Self-Organizing Agents for Service Composition in Cloud Computing

J. Octavio Gutierrez-Garcia & Kwang Mong Sim

joseogg@gmail.com / kmsim@gist.ac.kr
Cloud Service Composition

- Cloud services, which are deployed as self-contained components, are normally partial solutions that must be composed to provide a single virtualized service to Cloud consumers.

- This composition of services should be carried out in a dynamic and automated manner to promptly satisfy consumer requirements.

- Cloud-computing environments pose new challenges to automated service composition:
  - Dynamically contracting service providers, which set service fees on a supply-and-demand basis
  - Dealing with incomplete information regarding Cloud resources (e.g., location and providers).
Self-organizing agents

- Self-organizing systems are composed of interacting agents.

- Interaction among agents adapts and evolves the system to achieve Cloud service compositions.

- The Cloud service composition is determined by the feedback (e.g., service fees) obtained through the free interaction of nearby agents (cloud consumers/broker agents/service providers).

- Agents can collaborate to achieve shared objectives, even when self-interest behaviors to maximize utility are adopted.
Cloud service composition

- Cloud participants and Cloud resources are represented and instantiated by agents.

- The self-organizing service composition is supported by:
  - Acquaintance networks.
    - Incomplete list of known cloud services and its capabilities.
  - The contract net protocol.
    - Dynamically selecting services based on service fees.
A distributed Cloud service composition architecture

- **Consumer agents** (CAs) formalize consumer requirements and submit them to brokers.

- **Broker agents** (BAs) compose and provide a single virtualized service to Cloud consumers.

- **Service provider agents** (SPAs) manage Cloud providers’ resources by controlling and organizing RAs.

- **Resource agents** (RAs) orchestrate web services and control the access to them.

- **Web services** are interfaces to software applications or Cloud resources.
Acquaintance Networks

Consumer Agent \( k \)

Broker Agent \( k \)

Acquaintance Network of BAs

Quantitative Information

SPA1 \( \ldots \) SPAi \( \ldots \) SPAn

Cap1 1 0 1 1 1
Capk 1 1 0 1 0

Acquaintance Network of SPAs

Acquaintance Network of BAs

Dynamic, Incomplete, and Exact Tables

Define service capabilities

Establish consumer requirements

ELEMENTS

- Web service (WS)
- Consumer agent (CA)
- Resource agent (RA)
- Service provider agent (SPA)
- Broker agent (BA)
- Acquaintance network & delegated allocation
- Acquaintance network & imposed allocation
Acquaintance Networks

Service Provider Agent $k$

Resource Agent $k$

Acquaintance Network of SPAs

<table>
<thead>
<tr>
<th>SPA1</th>
<th>...</th>
<th>SPAi</th>
<th>...</th>
<th>SPAn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cap1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Capk</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Dynamic, Incomplete, and Exact Table

Acquaintance Network of RAs

<table>
<thead>
<tr>
<th>RA1</th>
<th>...</th>
<th>RAi</th>
<th>...</th>
<th>RAn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cap1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Capk</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Static, Complete, and Exact Table

Acquaintance Network of Sibling RAs

<table>
<thead>
<tr>
<th>RA1</th>
<th>...</th>
<th>RAi</th>
<th>...</th>
<th>RAn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cap1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Capk</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Static, Complete, and Exact Table

Elements:
- Web service (WS)
- Resource agent (RA)
- Service provider agent (SPA)
- Broker agent (BA)
- Consumer agent (CA)
- Acquaintance network & delegated allocation
- Acquaintance network & imposed allocation

Define service capabilities

Establish consumer requirements

Accept-Proposal

Service ontology
Agents adopt the contract net protocol for selecting and (sub) contracting resource needs to resolve consumer requirements.
The main behavior of a CA is derived from the *contract-net-protocol initiator* behavior that submits consumer requirements to broker agents.

**CNP_consumer_initiator behavior**

- **Input:** (i) Consumer requirements
- **Output:** (i) A single virtualized service

1. Decompose consumer requirements
2. for \(i \leftarrow 1; \ 1 \leq n\text{Requirements}; \ i++\) do
3. Send `call_for_proposals(Req)` to \(m\) acquaintances(BAs)
4. \(n\text{Proposals} \leftarrow \text{BlockReceive(Proposals, timeout)}\)
5. if \((n\text{Proposals} > 0)\) then
6. Send `reject_proposal` to \((n\text{Proposals}-1)\) BAs
7. Send `accept_proposal` to 1 BA
8. BlockReceive(Req, Output)
9. else
10. Throw exception
11. Integrate outputs into a single virtualized service
The contract-net-protocol participant behavior handles proposals to fulfill requirements coming from consumer agents or other broker agents when subcontracting is required.

**CNP_broker_participant behavior**

**Input:** (i) call_for_proposals from CAs or other BAs  
**Output:** (i) Instantiation of a Request_evaluator behavior

1: BlockReceive(call_for_proposals(Req))  
2: if (not visitedFor(Req)) then  
3: Prepare and Send Proposal  
4: BlockReceive(reply, timeout)  
5: if (reply = accept_proposal) then  
6: Instantiate a Request_evaluator(Req) behavior  
7: Else  
8: Start over  
9: else  
10: Send refuse message  
11: Start over
Broker agent behaviors

- The *request-evaluator* behavior verifies whether the proposal can be resolved by contracting SPAs’ acquaintances or whether another broker agent must be subcontracted.

```
Request_evaluator behavior

Input: (i) Requirement Req,
Output: (i) Instantiation of a CNP_broker_initiator(BAs) behavior or
       (ii) instantiation of a CNP_broker_initiator(SPAs) behavior or
       (iii) failure propagation

1: Check_for_feasible_acquaintances(SPAs, Req)
2: if (nFeasibleSPAs > 0) then
3:   Instantiate a CNP_broker_initiator(SPAs, Req) behavior
4: else
5:   Check_for_feasible_acquaintances(BAs)
6:   if (nFeasibleBAs > 0) then
7:     Instantiate a CNP_broker_initiator(BAs, Req) behavior
8:   else
9:     Send failure message to requester
```
Broker agent behaviors

- The *contract-net-protocol initiator* behavior submits requirements to possible contractors, either BAs or SPAs.

---

**CNP_broker_initiator behavior**

Input: (i) Contractors (BAs or SPAs) and (ii) Requirement $Req_i$

Output: (i) Instantiation of a *Result_handler* behavior or (ii) failure propagation

1. Check_for_feasible_acquaintances(Contractors)
2. if (nContractors > 0) then
3. The agent marks itself as visited for $Req_i$
4. Send *call_for_proposals($Req_i$)* to $m$ acquaintances(Contractors)
5. nProposals ← BlockReceive(Proposals, timeout)
6. if (nProposals > 0) then
7. Send *reject_proposal* to (nProposals-1) Contractors
8. Send *accept_proposal* to 1 Contractor
9. Instantiate a *Result_handler($Req_i$)* behavior
10. else
11. if (Contractors = SPAs) then
12. Instantiate a *CNP_broker_initiator(BAs, $Req_i$)* behavior
13. else-if (Contractors = BAs) then
14. Send *failure* message to the original requester (CA or BA)
15. else
16. if (Contractors = SPAs) then
17. Instantiate a *CNP_broker_initiator(BAs, $Req_i$)* behavior
18. else-if (Contractors = BAs) then
19. Send *failure* message to the original requester (CA or BA)
Broker agent behaviors

- The *result-handler* behavior receives outputs from SPAs/BAs regarding previously delegated requirements, and propagates the outputs to the original requesters either CAs or BAs.
- In case of receiving a failure message, the requirement is delegated to the remaining feasible SPAs.

### Result_handler behavior

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

1. BlockReceive(Result)
2. if (Result = Output) then
3. Send *inform_result* to its requester
4. else-if (Result = Failure) then
5. Add Result.sender to unfeasible agents
6. if (Result.sender = BA) then
7. Send *failure* to the original requester
8. else-if (Result.sender = SPA) then
9. Instantiate a *CNP_broker_initiator(SPAs, Req.)* behavior
The delegation of requirements to resource agents is done via the CNP-Initiator($RAs, Req_i$) behavior. However, the proposals of resource agents contain their availability, e.g., available or busy.

1. Only feasible RAs are contacted.
2. Looking for available Resources agents to delegate $r$.
3. Delegating requirement $r$. Only feasible RAs are contacted.
Service provider agent behaviors

- A SPA may subcontract services to other SPAs when
  - its RAs fail,
  - its RAs, as the normal process of resolving a given requirement, request to its SPA the fulfillment of an external requirement.

<table>
<thead>
<tr>
<th>Resource_agent_listener behavior</th>
</tr>
</thead>
</table>

**Input:** (i) Request to resolve $Req_i$

**Output:** (i) Instantiation of a $CNP\_provider\_initiator(\text{SPAs})$ behavior

1: BlockReceive(Request($Req_i$))
2: Instantiate a $CNP\_provider\_initiator(\text{SPAs}, Req_i)$ behavior
3: Start over
The *contract-net-protocol participant* behavior accepts new requests from the SPA or sibling RAs.

---

**CNP_resource_participant behavior**

<table>
<thead>
<tr>
<th>Input:</th>
<th>(i) <em>call_for_proposals</em> from the SPA or sibling RAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>(i) Initiation of the <em>Ad_hoc workflow</em> behavior</td>
</tr>
</tbody>
</table>

1. BlockReceive(*call_for_proposals*(Req,i))
2. if *(available and capableOf)(Req)* then
3. Send a *propose* message to indicate availability
4. BlockReceive(reply, timeout)
5. if *(reply = accept_proposal)* then
6. Initiate the *Ad_hoc workflow* behavior
7. Else
8. Start over
9. else
10. Send *refuse* message
11. Start over
Resource agent behaviors

- Behaviors of resource agents are pattern behaviors that allow specifying an ad-hoc web service workflow.

- The objective of the *Ad-hoc workflow* behavior is to fulfill a requirement and pass the result to either the SPA or a sibling RA.

---

**Ad_hoc workflow behavior**

- **Input:** (i) An ad-hoc workflow specification
- **Output:** (i) `inform_results(Req.Output)` or (ii) instantiation of a `CNP_resource_initiator(RAs)` behavior

```python
1: try
2: Ad-hoc workflow specification
3: Send `inform_results(Req.Output)` to the SPA or sibling RAs
4: catch(exception)
5: Instantiate a `CNP_resource_initiator(RAs)` behavior
```
The *contract-net-protocol initiator* behavior handles the imposed delegation of requirements to sibling RAs.
Resource agent behaviors

- The *internal-delegator* behavior delegates a requirement to a specific sibling RA and waits for its resolution.

---

**Internal_delegator behavior**

**Input:** (i) Requirement Req, to be delegated, and (ii) sibling RA’s id

**Output:** (i) Requirement delegation or (ii) failure propagation

1. Send *call_for_proposals(Req)* to a sibling RA
2. Proposal ← BlockReceive(Proposal, timeout)
3. if (Proposal = Available) then
4. Send *accept_proposal* to the sibling RA
5. BlockReceive(Result)
6. Continue with the *Ad_hoc workflow behavior*
7. else
8. Throw exception
The \textit{external-delegator} behavior delegates a requirement to the SPA and waits for its resolution.

### External_delegator behavior

**Input:** (i) Requirement $Req_i$ to be delegated  
**Output:** (i) Requirement’s output

1: Send $Request(Req_i)$ to the SPA  
2: BlockReceive(Result)  
3: if (Result = Failure) then  
4: Halt the \textit{Ad_hoc workflow} behavior  
5: else-if (Result = Output) then  
6: Continue with the \textit{Ad_hoc workflow} behavior
Objectives:
- To evaluate self-organizing characteristics of the agents during Cloud service composition.
- To evaluate the efficiency relation between exchanged messages and the # of agents’ acquaintances.

Experimental settings:
- Three types of Cloud resources:
  - A - memory instance
  - B - CPU instance
  - C - cluster instance
- Consumer service request \{A, B, C\}
- Resource agents were designed to fail with probabilities ranging from 0.0 to 1.0
- Service fees were randomly determined.
- Five service compositions per failure rate.

Performance measures:
- # of successful service compositions.
- # of messages exchanged.
Empirical Evaluation

- The number of successful compositions increased as the degree of agents’ connectivity increased.
- More connected agents’ acquaintance networks allow accessing more Cloud resources, and thus, having a higher probability of success.

- The number of messages exchanged increased as the probability of failure increased and the degree of agents’ connectivity increased.
- The more connected agents are, the more self-organization can be expressed. This results in a minor increment of the number of messages in exchange for a major efficacy.
Conclusions

- The **novelty** and **significance** of this paper is that distributed and cooperative agent-based problem solving techniques such as acquaintance networks and the contract net protocol were used to create a self-organizing service composition method.

- The first work in considering incomplete information about Cloud participants and its combination with dynamic service selection mechanisms.
Conclusions

- A test bed that evaluated and demonstrated the advantages of self-organizing agents in Cloud service composition was implemented.

- Patterns for agent behaviors that handle ad-hoc web service workflow specifications were designed.

- Dynamic and Automated Self-organizing service composition was supported by (sub) contracts among Cloud participants.
Future work

- Designing mechanisms to create and maintain acquaintance networks.
- Engineering agents’ decision-making process that considers complex proposals.
- Designing mechanisms to adjust existent service compositions to constantly changes in consumer requirements.
- Deploying the agent-based testbed in a semantic web service framework using RESTful web services.
Self-Organizing Agents for Service Composition in Cloud Computing

J. Octavio Gutierrez-Garcia & Kwang Mong Sim
joseogg@gmail.com / kmsim@gist.ac.kr

Questions?

CloudCom 2010
2nd IEEE International Conference on Cloud Computing Technology and Science
Web services calling other web services

- Examples when interaction is required:
  - Asking for public keys in encrypted communication.
  - Granting access to resources.
  - Retrieving global consecutive numbers, e.g., invoice control numbers.
  - Validating credentials or payments.
Agent interaction scenarios

CONSUMER AGENTS  BROKER AGENTS

CFP to achieve a Cloud Service Composition

CFP to achieve a Cloud Service Composition

BROKER AGENTS  SERVICE PROVIDER AGENTS

CFP to resolve a set of Requirements

CFP to resolve a set of Requirements

SERVICE PROVIDER AGENT  RESOURCE AGENTS

Only feasible RAs are contacted

Looking for available Resources agents to delegate r

Available

Busy

Available

Delegating requirement r

SERVICE PROVIDER AGENT  RESOURCE AGENTS

Available

Busy

Available

Only feasible RAs are contacted

Looking for available Resources agents to delegate r

Available

Busy

Available

Delegating requirement r