This paper appears in the 6th International Conference on Network and Service Management, Niagara Falls, Ontario, Canada, October 25-29, 2010.

# Gossip-based Resource Management for Cloud Environments

Fetahi Wuhib and Rolf Stadler
ACCESS Linnaeus Center
KTH Royal Institute of Technology
Email: {fetahi,stadler}@kth.se

Mike Spreitzer
IBM T.J. Watson Research Center
Email: mspreitz@us.ibm.com

***Abstract*—We address the problem of resource management for a large-scale cloud environment that hosts sites. Our contribution centers around outlining a distributed middleware architecture and presenting one of its key elements, a gossip protocol that meets our design goals: fairness of resource allocation with respect to hosted sites, efficient adaptation to load changes and scalability in terms of both the number of machines and sites. We formalize the resource allocation problem as that of dynamically maximizing the cloud utility under CPU and memory constraints. While we can show that an optimal solution without considering memory constraints is straightforward (but not useful), we provide an efficient heuristic solution for the complete problem instead. We evaluate the protocol through simulation and find its performance to be well-aligned with our design goals.**

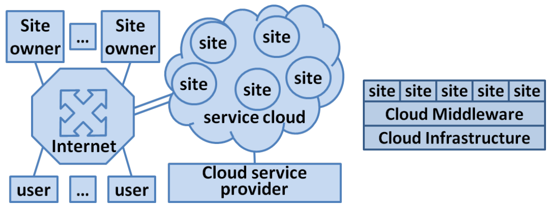
***Index Terms*—cloud computing, distributed management, resource allocation, gossip protocols**

## I. Introduction

We consider the problem of resource management for a large-scale cloud environment. Such an environment includes the physical infrastructure and associated control functionality that enables the provisioning and management of cloud services. The perspective we take is that of a *cloud service provider*, which hosts *sites* in a cloud environment.

The stakeholders are depicted in Figure 1a. The cloud service provider owns and administrates the physical infrastructure, on which cloud services are provided. It offers hosting services to site owners through a middleware that executes on its infrastructure (See Figure 1b). Site owners provide services to their respective users via sites that are hosted by the cloud service provider.

The type of sites we have in mind for this work ranges from smaller websites (that are hosted on a single physical machine) up to medium-scale e-commerce sites (whose demand can be satisfied with the resources of a few dozen machines per site). A cloud, by which we mean the infrastructure and the middleware depicted in 1b, would run *millions* of such sites and includes *hundreds of thousands* of machines.



(a) (b)

Fig. 1. (a) Deployment scenario with the stakeholders of the cloud environment considered in this work. (b) Overall architecture of the cloud environment; this work focuses on resource management performed by the middleware layer.

This work contributes towards engineering a middleware layer that performs resource allocation in such a cloud environment, with the following design goals.

1) Performance objective: We consider computational and memory resources, and the objective is to achieve max-min fairness for computational resources under memory constraints.
2) Adaptability: The resource allocation process must dynamically and efficiently adapt to changes in the demand for cloud services.
3) Scalability: The Resource allocation process must be scalable both in the number of machines in the cloud and the number of sites that the cloud hosts. This means that the resources consumed per machine in order to achieve a given performance objective must increase sublinearly with both the number of machines and the number of sites.

Our approach centers around a decentralized design whereby the components of the middleware layer run on every processing node of the cloud environment. (We refer to a processing node as a *machine* in the remainder of the paper.) To achieve scalability, we envision that all key tasks of the middleware layer, including estimating global states, placing site modules and computing policies for request forwarding are based on distributed algorithms. Further, we rely on a global directory for routing requests from users on the Internet to access points to particular sites inside the cloud.

The core contribution of the paper is a gossip protocol that can be used to meet the design goals above. While gossip protocols have been studied before for load balancing in distributed systems, there are no results available (to our knowledge) for the case of considering memory constraints and the cost of a configuration change, which make the resource allocation problem much harder. Coming up with an optimal and efficient solution for the problem that does not consider memory constraints is quite straightforward (but not useful in our context), and we can only provide an efficient heuristic for the problem we set out to solve, since the memory constraints make it NP-hard.

The paper is structured as follows. Section II outlines the architecture of a middleware layer that performs resource management for a large-scale cloud environment. Section III formalizes the resource allocation problem. Section IV presents two protocols to solve this problem, one of which is evaluated through simulation in Section V. Section VI reviews related work, and Section VII contains the conclusion of this research and outlines future work.

## II. System Architecture

A cloud environment spans several datacenters interconnected by an internet. Each of these datacenters contains a large number of machines that are connected by a high-speed network. Users access sites hosted by the cloud environment through the public Internet. A site is typically accessed through a URL that is translated to a network address through a global directory service, such as DNS. A request to a site is routed through the Internet to a machine inside a datacenter that either processes the request or forwards it. In this paper, we restrict ourselves to a cloud that spans a datacenter containing a single cluster of machines and leave for further work the extension of our contribution to an environment including multiple datacenters.

Figure 2 (left) shows the architecture of the cloud middleware. The components of the middleware layer run on all machines. The resources of the cloud are primarily consumed by module instances whereby the functionality of a site is made up of one or more modules. In the middleware, a module either contains part of the service logic of a site (denoted by $m_i$ in Figure 2) or a site manager (denoted by $SM_i$).

Each machine runs a *machine manager* component that computes the resource allocation policy, which includes deciding the module instances to run. The resource allocation policy is computed by a protocol (later in the paper called P) that runs in the *resource manager* component. This component takes as input the projected demand for each module that the machine runs. The computed allocation policy is sent to the *module scheduler* for implementation/execution, as well as the *site managers* for making decisions on request forwarding. The *overlay manager* implements a distributed algorithm that maintains an overlay graph of the machines in the cloud and provides each resource manager with a list of machines to interact with.

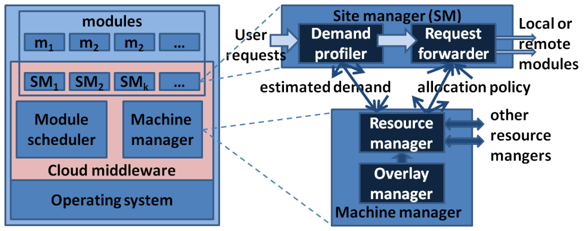


Fig. 2. The architecture for the cloud middleware (left) and components for request handling and resource allocation (right).

Our architecture associates (one or more) site manager with each site(s). Each site manager handles user requests to a particular site. It has two important components: a *demand profiler* and *request forwarder*. The *demand profiler* estimates the resource demand of each module of the site based on the request statistics, QoS targets, etc. (An example of such a profiler can be found in [1].) This estimate is forwarded to all machine managers that run instances of modules belonging to this site. Similarly, the *request forwarder* sends user requests for processing to instances of modules belonging to this site. Request forwarding decisions take into account the resource allocation policy and constraints such as session affinity. Figure 2 (right) shows the components of a site manager and how they relate to machine managers.

The remainder of this paper focuses on the functionality of the resource manager component. For other components of our architecture, such as overlay manager and demand profiler we rely on known solutions.

## III. Formalizing the Problem of Resource Allocation by the Cloud Middleware

For this work, we consider a cloud as having computational resources (i.e., CPU) and memory resources, which are available on the machines in the cloud infrastructure. As explained earlier, we restrict the discussion to the case where all machines belong to a single cluster and cooperate as peers in the task of resource allocation. The specific problem we address is that of placing modules (more precisely: identical instances of modules) on machines and allocating cloud resources to these modules, such that a cloud utility is maximized under constraints. As cloud utility we choose the minimum utility generated by any site, which we define as the minimum utility of its module instances. We formulate the resource allocation problem as that of maximizing the cloud utility under CPU and memory constraints. The solution to this problem is a configuration matrix that controls the module scheduler and request forwarder components. At discrete points in time, events occur, such as load changes, addition and removal of site or machines, etc. In response to such an event, the optimization problem is solved again, in order to keep the cloud utility maximized. We add a secondary objective to the optimization problem, which states that the cost of change from the current configuration to the new configuration must be minimized.

### *A. The Model*

We model the cloud as a system with a set of sites $S$ and a set of machines $N$ that run the sites. Each site $s \in S$ is composed of a set of modules denoted by $M_s$, and the set of all modules in the cloud is $M = \bigcup_{s \in S} M_s$.

We model the CPU demand as the vector $\boldsymbol{\omega}(t) = [\omega_1(t), \omega_2(t), \dots, \omega_{|M|}(t)]^T$ and the memory demand as the vector $\boldsymbol{\gamma} = [\gamma_1, \gamma_2, \dots, \gamma_{|M|}]^T$, assuming that CPU demand is time dependent while memory demand is not [2].

We consider a system that may run more than one instance of a module $m$, each on a different machine, in which case its CPU demand is divided among its instances. The demand $\omega_{n,m}(t)$ of an instance of $m$ running on machine $n$ is given by $\omega_{n,m}(t) = \alpha_{n,m}(t)\omega_m(t)$ where $\sum_{n \in N} \alpha_{n,m}(t) = 1$ and $\alpha_{n,m}(t) \geq 0$. We call the matrix $A$ with elements $\alpha_{n,m}(t)$ the *configuration (matrix)* of the system. $A$ is a non-negative matrix with $\mathbf{1}^T A = \mathbf{1}^T$.

A machine $n \in N$ in the cloud has a CPU capacity $\Omega_n$ and a memory capacity $\Gamma_n$. We use $\boldsymbol{\Omega}$ and $\boldsymbol{\Gamma}$ to denote the vectors of CPU and memory capacities of all the machines in the system. An instance of module $m$ running on machine $n$ demands $\omega_{n,m}(t)$ CPU resource and $\gamma_m$ memory resource from $n$. Machine $n$ allocates to module $m$ the CPU capacity $\hat{\omega}_{n,m}(t)$ (which may be different from $\omega_{n,m}(t)$) and the memory capacity $\gamma_m$. The value for $\hat{\omega}_{n,m}(t)$ depends on the allocation policy in the cloud, and our specific policy $\hat{\Omega}(t)$ is described in Section IV-A.

We define the utility $u_{n,m}(t)$ generated by an instance of module $m$ on machine $n$ as the ratio of the allocated CPU capacity to the demand of the instance on that particular machine, namely, $u_{n,m}(t) = \frac{\hat{\omega}_{n,m}(t)}{\omega_{n,m}(t)}$. (An instance with $\omega_{n,m} = 0$ generates a utility of $\infty$.) We further define the utility of a module $m$ as $u_m(t) = \min_{n \in N}\{u_{n,m}(t)\}$ and that of a site as the minimum of the utility of its modules. Finally, the utility of the cloud $U^c$ is the minimum of the utilities of the sites it hosts. As a consequence, the utility of the cloud becomes the minimum utility of any module instance in the system.

| | |
|---|---|
| $S$, $N$, $M$ | set of all sites, machines and modules |
| $M_s$, $s \in S$ | set of modules of site $s$ |
| $\boldsymbol{\omega}(t), \boldsymbol{\gamma} \in \mathbf{R}^{\|M\|}$ | CPU and memory demand vectors |
| $A(t) \in \mathbf{R}^{\|N\| \times \|M\|}$ | configuration matrix |
| $\omega_{n,m}(t) \in \mathbf{R}$ | CPU demand of the instance of module $m$ on machine $n$ |
| $\hat{\omega}_{n,m}(t) \in \mathbf{R}$ | CPU allocated to the instance of module $m$ on machine $n$ |
| $\hat{\Omega}(t) \in \mathbf{R}^{\|N\| \times \|M\|}$ | CPU allocation matrix with elements $\hat{\omega}_{n,m}(t)$ |
| $\boldsymbol{\Omega}, \boldsymbol{\Gamma} \in \mathbf{R}^{\|N\|}$ | CPU and memory capacity vectors |
| $u_{n,m}(t)$, $U^c$ | utility generated by an instance of module $m$ on machine $n$, utility generated by the cloud |

TABLE I
NOTATIONS USED IN FORMALIZING RESOURCE ALLOCATION

We model the system as evolving at discrete points in time $t = 0, 1, \dots$.

Table I summarizes the notations used in this paper.

### *B. The Optimization Problem*

For the above model, we consider a cloud with CPU capacity $\boldsymbol{\Omega}$, memory capacity $\boldsymbol{\Gamma}$, and demand vectors $\boldsymbol{\omega}, \boldsymbol{\gamma}$. We first discuss a simplified version of the problem. It consists of finding a configuration $A$ that maximizes the cloud utility $U^c$:

$$\begin{aligned} &\text{maximize} && U^c(A, \boldsymbol{\omega}) && \\ &\text{subject to} && A \geq 0, \quad \mathbf{1}^T A = \mathbf{1}^T && (a) \qquad \text{(OP(1))} \\ & && \hat{\Omega}(A, \boldsymbol{\omega})\mathbf{1} \preceq \boldsymbol{\Omega} && (b) \end{aligned}$$

Our concept of utility is max-min fairness (cf. [2]), and our goal is to achieve fairness among sites. This means that we want to maximize the minimum utility of all sites, which we achieve by maximizing the minimum utility of all module instances.

Constraint (a) of OP(1) relates to dividing into shares the CPU demand of each module into the demand of its instances. The constraint expresses that all shares are non-negative and add up to 1 for each module. Constraint (b) says that, for each machine in the cloud, the allocated CPU resources can not be larger than the available capacity. $\hat{\Omega}$ is the resource allocation function which we discuss in Section IV-A.

We now extend OP(1) to the problem that captures the cloud environment in more detail. First, we take into account the memory constraints on individual machines, which significantly increases the problem complexity. Second, we consider the fact that the system must adapt to external events described above, in order to keep the cloud utility maximized. Therefore, the problem becomes one of adapting the current configuration $A(t)$ at time $t$ to a new configuration $A(t+1)$ at time $t+1$ which achieves maximum utility at minimum cost of adapting the configuration.

$$\begin{aligned} \text{maximize} \quad & U^c(A(t+1), \boldsymbol{\omega}(t+1)) \\ \text{minimize} \quad & c^*(A(t), A(t+1)) \\ \text{subject to} \quad & A(t+1) \geq 0, \ \mathbf{1}^T A(t+1) = \mathbf{1}^T \\ & \hat{\Omega}(A(t+1), \boldsymbol{\omega}(t+1))\mathbf{1} \preceq \boldsymbol{\Omega} \\ & \mathbf{sign}(A(t+1))\boldsymbol{\gamma} \preceq \boldsymbol{\Gamma}. \end{aligned} \tag{OP(2)}$$

This optimization problem has prioritized objectives in the sense that, among all configurations $A$ that maximize the cloud utility, we select one that minimizes the cost function $c^*$. (The cost function we choose for this work gives the number of module instances that are started to reconfigure the system from the current to the new configuration.)

While this paper considers only events in form of changes in demand, OP(2) allows us to express (and solve) the problem of finding a new allocation after other events, including adding or removing sites or machines.

## IV. A Protocol for Distributed Resource Allocation

In this section, we present a protocol P, which is a heuristic algorithm for solving OP(2) and which represents our proposed protocol for resource allocation in a cloud environment.

P is a gossip protocol and has the structure of a round-based distributed algorithm (whereby round-based does not imply that the protocol is synchronous). When executing a round-based gossip protocol, each node selects a subset of other nodes to interact with, whereby the selection function is often probabilistic. Nodes interact via 'small' messages, which are processed and trigger local state changes. In this work, node interaction follows the so-called push-pull paradigm, whereby two nodes exchange state information, process this information and update their local states during a round.

P runs on all machines of the cloud. It is invoked at discrete points in time, in response to a load change. The output of the protocol, the configuration matrix $A$, is distributed across the machines of the system. $A$ controls the start and stop of module instances and determines the control policies for module schedulers and request forwarders. The protocol executes in the resource manager components of the middleware architecture (See Figure 2). A set of candidate machines to interact with is maintained by the overlay manager component of the machine manager. We assume that the time it takes for P to compute a new configuration $A$ is small compared to the time between events that trigger consecutive runs of the protocols. At the time of initialization, P reads as input a feasible configuration of the system (see below). At later invocations, the protocol reads as input the configuration matrix produced during the previous run.

### A. Functionalities the protocol P Uses

*a) Random selection of machines:* P relies on the ability of a machine to select another machine of the cloud uniformly at random. In this work, we approximate this ability by using CYCLON, an overlay protocol that produces a time-varying network graph with properties of a random network [3].

*b) Resource allocation and module scheduling policy:* In this work, machines apply a resource allocation policy $\hat{\Omega}$ that allocates CPU resources to module instances proportional to their respective demand, i.e., $\hat{\omega}_{n,m}(t) = \omega_{n,m}(t)\Omega_n/\sum_i \omega_{n,i}$. Such a policy respects the constraints in OP(1) and OP(2) regarding CPU capacity.

*c) Computing a feasible configuration:* P requires a feasible configuration as input during its initialization phase. A simple greedy algorithm can be used for this purpose, which we present in [4] due to space limitation.

### B. Protocol P': An Optimal Solution to OP(1)

We developed the protocol P', which is a distributed solution to OP(1). P' is a gossip protocol that produces a sequence of configuration matrices $A(r), r = 1, 2, \ldots$, such that the series of cloud utilities $U^c(A(r), \boldsymbol{\omega})$ converges exponentially fast to the optimal utility. Due to space limitation, P' is described and its properties proved in [4]. We would encourage the reader to look up this protocol, as it is quite simple and enables a better understanding of P, which can be seen as an extension of P'. During each round of P', two machines perform an equalization step whereby CPU demand is moved from one machine to another machine in such a way that their *relative demands* are equalized. The relative demand of a machine $n$ is defined as $v_n = \sum_m \omega_{n,m}/\Omega_n$.

## *C. Protocol P: A Heuristic Solution to OP(2)*

OP(2) differs from OP(1) in that memory constraints of individual machines are considered and a secondary objective is added for the purpose of minimizing the cost of adapting the system from the current to a new configuration that maximizes the utility for the new demand. Introducing local memory constraints to the optimization problem turns OP(1), which we showed can be efficiently solved for many practical cases [4], into an NP-hard problem [2].

P employs the same basic mechanism as P' as it attempts to equalize the relative demands of pairs of machines during a protocol round. Due to the local memory constraints, such a step does not always succeed.

P uses the following approach to achieve its objectives. First, pairs of machines that execute an equalization step are often chosen in such a way that they run instances of common modules. To support this concept, we maintain on each machine $n$ the set $N_n$ of machines in the cloud that run module instances common with $n$. To avoid the possibility of the cloud being partitioned into disjoint sets of interacting machines, $n$ is occasionally paired with a machine outside of the set $N_n$ to execute an equalization step. This dual approach keeps low the need for starting new module instances and thus keeps the cost low. Second, during an equalization step, P attempts to reduce the difference in relative demand between two machines, in case it cannot equalize the demand. Further, P attempts to execute an equalization step in such a way that the demand for a specific module is shifted to one machine only. This concept aims at increasing the probability that an equalization step succeeds in equalizing the relative demands, thus increasing the cloud utility.

The pseudocode of P is given in Algorithm 1. To keep the presentation simple, we omit thread synchronization primitives which prevent concurrent machine to machine interactions. Note that setting $\alpha_{n,m} = 0$ implies stopping module $m$ on machine $n$.

During the initialization of machine $n$, the algorithm reads the CPU demand vector, the CPU and memory capacity vectors, and the row of the configuration matrix for $n$. (For an efficient implementation, $n$ must only read those vector components that refer to itself and its module instances.) Then, it starts two threads: an active thread, in which the machine periodically executes a round, and a passive thread that waits for another machine to start an interaction.

**Algorithm 1** Protocol P computes a heuristic solution for OP(2) and returns a configuration matrix $A$. Code for node $n$.

```
initialization
read $\boldsymbol{\omega}, \boldsymbol{\Omega}, \boldsymbol{\Gamma}, row_n(A), N_n$;
start the passive and active threads
active thread
for $r = 1$ to $r_{max}$ do
  if $rand(0..1) < p$ then
    choose $n'$ at random from $N_n$;
  else
    choose $n'$ at random from $N - N_n$;
  send$(n', row_n(A))$;
  $row_{n'}(A)$ =receive($n'$);
  equalizeWith($n', row_{n'}(A)$);
  sleep(roundDuration);
write $row_n(A)$;
passive thread
while true do
  $row_{n'}(A)$=receive($n'$);
  send($n', row_n(A)$);
  equalizeWith($n', row_{n'}(A)$);

proc equalizeWith($j, row_j(A)$)
$l = \arg\max\{v_n, v_j\}$;  $l' = \arg\min\{v_n, v_j\}$;
if $j \in N_n$ then
  moveDemand1($l, row_l(A), l', row_{l'}(A)$);
else
  moveDemand2($l, row_l(A), l', row_{l'}(A)$);

proc moveDemand1($l, row_l(A), l', row_{l'}(A)$)
compute $\Delta\omega$ such that
  $\frac{1}{\Omega_l}(\sum_m \omega_{l,m} - \Delta\omega) = \frac{1}{\Omega_{l'}}(\sum_m \omega_{l',m} + \Delta\omega)$
let $mod$ be an array of all modules
  that run on both $l$ and $l'$, sorted by
  increasing $\omega_{l,m}$
for $i = 1$ to $|mod|$ do
  $m = mod[i]$;  $\delta\omega = \min(\Delta\omega, \omega_{l,m})$;
  $\Delta\omega$ $-= \delta\omega$;  $\delta\alpha = \alpha_{l,m}\frac{\delta\omega}{\omega_{l,m}}$;
  $\alpha_{l',m}$ $+= \delta\alpha$;  $\alpha_{l,m}$ $-= \delta\alpha$;

proc moveDemand2($l, row_l(A), l', row_{l'}(A)$)
compute $\Delta\omega$ such that
  $\frac{1}{\Omega_l}(\sum_m \omega_{l,m} - \Delta\omega) = \frac{1}{\Omega_{l'}}(\sum_m \omega_{l',m} + \Delta\omega)$
let $mod$ be an array of all modules
  that run on $l$, sorted by decreasing
  $\frac{\omega_{l,m}}{\gamma_m}$;
for $i = 1$ to $|mod|$ do
  $m = mod[i]$;  $\delta\omega = \min(\Delta\omega, \omega_{l,m})$;
  if $\gamma_m + \sum_{i|\alpha_{l',i}>0} \gamma_i \le \Gamma_{l'}$ then
    $\Delta\omega$ $-= \delta\omega$;  $\delta\alpha = \alpha_{l,m}\frac{\delta\omega}{\omega_{l,m}}$;
    $\alpha_{l',m}$ $+= \delta\alpha$;  $\alpha_{l,m}$ $-= \delta\alpha$;
```

The active thread executes $r_{max}$ rounds. In each round, $n$ chooses a machine $n'$ uniformly at random from the set $N_n$ with probability $p$ and from the set $N - N_n$ with probability $1 - p$. (For the evaluation of P in Section V, we set $p = \frac{|N_n|}{1+|N_n|}$.) Then, $n$ sends its state (i.e., $row_n(A)$) to $n'$ , receives $n'$'s state as a response, and calls the procedure `equalizeWith()`, which performs the equalization step. The passive thread executes in a continuous loop. Whenever $n$ receives the state from another machine $n'$, it responds by sending its own state to $n'$ and performing an equalization step by invoking `equalizeWith()`.

The procedure `equalizeWith()` attempts to equalize the relative demands of machines $n$ and $n'$. It first identifies the machine $l$ with the larger (or equal) relative demand and the machine $l'$ with the lower relative demand. Then, if $n'$ belongs to $N_n$ and thus runs at least one common module instance, procedure `moveDemand1()` is invoked. Otherwise `moveDemand2()` is invoked.

`moveDemand1()` equalizes (or reduces the difference) of the relative demands of the two machines, by shifting demand from the machine $l$ with the larger relative demand to the machine $l'$ with the smaller relative demand. It starts by computing the demand $\Delta\omega$ that needs to be shifted from $l$ to $l'$ (step 1). Then, from the set of modules that run on both machines, taking an instance with the smallest demand on $l$, it proceeds to shift the demand from $l$ to $l'$, until a total of $\Delta\omega$ demand is shifted, or it has exhausted the set of modules.

`moveDemand2()` equalizes (or reduces the difference) of the relative demands of the two machines, by moving demand from the machine with larger relative demand to the machine with smaller relative demand. Unlike `moveDemand1()`, `moveDemand2()` starts one or more module instances at the destination machine, to move demand from the source machine to the destination, if sufficient memory at the destination machine is available. Finding a set of instances at the source that equalize the relative demands of the participating machines while observing the available memory of the destination is a Knapsack problem. A greedy approximation method is applied, whereby the module $m$ with the largest value of $\frac{\omega_{l,m}}{\gamma_m}$ is moved first, followed by the second largest, etc., until the relative demands are equalized or the set of candidate modules is exhausted [5].

## V. Evaluation through Simulation

We have evaluated P through extensive simulations using a discrete event simulator that we developed inhouse. We simulate a distributed system that runs the machine manager components of all machines in the cloud. Specifically, these machine managers execute the protocol P, which computes the allocation matrix $A$, and also the CYCLON protocol, which provides for P the function of selecting a random neighbor. The external events for this simulation are the changes in demand vector $\boldsymbol{\omega}$.

*Evaluation metrics:* We evaluate the protocol in various scenarios and measure the following metrics. We express the *fairness* of resource allocation through the coefficient of variation of site utilities, computed as the ratio of the standard deviation divided by the average of the utilities. We measure the *satisfied demand* as the fraction of sites that generate a utility larger than 1. We measure the *cost of reconfiguration* as the ratio of module instances started to module instances running, per machine.

*Generating the demand vectors $\boldsymbol{\omega}$ and $\boldsymbol{\gamma}$:* In all scenarios, the number of modules of a site is chosen from a discrete Poisson distribution with mean 1, incremented by 1. The memory demand of a module is chosen uniformly at random from the set $c_\gamma\cdot${128MB, 256MB, 512MB, 1GB, 2GB}. For a site $s$, at each change in demand, the demand profiler generates CPU demands chosen from an exponential distribution with mean $\omega(s)$. We choose the distribution for $\omega(s)$ among all sites to be Zipf distributed with $\alpha = 0.7$, following evidence in [6]. The maximum value for the distribution is $c_\omega\cdot$500G CPU units and the population size used is 20,000. For a module $m$ of site $s$, we choose a demand factor $\beta_m$ with $\sum_{m\in M_s}\beta_m = 1$, chosen uniformly at random, which describes the share of module $m$ in the demand of the site $s$. $c_\gamma$ and $c_\omega$ are scaling factors, see below.

*Capacity of machines:* A machine has CPU capacity selected uniformly at random from {2, 4, 8, 16}G CPU units and a memory capacity selected uniformly at random from {2, 4, 8, 16}GB (CPU and memory capacities are chosen independently of each other).

*Choosing machines for interaction:* Each machine runs CYCLON, a gossip protocol that implements random neighbor selection.

*Scenario parameters:* We evaluate the performance of our resource allocation protocol P under varying intensities of CPU and memory load which are defined as follows. The CPU load intensity is measured by the *CPU load factor (CLF)*, which is the ratio of the total CPU demand of sites to the total CPU capacity of machines in the cloud. Similarly, the *memory load factor(MLF)* is the ratio of the total memory demand of sites (assuming each module runs only one instance) to the total memory capacity of all machines in the cloud. In the simulations, we vary CLF and MLF by changing $c_\gamma$ and $c_\omega$. In the reported experiments, we use the following parameters unless stated otherwise:

- $|N|$=10,000, $|S|$=24,000
- $r_{max} = 30$, $CLF = MLF = 0.5$
- maximum number of instances/module: 100
- number of load changes during a run: 100

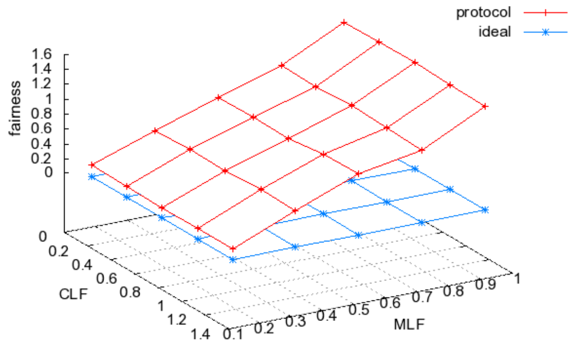
## A. *Fairness*

First, we evaluate P regarding the fairness of resource allocation for CLF=$\{0.1, 0.4, 0.7, 1.0, 1.3\}$ and MLF=$\{0.15, 0.35, 0.55, 0.75, 0.95\}$. (We leave out MLF=1 because there may not exist a feasible solution or our initialization algorithm may not find it.) We compare the performance of our protocol to that of an *ideal system*, which can be thought of as a centralized system which performs resource allocation on a giant single machine that has the aggregate CPU and aggregate memory capacities of the entire cloud. The performance of this centralized system gives us a bound on the performance P.

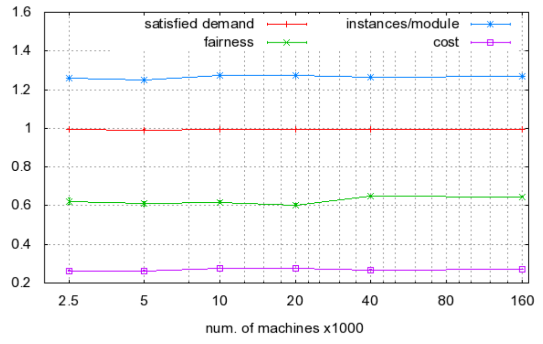
Figure 3a shows that the measured fairness metric seems to be independent of CLF. (Note that the value of this metric that corresponds to our fairness goal is 0.) We expect this because P allocates CPU resources proportional to the demand of a module instance, regardless of the available capacity on the particular machine. Second, the figure shows that resource allocation becomes less fair when MLF is increased. For example, the average deviation of the utilities of sites is about 16% from the average utility for MLF of 15%. However, this value increases to about 100% when MLF is at 75%. This behavior is also to be expected, since increasing MLF results in machines being less likely to have suffcient memory for starting new instances. Note that the ideal system always achieves our fairness goal since its memory is not fragmented.

## B. *Scalability*

In this scenario, we measure the dependence of our evaluation metrics on the size of the cloud. To achieve this, we run simulations for a cloud with (2,500, 5,000, 10,000, 20,000, 40,000, 160,000) machines and (6,000, 12,000, 24,000, 48,000, 96,000, 384,000) sites respectively (keeping the ratio of sites to machines at 2.4, which ensures that CLF and MLF are kept close to the default value of 0.5). Figure 3b shows the result obtained, which indicates that all metrics considered for this evaluation are independent of the system size. In other words, if the number of machines grows at the same rate as the number of sites, (while the CPU and memory capacities of a machine, as well as all parameters characterizing a site, such as demand, number of modules, etc., stay the same) then we expect all considered metrics to remain constant. Note that our conclusion is related exclusively to the scalability of the protocol P. The complete resource management system includes many more functions that have not been evaluated here, for instance, the scalability of effectively choosing a random peer.



(a) Fairness of the resource allocation in function of the CPU load factor (CLF) and the memory load factor (MLF) of the cloud. A value of 0 achieves the fairness objective of the cloud.



(b) Scalability with respect to the number of machines and sites.

Fig. 3. The performance of the resource allocation protocol P with respect to fairness and scalability.

## C. *Satisfied demand and cost of reconfiguration*

We evaluated P with regards to satisfied demand and cost of reconfiguration for different values of CLF and MLF. Due to space limitation, the details of the evaluation and the results are presented in [4]. With regards to satisfied demand, the results show that our protocol satisfies more than 95% of all site demands for CLF$\leq$ 70% and MLF$\leq$ 55%. With regards to cost of reconfiguration, the result shows that the cost increases with decreasing MLF. For instance, the cost of reconfiguration is less than 3% for MLF of 95%. However, it increases to 35% for MLF of 15%.

## VI. Related work

The problem of application placement in the context of resource management for datacenters has been studied before (e.g., [2], [7]), and solutions are already available in middleware products [8]. While these product solutions allow for a fair resource allocation in a similar way as our scheme does, they rely on centralized architectures, which do not at all scale to system sizes we consider in this paper.

The work in [9], which has been extended by [10] presents a distributed middleware layer for application placement in datacenters. As in this paper, the goal of that work is to maximize a cluster utility under changing demand, although a different concept of utility is used. The proposed design in [9], [10] scales with the number of machines, but it does not scale in the number of applications, as the design in this paper does. (The concept of an application in the referenced work roughly corresponds to concept of a site in this paper.)

Distributed load balancing algorithms have been extensively studied for homogeneous as well as heterogeneous systems, for both divisible and indivisible demands. These algorithms typically fall into two classes: *diffusion* algorithms (e.g., [11]) and *dimension exchange* algorithms (e.g., [12]). Convergence results for different network topologies and different norms (that measure the distance between the system state and the optimal state) have been reported, and it seems to us that the problem is well understood today. The key difference to the problem addressed in this paper is that these algorithms do not take into account memory constraints. Considering memory constraints makes the problem NP-hard and does require a new approach.

## VII. Discussion and conclusion

With this paper, we make a significant contribution towards engineering a resource management middleware for a site-hosting cloud environment. We identify a key component of such a middleware and present a protocol that can be used to meet our design goals for resource management: fairness of resource allocation with respect to sites, efficient adaptation to load changes and scalability of the middleware layer in terms of both the number of machines in the cloud as well as the number of hosted sites.

We presented a gossip protocol that computes a heuristic solution to the resource allocation problem and evaluated its performance through simulation. In all the scenarios we investigated, we observe that the protocol qualitatively behaves as expected based on its design. For instance, regarding fairness, the protocol performs close to an ideal system for scenarios where the ratio of the total memory capacity to the total memory demand is large. More importantly, the simulations suggest that the protocol is scalable in the sense that all investigated metrics do not change when the system size (i.e., the number of machines) increases proportional to the external load (i.e., the number of sites). Note that if we would solve the resource allocation problem expressed in OP(2) by P in a centralized system, then the CPU and memory demand for that resource allocation system would increase linearly with the system size. This strongly suggests to us that a centralized solution for the problem we address in this paper will not be feasible.

The results reported in this paper are building blocks towards engineering a resource management solution for large-scale clouds. Pursuing this goal, we plan to address the following issues in future work: (1) Develop a distributed mechanism that efficiently places new sites. (2) Extend the middleware design to become robust to machine failures. (3) Extend the middleware design to span several clusters and several datacenters, while keeping module instances of the same site "close to each other", in order to minimize response times and communication overhead.

## References

[1] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "Dynamic estimation of CPU demand of web traffic," in *valuetools '06: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*. New York, NY, USA: ACM, 2006, p. 26.

[2] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguade, "Utility-based placement of dynamic web applications with fairness goals," in *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, april 2008, pp. 9 –16.

[3] S. Voulgaris, D. Gavidia, and M. van Steen, "CYCLON: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[4] F. Wuhib, R. Stadler, and M. Spreitzer, "Gossip-based resource management for cloud environments (long version)," KTH Royal Institute of Technology, Tech. Rep., August 2010, TRITA-EE 2010:032.

[5] G. B. Dantzig, "Discrete-Variable Extremum Problems," *OPERATIONS RESEARCH*, vol. 5, no. 2, pp. 266–288, 1957.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: evidence and implications," in *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, 21-25 1999, pp. 126 –134 vol.1.

[7] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *WWW '07: Proceedings of the 16th international conference on World Wide Web*. New York, NY, USA: ACM, 2007, pp. 331–340.

[8] IBM, "IBM WebSphere Application Server." [Online]. Available: http://www.ibm.com/software/webservers/appserv/extend/virtualenterprise/

[9] C. Adam and R. Stadler, "Service middleware for self-managing large-scale systems," *Network and Service Management, IEEE Transactions on*, vol. 4, no. 3, pp. 50–64, April 2008.

[10] J. Famaey, W. De Cock, T. Wauters, F. De Turck, B. Dhoedt, and P. Demeester, "A latency-aware algorithm for dynamic service placement in large-scale overlays," in *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 414–421.

[11] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279 – 301, 1989.

[12] C. Z. Xu and F. C. M. Lau, "Analysis of the generalized dimension exchange method for dynamic load balancing," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 385–393, 1992.