was almost similar. This was because BAs as intermediaries only mapped consumer requests into service provider's resources. Whereas in some cases, subcontracting services to other BAs was also necessary, the priority was given to contracting SPAs instead of other BAs, this left just few requirements for other BAs, reducing the number of additional messages derived from subcontracting.

These results indicate that, the number of BAs can be increased while preserving a similar performance of the system. Then, recalling observation 1 of Experiment 4.2, the 95 % success rate of configuration LCA-SBA-LSPA (Fig. 7) can be improved by augmenting the number of BAs, as occurred in the configurations LCA-MBA-LSPA and LCA-LBA-LSPA with a 100 % (Fig. 7) of successful compositions. This suggests that the number of BAs can be increased to deal with more service requests when the number of CAs increases.

4.3 Evaluating persistent service compositions' updates

(*a*) *Objective*. A series of experiments was carried out to evaluate the effectiveness and efficiency of the agent-based testbed in dealing with persistent horizontal and vertical ser-

Table 6 Input data source for Experiment 4.3

Input data	Possible values	
Service composition update requests	[remove(n requirements) $\gg$ add(n requirements)] where $1 \leq n \leq 10$ and $[p \gg q]$ means p is executed before p	
Level of knowledge	From 1 % to 100 % of the agent universe	
Yellow page service provider	service capability tables	
Cloud resource types	$\{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$	
No. of requirements per service composition request	20	
SR-CNP's timeouts	*timeout1*	*timeout2*
	2 s	Undefined

vice compositions by varying the number of consumer requirements to be removed/added to persistent service compositions.

(*b*) *Experimental settings*. The experiment runs were divided into ten groups, one group for each update level n, where $1 \leq n \leq 10$. Each group contained ten experiment runs. An experiment run consisted of: (i) creating the persistent service compositions that were composed of 20 randomly determined requirements (the maximum allowed), (ii) removing n requirements, and then (iii) adding n requirements.

The maximum update level was set to 10 given that service compositions were composed of 20 requirements, resulting in a maximum update of 50 %. The agents involved were: 1 CA, 25 BAs, 25 SPAs, and 3000 RAs. The SCTs were defined randomly, considering the whole agent universe (from 1 % to 100 %). Only one CA was considered to focus the experiment on the update process of persistent service compositions, isolating the results from the ones obtained in Experiments 4.1 and 4.2. The timeouts of the SR-CNP: *timeout1* was set to 2 s and *timeout2* was left undefined, as defined in Experiment 4.1. See Table 6 for details.

(*c*) *Performance measures*. In the case of incremental updates: (i) Percentage of successful incremental updates, (ii) ratio of No. of messages exchanged for resolving additional requirements to No. of messages exchanged for resolving initial requirements (additional-to-initial message ratio), (iii) ratio of service composition time for additional requirements to service composition time for initial requirements (additional-to-initial service composition time ratio). In the case of subtractive updates: (i) Percentage of successful subtractive updates, (ii) average number of messages exchanged for removed requirements, (iii) service composition time for subtractive updates. Additionally, a performance measure is the percentage of successful initial service compositions. See Table 7 for details.

(*d*) *Results*. Empirical results are shown in Figs. 10 and 11. From these results, three observations are drawn.

Observation 1. Agents in the testbed can respond to changing consumers' requirements by handling both incremental (addition of requirements) and subtractive (removal of requirements) updates, and achieved a 100 % success rate in creating and updating service compositions.

Analysis: Agents in the testbed achieved a 100 % success rate in creating persistent service compositions, a 100 % success rate in subtractive updates, and a 100 % success rate in incremental updates. All the initial service compositions and incremental updates were successful because at least one path existed between the CAs' connections included in the SCTs and the SPAs that contained the necessary Cloud resources to carry out the service composition requests (as pointed in the analysis of observation 1 of Experiment 4.2). In the simulations, all agents in all the service compositions were capable of finding a path to succeed in the Cloud service composition. In the case of subtractive updates, agent behaviors worked as expected, referring to their service contracts to contact the providers and cancel the services.

Equipped with partial knowledge of other agents' service capabilities as represented in SCTs, agents in the testbed autonomously respond to changing consumers' requirements (both incremental and subtractive), and achieved a 100 % success rate in updating and creating service compositions through self-organization and interaction using the SR-CNP.

Observation 2. Incremental updates were achieved in a fraction of the overall service composition time and the number of messages exchanged for the initial service composition.

Analysis. It can be observed from Fig. 10 that even though both additional-to-initial message ratio and additional-to-initial service composition time ratio increased as the number of additional requirements updated for the persistent service compositions increased, both additional-to-initial ratios were smaller than 1, i.e., all additional values were smaller than the initial values. For incremental updates, the number of additional messages exchanged did not increase significantly. Adding new requirements consisted of (i) sending a small number of messages from the CA to the

Table 7 Performance measures for Experiment 4.3

Percentage of successful incremental updates	N_{ADD}/A_{ADD}
Ratio of No. of messages exchanged for resolving additional requirements to No. of messages exchanged for resolving initial requirements (additional-to-initial message ratio)	M_{ADD}/M_{INI}
Ratio of service composition time for additional requirements to service composition time for initial requirements (additional-to-initial service composition time ratio)	T_{ADD}/T_{INI}
Percentage of successful subtractive updates	N_{REM}/A_{REM}
Average number of messages exchanged for removed requirements	M_{REM}/R_{REM}
Service composition time for subtractive updates	
Percentage of successful initial service compositions	N_{INI}/A_{INI}

N_{ADD}: Number of successful incremental updates
A_{ADD}: Number of attempted incremental updates
M_{ADD}: Number of messages exchanged for incremental updates
M_{INI}: Number of messages exchanged for initial service compositions
T_{ADD}: Service composition time for incremental updates
T_{INI}: Service composition time for initial service compositions
N_{REM}: Number of successful subtractive updates
A_{REM}: Number of attempted subtractive updates
M_{REM}: Number of messages exchanged for subtractive updates
R_{REM}: Number of requirements removed
N_{INI}: Number of successful initial service compositions
A_{INI}: Number of attempted initial service compositions

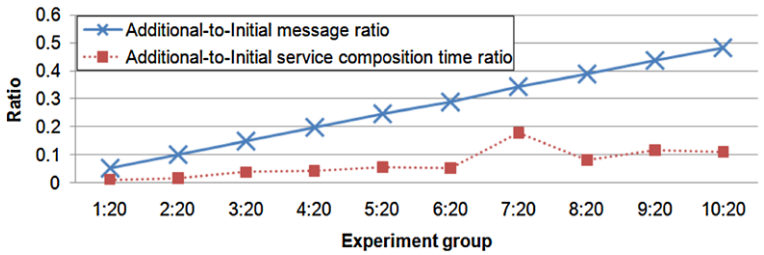


Fig. 10 Additional-to-initial message and service composition time ratios

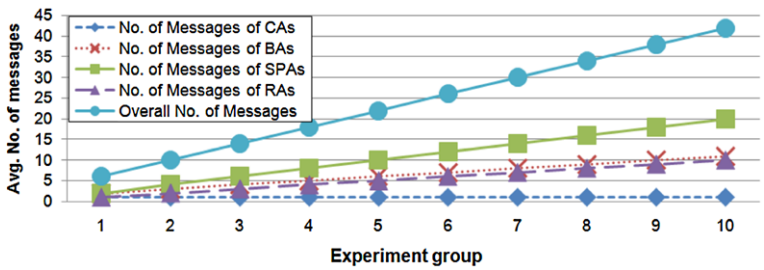


Fig. 11 Average number of messages exchanged in the removal process

contracted BA given the existence of a previous contract (see *ServiceAugmenterCA* behavior in Sect. 3.1), and (ii) the execution of the SR-CNP by the contracted BA to resolve the additional requirements. Subsequently, if some messages are exchanged for updating the requirements of persistent service compositions, some additional time is consumed that also has an increasing additional-to-initial service composition time ratio (Fig. 10). Nonetheless, the value in 20:7 of the service composition time ratio series (see Fig. 10) deviated from the general behavior. Some of the possible causes of this deviation are: (i) accumulated message latencies, and/or (ii) a delay in the completion of the SR-CNP caused by one or more SPAs, who took more time to send proposals to contracted BAs.

Incremental updates up to 50 % of additional requirements of persistent service compositions do not exceed the

effort (in terms of messages and time) of recomposing Cloud services from the beginning.

Whereas the analysis in observation 1 shows that agents in this work can deal with changing requirements and successfully compose services through self-organization, the results in Fig. 10 show that agents can carry out their tasks effectively and efficiently, because only a small fraction of composition time and number of messages exchanged are needed to process additional requirements.

Observation 3. The overall number of messages that agents exchanged to remove requirements from persistent service compositions was small. In addition, the time to process the subtractive updates was negligible.

Analysis: It can be seen from Fig. 11 that agents sent just a few messages for removed requirement. In fact, the overall number of messages (Fig. 11) involved in the subtractive update process is $4r + 2$, where r stands for No. of requirements, e.g., removing 7 requirements, involved 30 messages among all agents. The message complexity of the agent behaviors involved in subtractive updates is linear (see Behaviors 3 (*ServiceRevokerCA*), 7 (*ServiceRevokerBA*), 11 (*ServiceRevokerSPA*), and *ReleaserRA*).

In most occasions, the time to process subtractive updates was lower than 1 ms, because when some requirements were removed from a persistent service composition, CAs only sent 1 message to the contracted BA, and when the BA replied with an acknowledgement message, the composition is considered to be updated. Thus, from CAs' perspective, subtractive updates are carried out instantly. BAs were designed to reply first to CAs, and then carry out the removal with the involved providers to focus the system on improving consumer satisfaction.

Together with the analyses of observations 1 and 2 of Experiment 4.3, the results in Fig. 11 show that agent behaviors via self-organization and collaboration are effective and efficient for creating and updating persistent Cloud service compositions in distributed Cloud-computing environments with partial knowledge of service providers' capabilities and their location.

5 Related work

Since this work focuses on agent-based Cloud service composition, the related areas are: (i) agent-based service composition, and (ii) preliminary initiatives on Cloud service composition methods.

(*1*) *Agent-based service composition*. Automated web service composition supported by agents has been widely studied, from considering semantic aspects of web services [10, 28] through supporting service interaction [8, 39] and handling failures [29] to verifying and validating service compositions [35]. However, this section is only centered on automated web service composition approaches where agents show self-organization capabilities, reaction to environment's changes, and/or make use of cost-based service selection mechanisms, given their close relation with the present work.

In [22, 23], services are implemented by agents that register their capabilities with upper broker agents, which operate as directories and intermediaries for interrelated services, supporting agent communication. In turn, broker agents are registered with super broker agents. Agents organize themselves through brokers and connect with other agents to compose services. However, agents depend on agents of upper layers either broker or super broker agents, with no self-organization nor interaction within the same agent layer, centralizing the communication in brokers' layers. In addition, brokers are limited to route messages and no feedback, e.g., previous contacted brokers, is provided to upper layers. In addition, only simple agents' requests are passed, i.e., agents' requests that need several services' capabilities are not supported. In comparison with [22, 23], agents in this work are endowed with self-organizing skills that evolve the multiagent system based on agent interaction, which decentralize the service composition process. In addition, through collaboration agents decompose and distribute the complex task of Cloud service composition that may require multiple service capabilities.

In [12], Chafle *et al.* proposed partitioning service compositions. For each partition, an agent that monitors and records the progress of the web service composition is assigned. Monitoring agents maintain a global monitor agent by periodically sending information related to their partitions. When a monitoring agent detects an error inside its partition, the partition is stopped, and the error is replicated to the central monitor, this, in turn, stops all the running partitions and determines the last known correct state for every partition. Then, the central monitor sends indications to monitor agents to roll back their partitions and continue with the composition. Whereas agents react to failures and adjust service compositions, additional work is carried out in all the partitions to proceed consistently. In addition, a single failure stops the whole service composition process. Moreover, partitioning service compositions requires an initial phase that analyzes all the specifications of the involved services to determine appropriate partitions. In contrast to [12], the agent-based Cloud service composition mechanisms supported by the SR-CNP allow agents to organize themselves dynamically to adjust the service composition to deal with unexpected failures. Moreover, the additional effort required is distributed among all nearby agents where the failure occurred. Furthermore, no prior analysis of agents' capabilities nor full knowledge of their capabilities is required.

In [51], a multi-layered multiagent system consisting of broker agents, workflow planner agents, ontology agents,

and service agents is proposed to dynamically execute user workflows in web environments. Broker agents are in charge of coordinating the execution of tasks among service agents. Workflow planner agents indicate to broker agents the correct order of tasks to be executed. Ontology agents match user requirements with services, which are handed to broker agents to execute tasks on the set of possible services. Then, broker agents contract service agents by adopting the contract net protocol. Whereas a cost-based selection mechanism is integrated into the service composition method to execute workflows, the workflow planner agent assumes complete knowledge about existing services, and both the workflow planner and broker agents centralize the service composition process. In comparison to [51], agents in this work handle service compositions with incomplete knowledge about the services deployed and their capabilities by making use of SCTs. Both SCTs and SR-CNP endow agents with skills to compose Cloud services without planning ahead and in a fully distributed manner.

(*2*) *Cloud service composition*. Preliminary efforts that tackle Cloud service composition are included in [55] and [56].

In [55], a semantic-based matching method to compose Cloud services is proposed. Cloud services are endowed with semantic-enhanced input and output interfaces. Then, a function that determines the similarity level between services' interfaces of correlated web services is used to create a chain of services, which results in the service composition.

In [56], Cloud service composition is achieved in multiple Cloud-computing environments. From the service directories of each Cloud, a search tree is created to which artificial intelligence planning techniques are applied to obtain service compositions that involve the minimum number of Clouds.

Although [55] and [56] are focused on different aspects of Cloud service compositions, they share some strong assumptions, such as: complete knowledge of the Cloud-computing environments, and that all the web services are atomic. In addition, both [55] and [56] are centralized approaches. Moreover, inherent features of Cloud-computing environments are ignored, such as: (i) service fees associated to Cloud services, nor dynamic selection of services based on fees, (ii) service contract management, and (iii) updating Cloud service compositions. Furthermore, all Cloud service compositions are seen as one-time compositions. While in this work, (i) Cloud service composition is achieved with incomplete knowledge of the Cloud-computing environment, (ii) services can be atomic or complex, (iii) Cloud service selection based on market-driven fees is supported, (iv) agents create, update, and manage contracts derived from Cloud service compositions, and (v) not only one-time service composition is achieved but vertical, horizontal, and persistent Cloud service compositions are supported.

Finally, it is acknowledged that, this paper is a significantly and considerably extended version of the preliminary works reported in [20] and [21].

(1) Gutierrez and Sim [20] present initial Petri net agent models of Cloud Participants and a Petri net-based methodology to design workflow definitions for RAs. In addition, [20] makes use of Smith's contract net protocol [46] as a service selection mechanism. However, (i) the agent models do not consider self-organization capabilities and agent collaboration is limited to that of the contract net protocol, (ii) the testbed considers a central directory to locate Cloud services, assuming complete knowledge, and (iii) no collaboration among SPAs is implemented.
(2) Gutierrez and Sim [21] introduce self-organization capabilities into agent behaviors and the use of SCTs to replace the central directory of [20]. In addition, [21], only presents a very small set of results regarding self-organizing service compositions, where agents were included in fixed scenarios with three different levels of connectivity and with relaxed constraints. In addition, agent behaviors were designed to run in parallel and concurrently, but within the context of one service composition, i.e., a BA can run several behaviors at the same time, but just one composition can be executed at a time.

This present work significantly and considerably enhances [20] and [21] as follows: (i) BA collaboration is enhanced by dividing service composition requests into two main sets, a set that can be carried out by the current BA, and the other set that is delegated to other BAs by means of subcontracting (Sect. 3.2). (ii) Many new agent behaviors are included in CAs (Sect. 3.1), BAs (Sect. 3.2), SPAs (Sect. 3.3), and RAs (Sect. 3.4) to handle persistent service compositions as well as incremental and subtractive updates. (iii) The agent-based testbed includes the management of service contracts that support concurrent and parallel management of n Cloud service compositions. (iv) The results with respect to self-organizing service compositions are augmented and generalized by conducting experiments with stringent constraints (Sect. 4.1), e.g., randomly created SCTs, high service request rates, randomly generated service composition requests, etc. (v) In addition, new experiments for evaluating the performance of the testbed's self-organizing agent layers are conducted (Sect. 4.2). (vi) Moreover, an empirical comparison (Sect. 4.1) of self-organizing agents adopting SCTs versus self-organizing agents using a central directory is included. (vii) Furthermore, new results derived from evaluating the management of persistent service compositions are included (Sect. 4.3).

6 Conclusion and future work

The novelty and significance of this research is that, to the best of the authors' knowledge, it is the earliest effort in providing an agent-based approach for dealing with one-time, persistent, vertical, and horizontal Cloud service compositions as well as providing mechanisms to efficiently update persistent service compositions.

In this research effort, the new challenges that Cloud-computing environments in single and multiple Cloud schemes pose to automated and dynamic service composition were highlighted (Sect. 1). Self-organizing agent behaviors capable of reflecting self-interested characteristics of distributed and parallel executing Cloud participants were included (Sects. 2 and 3). A combination of two agent-based distributed problem solving techniques: SCTs (Sect. 2.1) and the SR-CNP (Sect. 2.2), was devised and integrated into agent behaviors to cope with (i) service selection based on dynamic services fees, and (ii) incomplete knowledge about the existence and location of service providers and the Cloud resources they offer. An agent-based Cloud service composition testbed (Sects. 2 and 3) was implemented to support one-time, persistent, horizontal, and vertical Cloud service compositions. Using self-organizing agents as building blocks, mechanisms to update and create service compositions based on constantly changing consumers' needs were designed. Finally, a series of experiments (Sect. 4) were conducted: (i) to evaluate the self-organization capabilities of agents, (ii) to compare agents adopting SCTs with incomplete knowledge versus agents using a central directory with complete knowledge, (iii) to study the performance of the agent-based Cloud service composition testbed, and (iv) to evaluate the effectiveness and efficiency of updating persistent service compositions. The empirical results (Sect. 4) show that via agent collaboration and self-organization, Cloud service compositions can be efficiently achieved and evolved based on constantly changing consumers' requirements, even in Cloud-computing environments where services fees vary based on a supply-and-demand basis, and where no complete information about distributed Cloud participants is available.

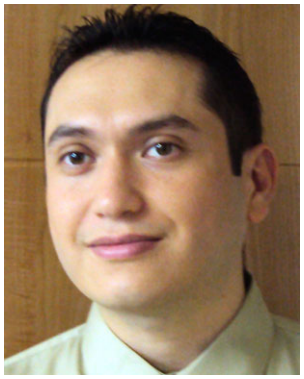
Since this work is among the earliest works in agent-based Cloud service composition, we focus on demonstrating the effectiveness of adopting agent-based techniques for Cloud service composition by showing the desirable property that our agents can autonomously and successfully deal with changing service requirements through self-organization and collaboration. In the future, we plan to conduct experiments on a much larger scale to evaluate the scalability of the agent-based Cloud service composition approach in real world settings by deploying the testbed using RESTful web services APIs for accessing Cloud resources following [5]. Additional directions for future work consist of: (i) Engineering agent behaviors for creating and maintaining SCTs through agent collaboration. (ii) Implementing and integrating utility-based functions into Cloud participants for determining appropriate quotations that maximize agents' earnings. (iii) Implementing and evaluating multi-round SR-CNP executions for conducting a distributed search among SPAs across multiple Clouds to determine the best (e.g., cheapest and/or most efficient regarding performance) Cloud service compositions.

Acknowledgements This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MEST) (KRF-2009-220-D00092). From May 18, 2010 through January 16, 2012, the first author was supported by a postdoctoral fellowship at the Multiagent and Cloud Computing Systems Laboratory at the Gwangju Institute of Science and Technology, South Korea. The first author acknowledges with thanks the support provided by Asociación Mexicana de Cultura A. C. from August 1, 2012. In addition, the authors would like to thank the Editor-in-Chief and the anonymous referees for their comments and suggestions.

References

1. Amazon EC2 FAQs (2012) http://aws.amazon.com/ec2/faqs/. Accessed 10 May 2012
2. Amazon EC2 Instance Types (2012) http://aws.amazon.com/ec2/instance-types/. Accessed 10 May 2012
3. Amazon Elastic Compute Cloud—Amazon EC2 (2012) http://aws.amazon.com/es/ec2/. Accessed 10 May 2012
4. Amazon Product Advertising API License Agreement (2012) https://affiliate-program.amazon.com/gp/advertising/api/detail/agreement.html. Accessed 10 May 2012
5. Battle R, Benson E (2008) Bridging the semantic web and web 2.0 with representational state transfer. J Web Semant 6(1):61–69
6. Bauer B, Müller JP, Odell J (2001) Agent uml: a formalism for specifying multiagent software systems. Int J Softw Eng Knowl Eng 11(3):207–230
7. Bellifemine F, Poggi A, Rimassa G (1999) JADE—a FIPA-compliant agent framework. In: Proc 4th international conference and exhibition on the practical application of intelligent agents and multi-agents, pp 97–108
8. Blake MB, Gomaa H (2005) Agent-oriented compositional approaches to services-based cross-organizational workflow. Decis Support Syst 40(1):31–50
9. Both F, Hoogendoorn M, Mee A, Treur J, Vos M (2012) An intelligent agent model with awareness of workflow progress. Appl Intell 36(2):498–510
10. Bryson J, Martin D, McIlraith S, Stein LA (2003) Agent-based composite services in daml-s: the behavior-oriented design of an intelligent semantic web. In: Zhong N, Liu J, Yao Y (eds) Web intelligence. Springer, Heidelberg, pp 37–58
11. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging it platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comput Syst 25(6):599–616
12. Chafle GB, Chandra S, Mann V, Nanda MG (2004) Decentralized orchestration of composite web services. In: Proc 13th international world wide web conference on alternate track papers & posters. ACM Press, New York, pp 134–143
13. Choudhary V (2007) Software as a service: implications for investment in software development. In: Proc 40th annual Hawaii international conference on system sciences. IEEE Computer Society, Washington, pp 209–218

14. Cloud Hosting, Cloud Servers, Hybrid Hosting, Cloud infrastructure from GoGrid (2012) http://www.gogrid.com/. Accessed 10 May 2012
15. Ferber J (1999) Multi-agent systems: an introduction to distributed artificial intelligence, 1st edn. Addison-Wesley, Longman, Boston
16. Gao J, Lv H (2012) Institution-governed cross-domain agent service cooperation: a model for trusted and autonomic service cooperation. Appl Intell 37(2):223–238
17. Gershenson C, Heylighen F (2003) When can we call a system self-organizing? In: Banzhaf W, Christaller T, Dittrich P, Kim JT, Ziegler J (eds) Advances in artificial life, ECAL 2003. LNAI, vol 2801. Springer, Heidelberg, pp 606–614
18. Google Cloud Services (2012) http://www.google.com/enterprise/cloud/. Accessed 10 May 2012
19. Graham RL, Knuth DE, Patashnik O (1994) Concrete mathematics, 2nd edn. Addison-Wesley, Reading
20. Gutierrez-Garcia JO, Sim KM (2010) Agent-based service composition in cloud computing. In: Kim TH, et al (eds) GDC/CA 2010. CCIS, vol 121. Springer, Heidelberg, pp 1–10
21. Gutierrez-Garcia JO, Sim KM (2010) Self-organizing agents for service composition in cloud computing. In: 2010 IEEE second international conference on cloud computing technology and science, USA, pp 59–66
22. Helal S, Wang M (2001) Service centric brokering in dynamic e-business agent communities. J Electron Commer Res 2(1):32–47
23. Helal S, Wang M, Jagatheesan A, Krithivasan R (2001) Brokering based self organizing e-service communities. In: Proc 5th international symposium on autonomous decentralized systems. IEEE Computer Society, Washington, pp 349–356
24. Hewitt C (1977) Viewing control structures as patterns of passing messages. Artif Intell 8(3):323–364
25. Heylighen F, Gershenson C (2003) The meaning of self-organization in computing. IEEE Intell Syst 18(4):72–75
26. Isern D, Moreno A, Sánchez D, Hajnal Á., Pedone G, Varga L (2011) Agent-based execution of personalised home care treatments. Appl Intell 34(2):155–180
27. Kang J, Sim KM (2012) A multiagent brokering protocol for supporting Grid resource discovery. Appl Intell. doi:10.1007/s10489-012-0347-y
28. Korhonen J, Pajunen L, Puustjarvi J (2003) Automatic composition of web service workflows using a semantic agent. In: Proc IEEE/WIC/ACM international conference on web intelligence. IEEE Computer Society, Washington, pp 5663–5666
29. Kuzu M, Cicekli N (2012) Dynamic planning approach to automated web service composition. Appl Intell 36(1):1–28
30. Lai K, Lin M, Yu T (2010) Learning opponent's beliefs via fuzzy constraint-directed approach to make effective agent negotiation. Appl Intell 33(2):232–246
31. Lenk A, Klems M, Nimis J, Tai S, Sandholm T (2009) What's inside the Cloud? An architectural map of the Cloud landscape. In: Proc 2009 ICSE workshop on software engineering challenges of Cloud computing. IEEE Computer Society, Washington, pp 23–31
32. Mei L, Chan WK, Tse TH (2008) A tale of clouds: paradigm comparisons and some thoughts on research issues. In: Proc IEEE Asia–Pacific services computing conference. IEEE Computer Society, Washington, pp 464–469
33. Mousavi A, Nordin MJ, Othman ZA (2012) Ontology-driven coordination model for multiagent-based mobile workforce brokering systems. Appl Intell 36(4):768–787
34. Murillo J, Muñoz V, Busquets D, López B (2011) Schedule coordination through egalitarian recurrent multi-unit combinatorial auctions. Appl Intell 34(1):47–63
35. Narayanan S, McIlraith S (2002) Simulation, verification and automated composition of web services. In: Proc 11th international world wide web conference. ACM Press, New York, pp 77–88
36. O'Shea K (2012) An approach to conversational agent design using semantic sentence similarity. Appl Intell. doi:10.1007/s10489-012-0349-9
37. Öztürk P, Rossland K, Gundersen O (2010) A multiagent framework for coordinated parallel problem solving. Appl Intell 33(2):132–143
38. Pallis G (2010) Cloud computing: the new frontier of internet computing. IEEE Internet Comput 14(5):70–73
39. Peltz C (2003) Web services orchestration and choreography. Computer 36(10):46–52
40. Sandholm T (1993) An implementation of the contract net protocol based on marginal cost calculations. In: Proc of the 11th national conference on artificial intelligence. AAAI Press, Menlo Park, pp 256–262
41. Sim KM (2006) Guest editorial: agent-based grid computing. Appl Intell 25(2):127–129
42. Sim KM (2009) Agent-based cloud commerce. In: Proc IEEE international conference on industrial engineering and engineering management, Hong Kong, pp 717–721
43. Sim KM (2010) Towards complex negotiation for cloud economy. In: Chang RS, et al (eds) GPC 2010. LNCS, vol 6104. Springer, Heidelberg, pp 395–406
44. Sim KM (2011) Agent-based cloud computing. IEEE Trans Serv Comput. doi:10.1109/TSC.2011.52
45. Sim KM (2012) Complex and concurrent negotiations for multiple interrelated e-markets. IEEE Trans Syst Man Cybern B Cybern. doi:10.1109/TSMCB.2012.2204742
46. Smith RG (1980) The contract net protocol: high-level communication and control in a distributed problem solver. IEEE Trans Comput 29(12):1104–1113
47. Sun Microsystems, Inc (2012) Determining acceptable response delays. http://java.sun.com/products/jlf/at/book/Responsiveness5.html. Accessed 10 May 2012
48. Sycara KP (1998) Multiagent systems. AI Mag 19(2):79–92
49. VMware Public Cloud Computing Resources (2012) http://www.vmware.com/solutions/cloud-computing/public-cloud/resources.htmlttp://java.sun.com/products/jlf/at/book/Responsiveness5.html. Accessed 10 May 2012
50. Vouk MA (2008) Cloud computing—issues, research and implementations. J Comput Inf Technol 16(4):235–246
51. Wang S, Shen W, Hao Q (2006) An agent-based web service workflow model for inter-enterprise collaboration. Expert Syst Appl 31(4):787–799
52. Weiss G (1999) Multiagent systems: a modern approach to distributed artificial intelligence. MIT Press, Cambridge
53. Wooldridge M (2009) An introduction to multiagent systems, 2nd edn. Wiley, Chichester
54. World Wide Web Consortium—W3C (2012) Web services Architecture, http://www.w3.org/TR/ws-arch/. Accessed 10 May 2012
55. Zeng C, Guo X, Ou W, Han D (2009) Cloud computing service composition and search based on semantic. In: Jaatun MG, Zhao G, Rong C (eds) CloudCom. LNCS, vol 5931. Springer, Heidelberg, pp 290–300
56. Zou G, Chen Y, Yang Y, Huang R, Xu Y (2010) AI planning and combinatorial optimization for web service composition in cloud computing. In: Proc international conference on cloud computing and virtualization, pp 1–8



J. Octavio Gutierrez-Garcia received his Ph.D. in Electrical Engineering and Computer Science from CINVESTAV (Mexico) and Grenoble Institute of Technology (France), respectively. Dr. Gutierrez-Garcia is an Associate Professor in the Department of Computer Science at Instituto Tecnologico Autonomo de Mexico. He has served as a reviewer for numerous international conferences and as a member of various editorial boards of scientific journals. His current research interests include Cloud computing, distributed artificial intelligence, multiagent systems, and service-oriented computing.

Kwang Mong Sim is the Medway Chair and Professor in Computer Science at the University of Kent, Chatham Maritime, Kent, UK. Professor Sim was the Founder and Director of the Multiagent and Cloud Computing Laboratory at the Gwangju Institute of Science and Technology in South Korea. He is an Associate Editor for the IEEE Transactions on Systems, Man and Cybernetics-Part C, the International Journal of Cloud Computing and many other international journals. He is also the Guest Editor of five (IEEE) journal special issues in agent-based Grid computing and automated negotiation, including a special issue on Grid Resource Management in the IEEE Systems Journal (IEEE Systems Council). He served as a referee for several national research grant councils including the National Science Foundation in USA. He has delivered many keynote lectures on agent-based Cloud computing and automated negotiation in many international conferences, and has published many survey papers, technical papers, and award-winning conference papers in Cloud computing and multiagent systems. He received his Ph.D. and M.Sc. from the University of Calgary, AB, Canada, and graduated Summa Cum Laude with a B.Sc. (Hon) from the University of Ottawa, ON, Canada.