SOS Cloud: Self-Organizing Services in the Cloud

Bogdan A. Caprarescu, Nicolò M. Calcavecchia, Elisabetta Di Nitto, Daniel J. Dubois

Politecnico di Milano, Dipartimento di Elettronica e Informazione

Piazza Leonardo da Vinci, 32 – 20133 Milano, Italy

We propose a bio-inspired, self-organizing solution for virtual machines provisioning and service deployment in a cloud infrastructure. The goal is twofold: meet the QoS of each service and minimize the number of virtual machines.

Motivation

"**Cloud computing** is Web-based processing, whereby shared resources, software, and information are provided to computers and other devices on demand over the Internet." [1]

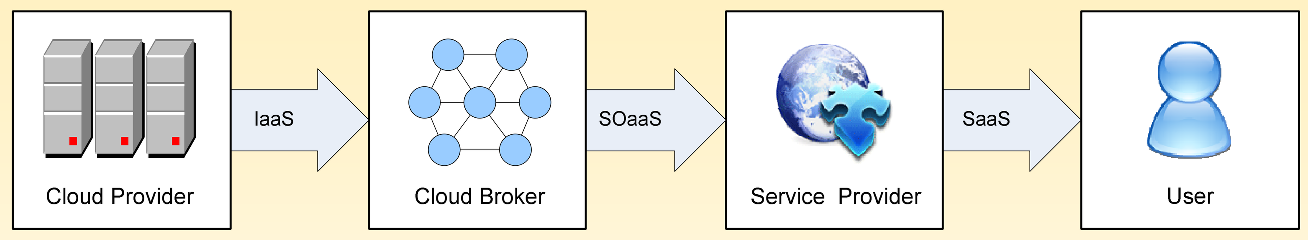
Top two obstacles to the massive adoption of cloud computing [2] and **our solutions**:

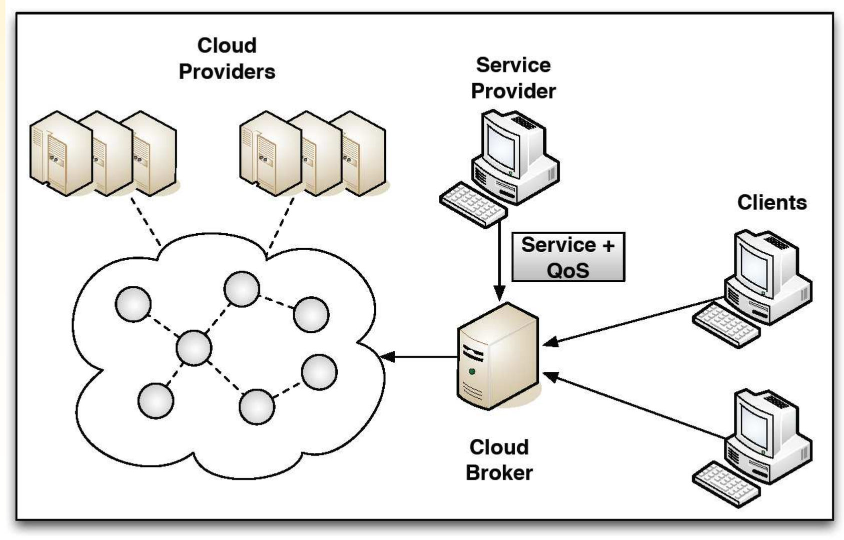
Obstacle	Our solution
Inability to guarantee high availability of services	Service Optimization as a Service (SOaaS)
Lock-in to a cloud provider	Cloud Broker relying on multiple cloud providers

Cloud Broker and Service Optimization as a Service

Cloud Broker

- Is a company that rents virtual machines from cloud providers
- Receives services and their QoS from service providers
- Dynamically provisions virtual machines to services
- Balances the virtual machines (named **nodes)** among many services
- Deploys exactly one service on a node at a time (for security reasons)
- For the beginning, we assume one cloud provider and one data center





Problem

Problem: Build an autonomic system that dynamically provisions nodes to services with two goals: meet QoS of each service and minimize the number of nodes.

Inputs: for each service we know

- *QoS* expressed through two parameters: *maximum response time* and *maximum rejection rate* (a request not answered within the maximum response time is considered rejected).
- *Estimated processing time*
- *Request rate*

A Self-Organizing Solution

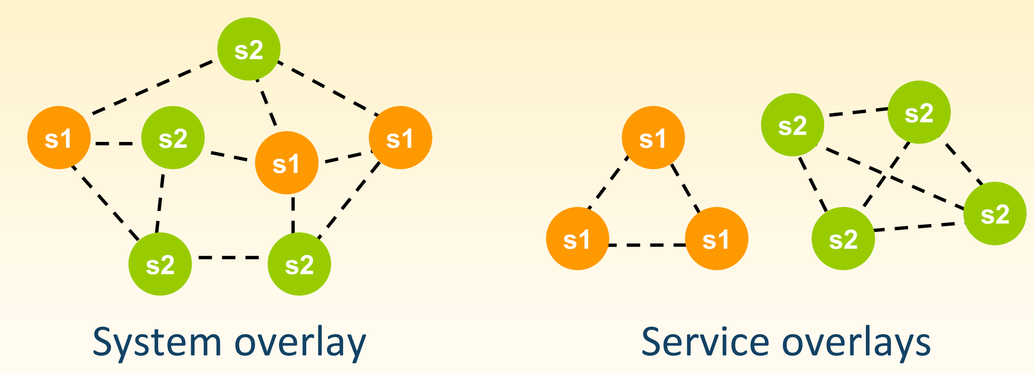
Why not a centralized solution?

- A central manager introduces a single-point of failure and becomes a scalability bottleneck.

A self-organizing architecture

- Inspired by the self-organizing systems in nature (e.g., ant colonies)
- The nodes are organized into overlays and each node knows only about a few other nodes (called *neighbors*).

The overlays of a couple of nodes that run two services: s1 and s2.



- Each node is part of two overlays: its *service overlay* (used for node provisioning and request routing) and the *system overlay* (used for changing the service of a node).
- The *neighborhood* of one node with respect to an overlay is the set composed of the node itself and its neighbors in that overlay.

Node Provisioning

Utility Function

An **utility function** is defined to assess the performance of a node.

Each node tries to maximize the **average utility of its service neighborhood**.

$$U(n) = \frac{U^{SLA}(n) + U^{CPU}(n)}{2}$$

$$U^{SLA}(n) = \begin{cases} \frac{P_s - P_n}{P_s} & \text{if } P_n < 2 * P_s \\ C & \text{otherwise} \end{cases} \qquad U^{CPU}(n) = \begin{cases} \frac{L_n}{L_{des}} & \text{if } L_n < L_{des} \\ \frac{100 - L_n}{100 - L_{des}} & \text{otherwise} \end{cases}$$

- P_s: QoS maximum rejection rate
- P_n: actual rejection rate of node *n*
- *C*: a dominant penalty
- L_n: CPU utilization of node *n*
- L_{des}: desired CPU utilization threshold

Monitoring

Each node monitors its performance and maintains one management table for each overlay.

A **management table** stores the following data for each neighbor:

- IP address and port number
- Service
- Rejected requests rate
- CPU utilization

A gossip protocol [3] is used to update the tables.

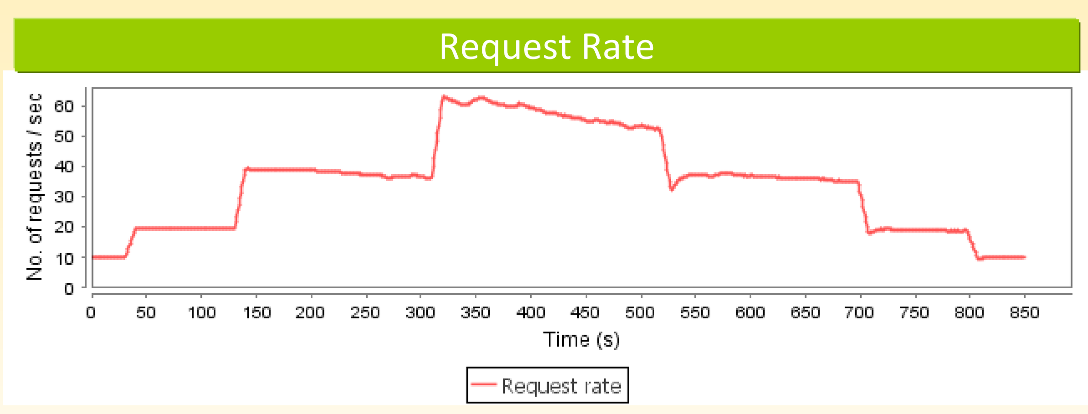
Analysis and Execution

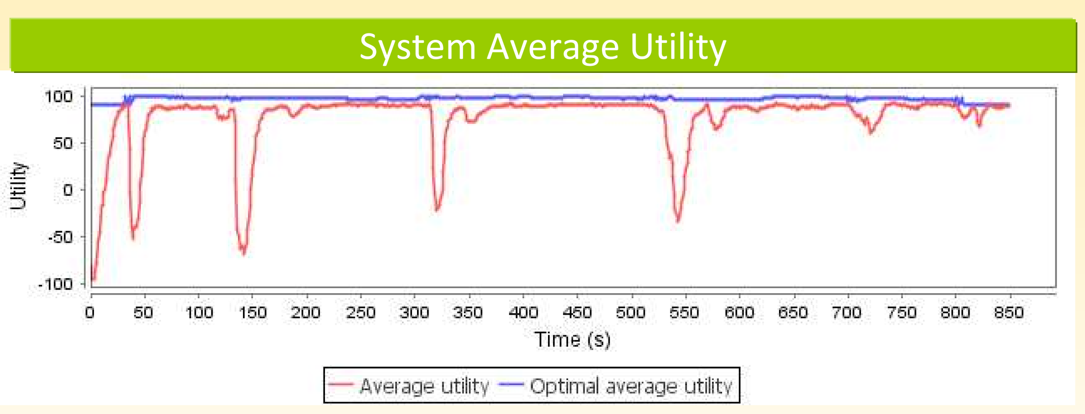
```
do forever
    wait(T)
    success = acquireLock()
    if success
        currentLoad = getCpuUtilization()
            if currentLoad < DESIRED_LOADING - LOADING_MARGIN
                underUtilized()
            else if currentLoad > DESIRED_LOADING + LOADING_MARGIN
                overUtilized()
```

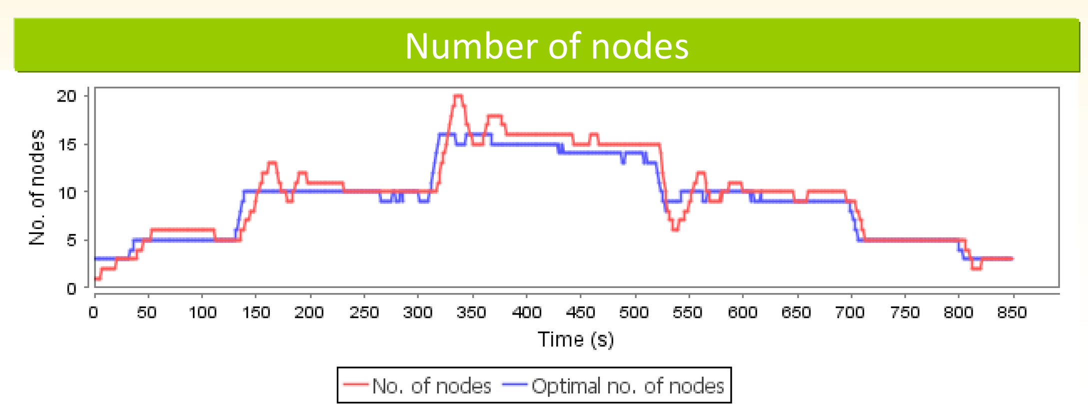
```
underUtilized()
cServNhood = getServNhood()
cUtility = cServNhood.compUtility()
pServNhood = predictRemSelf(cServNhood)
pUtility = pServNhood.compUtility()
if cUtility < pUtility - UTILITY_MARGIN
    if isCloudTimeQuotaAboutToExpire()
        removeMyself()
    else
        s = selectAnotherService()
        switchTo(s)
```

```
overUtilized()
cServNhood = getServNhood()
cUtility = cServNhood.compUtility()
pServNhood = predictAddNode(cServNhood)
pUtility = pServNhood.compUtility()
if cUtility < pUtility - UTILITY_MARGIN
    if random(100) < PROBABILITY
        addNewNode()
```

Simulation Results







A custom simulation was implemented in Java.

The figures show the average results of a couple of tests where just one service was considered.

Following a few oscillations, at constant request rate, the system stabilizes : the QoS is met and the number of nodes is close to the optimal one.

Conclusion

A self-organizing approach for virtual machines provisioning and service deployment in the cloud was described and simulated.

Although the solution is not optimal, the built-in robustness and scalability of the architecture pay the trade-off.

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Cloud_computing

[2] M. Ambrust at al. A view of cloud computing. *Communications of the ACM*, 53:4, 2010.

[3] M. Jelasity et al. Gossip-based Peer Sampling. *ACM Transactions on Computer Systems*, 25:3, 2007.