A Gossip Protocol for Dynamic Resource Management in Large Cloud Environments

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

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

***Abstract*—We address the problem of dynamic resource management for a large-scale cloud environment. Our contribution includes outlining a distributed middleware architecture and presenting one of its key elements: a gossip protocol that (1) ensures fair resource allocation among sites/applications, (2) dynamically adapts the allocation to load changes and (3) scales both in the number of physical machines and sites/applications. We formalize the resource allocation problem as that of dynamically maximizing the cloud utility under CPU and memory constraints. We first present a protocol that computes an optimal solution without considering memory constraints and prove correctness and convergence properties. Then, we extend that protocol to provide an efficient heuristic solution for the complete problem, which includes minimizing the cost for adapting an allocation. The protocol continuously executes on dynamic, local input and does not require global synchronization, as other proposed gossip protocols do. We evaluate the heuristic 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. While our contribution is relevant in a more general context, we conduct the discussion from the perspective of the Platform-as-a-Service (PaaS) concept, with the specific use case of a cloud service provider which hosts sites in a cloud environment. The stakeholders for this use case are depicted in figure 1a. The cloud service provider owns and administers 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. Our contribution can also be applied (with slight modifications) to the Infrastructure-as-a-Service (IaaS) concept. A use case for this concept could include a cloud tenant running a collection of virtual appliances that are hosted on the cloud infrastructure, with services provided to end users through the public Internet. For both perspectives, this paper introduces a resource allocation protocol that dynamically places site modules (or virtual machines, respectively) on servers within the cloud, following global management objectives.

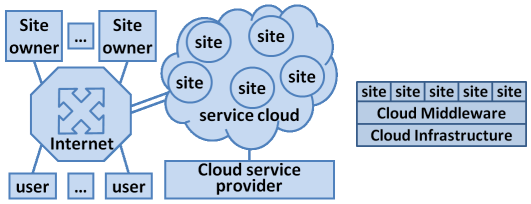


Fig. 1. (a) Deployment scenario with the stakeholders of the cloud environment considered in this work. (b) Overall architecture of the cloud environment; this work focuses on resource management performed by the middleware layer.

Technically speaking, this work contributes towards engineering a middleware layer that performs resource allocation in 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 among sites for computational resources under memory constraints. Under this objective, each site receives CPU resources

proportional to its CPU demand. (For applying our approach to minimize energy consumption, see [1].)

2) Adaptability: The resource allocation process must dynamically and efficiently adapt to changes in the demand from sites.
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 (by the process) 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.

How do the concepts presented in this paper relate to available management software for private clouds including (1) IaaS solutions, such as OpenNebula [2], OpenStack [3] and Eucalyptus [4], or (2) PaaS solutions, such as AppScale [5], WebSphere XD [6] and Cloud Foundry [7]? These solutions include functions that compute placements of applications or virtual machines onto specific physical machines. However, they do not, in a combined and integrated form, (a) dynamically adapt existing placements in response to a change (in demand, capacity, etc.), (b) dynamically scale resources for an application beyond a single physical machine, (c) scale beyond some thousand physical machines (due to their centralized underlying architecture). These three features in integrated form characterize our contribution. The concepts in this paper thus outline a way to improve placement functions in these solutions. Regarding public clouds, little information is available with respect to the underlying capabilities of services provided by companies like Amazon [8], Google [9] and Microsoft [10]. We expect that the road maps for their platforms include design goals for application placement which are similar to ours.

The core contribution of the paper is a gossip protocol P*, which executes in a middleware platform and meets the design goals outlined above. The protocol has two innovative characteristics. First, while gossip protocols for load balancing in distributed systems have been studied before, (to our knowledge) no results are available for cases that consider memory constraints and the cost of reconfiguration, which makes the resource allocation problem hard to solve (memory constraints alone make it NP-hard). In this paper, we give an optimal solution for a simplified version of the resource allocation problem and an efficient heuristic for the hard problem. Second, the protocol we propose continuously executes, while its input—and consequently its output—dynamically changes. Most gossip protocols that have been proposed to date are used in a different way. They assume static input and produce a single output value (e.g., [11]–[13]). Whenever the input changes, they are restarted and produce a new output value, which requires global synchronization. The benefit of a single, continuous execution vs. a sequence of executions with restarts is that global synchronization can be avoided and that the system can continuously adapt to changes in local input. On the other hand, its drawback is that the behavior of a protocol with dynamic input is more difficult to analyze. Also, the cost of the system to react to a high rate of change in local output can potentially be higher than implementing a set of changes after each synchronized run. Based on our work thus far, we believe that, for a gossip protocol running in a large-scale dynamic environments, the advantages of continuous execution with dynamic input outweigh its potential drawbacks [14], [15].

This paper is a significant extension of earlier work reported in [13]. In addition to numerous minor improvements in presentation and extensions to most sections, it contains the proof for the properties of protocol P', which solves a simplified resource allocation problem (introduced in [13]). Most importantly, it contains a description and evaluation of protocol P*, which continuously executes and dynamically solves the problem of optimally placing applications in a cloud, achieving fair resource allocation. P* can be understood as a "continuous" version of the protocol P described in [13].

In this paper, we restrict ourselves to a cloud that spans a single datacenter containing a single cluster of machines. Further, we treat all machines as equivalent in the sense that we do not take into account that they may belong to specific racks, clusters, or computing pods. An extension of our contribution to a more heterogeneous environment is planned for future work.

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 analyzed analytically, the other evaluated through simulation in Section VI. Section VII reviews related work, and Section VIII contains the conclusion of this research and outlines future work.

II. System Architecture

Datacenters running a cloud environment often contain 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 the datacenter that either processes the request or forwards it.

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).

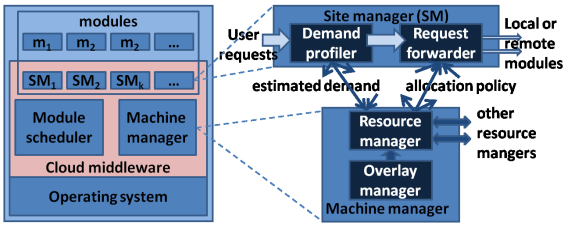


Fig. 2. The architecture for the cloud middleware (left) and components for request handling and resource allocation (right).

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 estimated 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.

Our architecture associates one site manager with each site. A site manager handles user requests to a particular site. It has two components: a *demand profiler* and a *request forwarder*. The *demand profiler* estimates the resource demand of each module of the site based on request statistics, QoS targets, etc. (Examples of such a profiler can be found in [16], [17].) This demand 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 above architecture is not appropriate for the case where a single site manager can not handle the incoming request stream for a site. However, a scheme for a site manager to scale can be envisioned. For instance, a layer 4/7 switch could be introduced that splits the load among several instances of site managers, whereby each such instance would function like a site manager associated with a single site.

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 the request forwarder components. At discrete points in time, events occur, such as demand 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 $\boldsymbol{\omega}(t) = [\omega_1(t), \omega_2(t), \ldots, \omega_{|M|}(t)]^T$ and the memory demand as the vector $\boldsymbol{\gamma} = [\gamma_1, \gamma_2, \ldots, \gamma_{|M|}]^T$, assuming that CPU demand is time dependent while memory demand is not [18].

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. (For the purpose of architectural simplicity we allow a single instance for each site manager module.) 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 Ω_n and a memory capacity Γ_n. We use $\boldsymbol{\Omega}$ and $\boldsymbol{\Gamma}$ to denote the vectors of CPU and memory capacities of all machines in the system. An instance of module m running on machine n demands $\omega_{n,m}(t)$ CPU resource and γ_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 γ_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 ∞.) 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.

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

Table I summarizes the notations used in this paper.

Notations

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

Formulas

$\sum_n \alpha_{n,m} = 1$ $\alpha_{n,m}(t) \geq 0, \forall n, m$	properties of $\alpha_{n,m}$
$\omega_{n,m}(t) = \alpha_{n,m}(t)\omega_n(t)$	resource demand of an instance of module m on machine n
$u_{n,m}(t) = \frac{\hat{\omega}_{n,m}(t)}{\omega_{n,m}(t)}$	utility of an instance of a module
$U^c(t) = \min_{n,m} u_{n,m}(t)$	utility of the cloud
$\hat{\omega}_{n,m}(t) = \frac{\omega_{n,m}(t)}{\sum_i \omega_{n,i}}\Omega_n$	local resource allocation policy (Section IV-A)
$v_n(t) = \frac{\sum_m \omega_{n,m}(t)}{\Omega_n}$	relative CPU demand of a machine n (Section IV-B)

TABLE I
NOTATIONS AND FORMULAS FOR RESOURCE ALLOCATION

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:

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

Our concept of utility is max-min fairness (cf. [18]), 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. Given the definition of the utility of a module instance and the local CPU allocation policy (see Table I), this results in allocating the CPU resources of the cloud to the module instances proportional to their CPU demands.

Constraint (a) of OP(1) says that each module's CPU demand is divided among its instances. 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 policy 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.

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

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 need to be 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 our protocol for resource allocation in a cloud environment, which we call P*. It is based on a heuristic algorithm for solving OP(2) and is implemented in form of a gossip protocol.

As a *gossip protocol*, P* 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. Node interaction with P* follows the so-called push-pull paradigm, whereby two nodes exchange state information, process this information and update their local states during a round. Compared to alternative distributed solutions, gossip-based protocols tend to be simpler, more scalable and more robust.

P* runs on all machines of the cloud. More precisely, it executes in the resource manager components of the middleware architecture (See Figure 2). At the time of initialization, the resource manager implements a feasible cloud configuration A (see below for a possible algorithm). After that, it invokes P* to compute and dynamically adapt the configuration with the goal to optimize the cloud utility (which means achieving max-min fairness among sites in our case). Whenever the protocol has computed a new configuration, encoded in the matrix A, the resource manager checks whether the gain in utility of the newly computed configuration over the currently implemented configuration outweighs the cost of realizing the change. If this is the case, then the resource manager implements the new configuration encoded in A.

The protocol P* takes as input the available cloud resources, the current configuration A and the current resource demand. It further relies on a set of candidate machines to interact with a given machine. This set is produced and maintained by the overlay manager component of the machine manager (See Figure 2).

Note that P* is distributed. The protocol follows the concept that each machine has only partial view of the cloud at any point in time. A machine is aware of the load of the modules instances it runs, and it maintains a row of the configuration matrix A that relates to the allocation of its own resources. As a consequence, the configuration matrix A is distributed across the machines of the cloud.

P* is designed to run continuously in an asynchronous environment where a machine does not synchronize the start time of a protocol round with any other machine. Further, a machine coordinates an update of the configuration A only with one additional machine at a time, namely its current interaction partner in the gossip protocol. Therefore, during the evolution of the system, the implemented cloud configuration A changes dynamically and asynchronously, as each machine maintains its part of the configuration. (Note that, to be precise, we use A in this section in two ways: (1) as the output of the protocol P* and (2) as the implemented cloud configuration.)

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 through CYCLON, an overlay protocol that

Algorithm 1 Initialize P' or P* with a feasible configuration.

let M be an array of all modules sorted by γ_m in decreasing order
let N be the set of all machines
let function freeMemNode(N) return a machine $n \in N$ with the largest free memory, namely, $\Gamma_n - \sum_m \gamma_m \mathbf{sign}(\alpha_{m,n})$
$\alpha_{n,m} = 0, \ \forall n, m$
for $i = 1$ to $|M|$ **do**
  $m = M[i]$; $n =$ freeMemNode(N)
  if $\Gamma_n - \sum_{i \in M} \gamma_i \mathbf{sign}(\alpha_{n,i}) \geq \gamma_m$ **then**
    $\alpha_{n,m} = 1$

produces a time-varying network graph with properties of a random network [19].

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., $\hat{\omega}_{n,m}(t) = (\omega_{n,m}(t)/\sum_i \omega_{n,i})\Omega_n$. Such a policy respects the constraints in OP(1) and OP(2) regarding CPU capacity, as $\sum_m \hat{\omega}_{n,m}(t) = \Omega_n$.

c) Computing a feasible configuration: The (centralized) Algorithm 1 produces a feasible solution A to the optimization problems OP(1) and OP(2) for a demand vector $\boldsymbol{\omega}$, assuming the above scheduling policy $\hat{\Omega}$. For the uniform case where $\Gamma_i = \Gamma_j, \forall i, j$, the algorithm is guaranteed to find a feasible configuration, provided that there exists some configuration that uses at most $\frac{3}{4}$ of the machines [20]. The algorithm can be used for initializing P' or P* during the boostrap phase of the resource management system. Later executions of P' and P* use for initialization the value of A produced by the previous run of the respective protocol. An alternative way of computing a feasible configuration is given in [21].

B. The Protocol and its Analysis

In this subsection, we present P', a simplified version of P* that ignores the memory constraints and the cost of change in configuration. It assumes the ability of a machine to choose uniformly at random another machine for interaction. P' approximates P* well in cases where the memory demand is significantly smaller than the available memory capacity and the cost of change is small. P' can be analyzed with respect to correctness and convergence, whereas a similar analysis of P* seems to be very hard given the current state-of-the-art.

P' (and also P*) is based on a gossip protocol proposed by Jelasity et al. that computes the global average of local node variables [11]. In each round of that protocol, a node averages its local value with that of another node selected uniformly at random. The authors show that each local value converges exponentially fast to the global average.

The key difference between P' and the protocol in [11] lies in the definition of the local states and the way these states are updated when two machines interact. While in [11] states are updated with the arithmetic average of two local state variables, the update procedure in P' is more complex. During the interaction of two machines, P' equalizes the relative demand of both machines, whereby the relative demand v_n of a machine n is defined as $v_n = \sum_m \omega_{n,m}/\Omega_n$. Such an equalization step involves moving demand from the machine with the larger relative demand to the machine with the lower relative demand. Over time, P' equalizes the relative demand across all machines in the cloud.

P' is executed in response to a change in demand $\boldsymbol{\omega}$. For easier understanding, consider a synchronous execution model for this protocol, whereby all rounds are globally synchronized. (Theorem 1 holds also for an asynchronous execution model.) We give P' in a simple form with no limit of the number of executed rounds. In a real system, one would stop the protocol when the output (i.e., the configuration matrix A) is sufficiently close to an optimal solution to OP(1).

The pseudocode of P' is given in Algorithm 2. $row_n(A)$ denotes the n^{th} row of the configuration matrix A and corresponds to the configuration of machine n.

During the initialization of machine n, the algorithm reads the CPU capacity of the machine, the row of the configuration matrix for n (i.e., $row_n(A)$), as well as the demand of the modules that run on n (i.e., $\boldsymbol{\omega}_n$). Then, it starts two threads: an active thread that periodically executes a round and a passive thread that waits for another machine to initiate an interaction.

In the active thread, each round starts with n choosing another machine n' uniformly at random from N, the set of machines in the cloud. Then, n sends its state (i.e., $row_n(A)$) to n', receives n''s state as a response and calls the procedure `equalize()`, which equalizes its own relative demand with that of n'. In the passive thread, whenever n receives the state from another machine n', it responds by sending its own state to n' and by invoking `equalize()`.

The procedure `equalize()` equalizes the relative demand of two machines. First, it identifies the machine l with the larger relative demand and the machine l' with the lower relative demand. Then, it computes the

Algorithm 2 ***Protocol P'*** computes an optimal solution to OP(1) and returns configuration matrix A. Code for node n.

initialization

1: read $\boldsymbol{\omega}_n, row_n(A), \Omega_n$;
2: start the passive and active threads

active thread

3: **while** true **do**
4: choose n' uniformly at random from N;
5: send $row_n(A), \Omega_n$ to n';
6: receive $row_{n'}(A), \Omega_{n'}$ from n';
7: equalize($n', row_{n'}(A), \Omega_{n'}$);
8: write $row_n(A)$;

passive thread

1: **while** true **do**
2: receive $row_{n'}(A), \Omega_{n'}$ from n';
3: send $row_n(A), \Omega_n$ to n';
4: equalize($n', row_{n'}(A), \Omega_{n'}$);
5: write $row_n(A)$;

proc equalize($j, row_j(A), \Omega_j$)

6: $l = \arg\max\{v_n, v_j\}$; $l' = \arg\min\{v_n, v_j\}$;
7: compute $\Delta\omega$ such that $\frac{1}{\Omega_l}(\sum_m \omega_{m,l} - \Delta\omega) = \frac{1}{\Omega_{l'}}(\sum_m \omega_{m,l'} + \Delta\omega)$
8: choose set of modules s and module $m \notin s$ running on l such that $\sum_{i \in s} \omega_{i,l} < \Delta\omega$ and $\sum_{i \in s} \omega_{i,l} + \omega_{m,l} \geq \Delta\omega$
9: **for** $i \in s$ **do**
10: $\Delta\omega -= \omega_{i,l}$; $\alpha_{i,l'} += \alpha_{i,l}$; $\alpha_{i,l} = 0$;
11: $\alpha_{m,l'} += \alpha_{m,l}\frac{\Delta\omega}{\omega_{m,l}}$; $\alpha_{m,l} -= \alpha_{m,l}\frac{\Delta\omega}{\omega_{m,l}}$;

demand $\Delta\omega$ to be moved from l to l'. In step 8, a set s of module instances on l is identified whose demand is completely moved from l to l' and also an instance m on l is chosen whose demand is partially moved from l to l'. In the remaining steps 9-11, the $row_n(A)$ is updated to reflect the shift in demand. Note that in this procedure, both interacting machines must choose the same set s and module m. This can be achieved, for instance, by having global identifiers for all modules and using the sorted list of modules as a prioritized candidate list for selecting the modules.

The code of Algorithm 2 requires synchronization primitives that control access to the node state for both threads, in case a machine engages, at the same time, in interactions with several machines. To keep the presentation simple, such primitives have been omitted.

The following theorem states that P' produces an optimal solution for OP(1) by computing a sequence of approximate solutions that converge towards an optimal one. Since P' is probabilistic (with respect to neighbor selection) the statement below is probabilistic.

Theorem 1. *Assume a cloud with CPU resources $\boldsymbol{\Omega}$, our CPU allocation policy $\hat{\Omega}$, CPU demand vector $\boldsymbol{\omega}$ and a configuration $A(0)$ representing a feasible solution for OP(1). Then, executing the protocol P' on the machines of the cloud produces a sequence of configurations $\{A(r)\}_r, r \geq 1$, such that $\{U^c(A(r), \boldsymbol{\omega})\}_r$ converges in expectation to the limit $U^c_{max} = \sum_n \Omega_n / \sum_m \omega_m$, which is the utility of an optimal solution for OP(1). The convergence is exponential, at the rate of $\frac{1}{\sqrt{2\sqrt{e}}}$.*

This means that, for a given $\epsilon > 0$ and $\delta > 0$, with probability at least $1 - \delta$, there exists a round r^* such that $|U^c(A(r), \boldsymbol{\omega}) - U^c_{max}| < \epsilon, \forall r > r^*$. We prove the theorem for a cloud of homogeneous machines, i.e., where $\Omega_i = \Omega_j, \forall i, j$. We believe that the theorem holds for the inhomogeneous case as well, which we will address in future work.

Lemma 1. *Assume a cloud as in Theorem 1 with our allocation policy $\hat{\Omega}$. Then, a feasible configuration A that equalizes the relative demands among all machines in the cloud is an optimal solution for OP(1), and the utility generated by the cloud is U^c_{max}.*

First, we show that the cloud utility can not exceed U^c_{max}. Second, we prove that a configuration that equalizes the relative demand among machines results in a cloud utility of U^c_{max}.

Proof: We prove the first step by contradiction. Let A be a feasible configuration with cloud utility $U^c = U(A, \boldsymbol{\omega}) > U^c_{max}$ under $\hat{\Omega}$. From the definition of the cloud utility, $U^c = \min_{n,m}\{u_{n,m}\}$, it follows that $u_{n,m} > U^c_{max}, \forall n, m$. Using the definition of the utility of a module instance, $u_{n,m} = \frac{\hat{\omega}_{n,m}}{\omega_{n,m}}$, it follows that $\hat{\omega}_{n,m} > \omega_{n,m} U^c_{max}, \forall n, m$. Summing over all n, m, we get $\sum_{n,m} \hat{\omega}_{n,m} > U^c_{max} \sum_{n,m} \omega_{n,m}$. The left side of this inequality is equal to $\sum_n \Omega_n$ as a consequence of our allocation policy. Using the definition of U^c_{max}, the right side of the inequality equals $\frac{\sum_n \Omega_n}{\sum_m \omega_m} \sum_{n,m} \omega_{n,m}$ which is equal to $\sum_n \Omega_n$, due to the definition of A. This gives $\sum_n \Omega_n > \sum_n \Omega_n$, which is a contradiction.

To prove the second step, let A be a feasible configuration such that the relative demand v_i on all machines i is the same, i.e., $v_i = v_n, \forall i, n$. Following the definition of relative demand, $\frac{\sum_m \omega_{i,m}}{\Omega_i} = \frac{\sum_m \omega_{n,m}}{\Omega_n}$. Swapping Ω_n with $\sum_m \omega_{i,m}$ and summing up over all n, we have $\frac{\sum_n \Omega_n}{\Omega_i} = \frac{\sum_n \sum_m \omega_{n,m}}{\sum_m \omega_{i,m}}$. The right side is equal to $\frac{\sum_m \omega_m}{\sum_m \omega_{i,m}}$, due to the properties of A. From this, we conclude $\frac{\sum_m \omega_{i,m}}{\Omega_i} = v_i = \frac{\sum_m \omega_m}{\sum_n \Omega_n}, \forall i$. Due to $\hat{\Omega}$, the utility generated by a module instance m on machine n

is given by $u_{n,m} = \frac{\Omega_n}{\sum_m \omega_{n,m}} = \frac{1}{v_n}, \forall m$, from which we get $U^c = \min_{i,k}\{u_{i,k}\} = \frac{\sum_n \Omega_n}{\sum_m \omega_m} = U^c_{max}$. ∎

In order to prove Theorem 1, we show that the sequence of utilities $\{U^c(r)\}_r$ converges exponentially to the limit U^c_{max}. The first part of the proof is based on the analysis of a gossip-based averaging protocol, that has been performed by M. Jelasity et al. [11]. During each round r of that protocol, a node i selects uniformly at random a different node j from the system to interact with. During a node interaction, both nodes i and j update their respective state values s_i and s_j to $\frac{s_i(r)+s_j(r)}{2}$. As a consequence, for each node i, the values $\{s_i(r)\}_r$ converge to the global average of the initial values $\overline{\boldsymbol{x}} = \frac{1}{N}\sum_i x_i$.

The proof idea in [11] starts with the vector $(x_1, \ldots, x_N)$ of the local variables, for which the gossip protocol computes the average. The authors argue that the execution of the protocol can be modelled by the evolution of the vector of random variables $\boldsymbol{S}(r) = (S_1(r), \ldots, S_N(r))$, which is conditioned on $\sum_i S_i(r) = \sum_i x_i$, for $r = 0, 1, 2, \ldots$. The components $S_i(0)$ of the vector $\boldsymbol{S}(0)$ are modelled as independent random variables with identical expectation values and with finite variance. (Although the assumption of independence among the components $S_i(r)$ is not technically correct, the authors validate their assumption through simulations, showing that the correlation among the random variables can be ignored to investigate their convergence in sufficiently large systems.) The authors show that $\{\mathbf{E}[\sum_i (S_i(r) - \overline{\boldsymbol{x}})^2]\}_r$ converges exponentially fast to 0 for $r \to \infty$, at the rate:

$$\frac{\mathbf{E}[\sum_i (S_i(r+1) - \overline{\boldsymbol{x}})^2]}{\mathbf{E}[\sum_i (S_i(r) - \overline{\boldsymbol{x}})^2]} = \frac{1}{2\sqrt{e}} \tag{3}$$

The analysis in [11] is applicable to our protocol P' as follows. If we replace the vector of local variables $(x_1, \ldots, x_N)$ in [11] with the vector of relative demands $(v_1(0), \ldots, v_N(0))$, then, the protocol in [11] produces equivalent traces to P', in the sense that an evolution of $(s_1(r), \ldots, s_N(r))$ in [11] produces a trace that can be interpreted as an evolution of $(v_1(r), \ldots, v_N(r))$ in P', and vice versa. This is the case, because both protocols use the same gossip interaction pattern, i.e., push-pull, they employ the same strategies for neighbor selection, and they update their local state after a node interaction with the same function, i.e., average.

Following [11], we study the evolution of the vector of the relative demands $\boldsymbol{v}(r) = (v_1(r), \ldots, v_N(r))$. We do this by introducing a vector of random variables $\boldsymbol{V}(r) = (V_1(r), \ldots, V_N(r))$ conditioned on $\sum_n V_n(r) = \frac{N}{U^c_{max}}, \forall r$. The vector $\boldsymbol{V}(0)$ is an array of random variables whose values are all possible permutations of the components of $\boldsymbol{v}(0)$.

Based on the above discussion, we obtain the following lemma:

Lemma 2. *Let $\boldsymbol{v}(r)$ be the vector of relative demands of the machines in the cloud at round r. Then, the sequence of expectations of the terms $\sum_n (V_n(r) - \frac{1}{U^c_{max}})^2$ converges to 0, at the rate:*

$$\frac{\mathbf{E}[\sum_n (V_n(r+1) - \frac{1}{U^c_{max}})^2]}{\mathbf{E}[\sum_n (V_n(r) - \frac{1}{U^c_{max}})^2]} = \frac{1}{2\sqrt{e}}$$

.

We now define $\hat{V}_n(r) := V_n(r) - \frac{1}{U^c_{max}}$ and study the evolution of the vector $\hat{\boldsymbol{V}}(r) = (\hat{V}_1(r), \ldots, \hat{V}_N(r))$ with respect to r. Lemma 2 implies that the sequence $\{\mathbf{E}[\sum_n \hat{V}_n^2(r)]\}_r$ converges exponentially fast at the rate $\frac{1}{2\sqrt{e}}$.

Since, for each possible value of $\hat{\boldsymbol{V}}(r)$, $\max_n \hat{v}_n(r) \leq \sqrt{\sum_n \hat{v}_n^2(r)}$, $\forall r$, we get $\mathbf{E}[\max_n \hat{V}_n(r)] \leq \mathbf{E}[\sqrt{\sum_n \hat{V}_n^2(r)}] \leq \sqrt{\mathbf{E}[\sum_n \hat{V}_n^2(r)]}$. The last inequality follows from Jensen's inequality, as $\sqrt{\ }$ is concave. Since $0 \leq \mathbf{E}[\max_n \hat{V}_n(r)] \leq \sqrt{\mathbf{E}[\sum_n \hat{V}_n^2(r)]}, \forall r$, the sequence $\{\mathbf{E}[\max_n \hat{V}_n(r)]\}_r$ is dominated by $\{\sqrt{\mathbf{E}[\sum_n \hat{V}_n^2(r)]}\}_r$ and we conclude that $\{\mathbf{E}[\max_n \hat{V}_n(r)]\}_r$ converges (at least) exponentially fast at the rate $\sqrt{\frac{1}{2\sqrt{e}}}$.

Now, consider a value $\hat{\boldsymbol{v}}(r)$ of $\hat{\boldsymbol{V}}(r)$. We wish to compute $U^c(r) = U^c(\hat{\boldsymbol{v}}(r))$. Knowing $U^c(r) = \min_{n,m} u_{n,m}(r)$, from the definition of $U^c(r)$, and $u_{n,m}(r) = \frac{\Omega_n}{\sum_m \omega_{n,m}(r)} = \frac{1}{v_n(r)}$, from the proof of Lemma 1, we can conclude that $U^c(r) = \frac{1}{\max_n v_n(r)}$. Finally, from the definition of $\hat{V}_n(r)$, we have:

$$U^c(r) = \frac{1}{\frac{1}{U^c_{max}} + \max_n \hat{v}_n(r)}$$

The difference between the optimal utility and the cloud utility in round r is given by:

$$\begin{aligned} U^c_{max} - U^c(r) &= U^c_{max} - \frac{1}{\frac{1}{U^c_{max}} + \max_n \hat{v}_n(r)} \\ &= U^c_{max}\left(1 - \frac{1}{1 + U^c_{max} \max_n \hat{v}_n(r)}\right) \\ &= (U^c_{max})^2 \frac{\max_n \hat{v}_n(r)}{1 + U^c_{max} \max_n \hat{v}_n(r)} \\ &\leq (U^c_{max})^2 \max_n \hat{v}_n(r) \end{aligned}$$

For the corresponding random variables, we get $0 \leq U^c_{max} - U^c(\hat{\boldsymbol{V}}(r)) \leq k \cdot \max_n \hat{V}_n(r)$. Taking expectations, and knowing the convergence properties of $\{\mathbf{E}[\max_n \hat{v}_n(r)]\}_r$, we obtain following lemma:

Lemma 3. *The sequence $\{U^c_{max} - \mathbf{E}[U^c(r)]\}_r$, converges (at least) exponentially fast to 0 for $r \to \infty$, at the rate $\sqrt{\frac{1}{2\sqrt{e}}}$.*

With this lemma and Markov's inequality, it is straightforward to prove Theorem 1.

Proof of Theorem 1: Choose $\epsilon > 0$ and $\delta > 0$. Applying Markov's inequality to the random variable $(U^c_{max} - U^c(r))$, we get:

$$\Pr(U^c_{max} - U^c(r) \geq \epsilon) \leq \frac{\mathbf{E}[U^c_{max} - U^c(r)]}{\epsilon}$$

Since $\{U^c_{max} - \mathbf{E}[U^c(r)]\}_r$ converges to 0, there exists an r^* such that $\frac{\mathbf{E}[U^c_{max} - U^c(r)]}{\epsilon} \leq \delta$ holds for $r \geq r^*$, which implies that $|U^c(r) - U^c_{max}| < \epsilon, r > r^*$, with probability $1 - \delta$. Since $\{\mathbf{E}[U^c_{max} - U^c(r)]\}_r$ converges (at least) exponentially fast to 0, $r^* = \mathbf{O}(\log(\frac{1}{\delta}) + \log(\frac{1}{\epsilon}))$. ■

V. P*: A Heuristic Solution to OP(2)

In this subsection, we present the protocol P*, a distributed heuristic algorithm to solve OP(2). P* can be seen as an extension of P'. Recall that OP(2) differs from OP(1) in that it considers memory constraints and includes the secondary objective of minimizing the cost of changing the current to a new configuration. Considering the memory constraints for each machine turns OP(1) into an NP-hard optimization problem [22].

P* is an asynchronous protocol. This means that a machine does not synchronize the start time of a protocol round with any other machine of the cloud. At the beginning of a round (more precisely, at the start of the loop of the active or passive thread), a machine reads the current demands of the modules it runs. At the end of a round (more precisely, at the end of the loop of the active or passive thread) a machine updates its part of the configuration matrix A. The matrix A thus changes dynamically and asynchronously during the evolution of the system.

P* employs the same basic mechanism as P': it attempts to equalize the relative demands of two machines during a protocol round. However, due to the local memory constraints, equalization does not always succeed, in which case P* attempts instead to reduce the difference in relative demand. P* performs an equalization step in such a way that memory is either freed up or, if additional memory is needed, the amount is kept small. Further, P* attempts to keep down the cost of reconfiguration by preferring not to start a new module instance during an equalization step.

Out of consideration for local memory constraints and reconfiguration costs, a machine n prefers to perform an equalization step with another machine that runs some of the same modules. To this end, each machine maintains the set N_n of machines in the cloud that run some of the same modules as n (The machine manager maintains the set N_n with updates received from the respective site managers). However, if a machine n performs equalization steps only with machines in N_n, there is a chance of partitioning the cloud into disjoint sets of interacting machines, which can result in a system state far from optimal. For this reason, we introduce a parameter p in P* that indicates the probability of choosing a machine from N_n, otherwise, a machine from $N - N_n$ will be chosen.

The pseudocode of P* is given in Algorithm 3.

The initialization of P* is similar to that of P', except that $\boldsymbol{\omega}$ and $row_N(A)$ are not read in P*. Rather, they are read at the start of the loops for the passive and the active threads.

In the active thread of Algorithm 3, machine n chooses n' uniformly at random from the set N_n with probability p and from the set $N - N_n$ with probability $1 - p$.

The procedure `equalize()` attempts to equalize the relative demands of machines n and n'. It identifies the machine l with the larger 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 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 (step 1). Then, from the set of modules that run on both machines, it takes an instance with the smallest demand on l and shifts the demand to l'. This step is repeated until a total of $\Delta\omega$ demand has been moved, or until all modules in the set have been considered.

`moveDemand2()` equalizes (or reduces the difference of) the relative demands of two machines, by moving demand from the machine with the larger relative demand to the machine with the 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

Algorithm 3 ***Protocol P**** dynamically computes a heuristic solution for OP(2) and continuously updates a configuration matrix A. Code for node n.

initialization
1: read Ω_n, Γ_n;
2: start the passive and active threads
active thread
3: **while** true **do**
4: read $\boldsymbol{\omega}_n, \boldsymbol{\gamma}_n, row_n(A), N_n$;
5: **if** $rand(0..1) < p$ **then**
6: choose n' at random from N_n;
7: **else**
8: choose n' at random from $N - N_n$;
9: send $(\boldsymbol{\omega}_n, \boldsymbol{\gamma}_n, row_n(A), \Omega_n)$ to n';
10: receive $(\boldsymbol{\omega}_{n'}, \boldsymbol{\gamma}_{n'}, row_{n'}(A), \Omega_{n'})$ from n';
11: equalize($n', (\boldsymbol{\omega}_{n'}, \boldsymbol{\gamma}_{n'}, row_{n'}(A), \Omega_{n'})$);
12: write $row_n(A)$;
13: sleep(roundDuration);
passive thread
1: **while** true **do**
2: receive $(\boldsymbol{\omega}_{n'}, \boldsymbol{\gamma}_{n'}, row_{n'}(A), \Omega_{n'})$ from n';
3: read $\boldsymbol{\omega}_n, \boldsymbol{\gamma}_n, row_n(A), N_n$;
4: send $(\boldsymbol{\omega}_n, \boldsymbol{\gamma}_n, row_n(A))$ to n';
5: equalize($n', (\boldsymbol{\omega}_{n'}, \boldsymbol{\gamma}_{n'}, row_{n'}(A), \Omega_{n'})$);
6: write $row_n(A)$;
proc equalize($j, (\boldsymbol{\omega}_j, \boldsymbol{\gamma}_j, row_j(A), , \Omega_j)$)
7: $l = \arg\max\{v_n, v_j\}$;
$l' = \arg\min\{v_n, v_j\}$;
8: **if** $j \in N_n$ **then**
9: moveDemand1(l, l');
10: **else**
11: moveDemand2(l, l');

Algorithm 4 Procedures for moving demand between two machines.

proc moveDemand1(l, l')
1: compute $\Delta\omega$ such that $\frac{1}{\Omega_l}(\sum_m \omega_{m,l} - \Delta\omega) = \frac{1}{\Omega_{l'}}(\sum_m \omega_{m,l'} + \Delta\omega)$
2: let mod be an array of all modules that run on both l and l' sorted by increasing $\omega_{l,m}$
3: **for** $i = 1$ to $|mod|$ **do**
4: $m = mod[i]$; $\delta\omega = \min(\Delta\omega, \omega_{m,l})$;
5: $\Delta\omega -= \delta\omega$; $\delta\alpha = \alpha_{m,l}\frac{\delta\omega}{\omega_{m,l}}$; $\alpha_{m,l'} += \delta\alpha$; $\alpha_{m,l} -= \delta\alpha$;
proc moveDemand2(l, l')
1: compute $\Delta\omega$ such that $\frac{\sum_m \omega_{m,l} - \Delta\omega}{\Omega_l} = \frac{\sum_m \omega_{m,l'} + \Delta\omega}{\Omega_{l'}} = v^*$
2: let mod be an array of all modules that run on l sorted by decreasing $\frac{\omega_{l,m}}{\gamma_m}$;
3: **for** $i = 1$ to $|mod|$ **do**
4: $m = mod[i]$; $\delta\omega = \min(\Delta\omega, \omega_{m,l})$;
5: **if** $\gamma_m + \sum_{i|\alpha_{l',i}>0} \gamma_i \leq \Gamma_{l'}$ **then**
6: $\Delta\omega -= \delta\omega$; $\delta\alpha = \alpha_{m,l}\frac{\delta\omega}{\omega_{m,l}}$; $\alpha_{m,l'} += \delta\alpha$; $\alpha_{m,l} -= \delta\alpha$;

source that equalize the relative demands of the participating machines while observing the available memory of the destination means solving a Knapsack problem. A greedy 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 [23].

After invoking `equalize()`, both the active and passive thread write out the new row of the configuration matrix. (It is then the decision of the resource manager whether this change in configuration is implemented.)

VI. Evaluation through Simulation

We evaluate the protocol P* through simulations using a discrete event simulator that we developed in-house. We simulate the execution of P* in the resource manager components, as well as the execution of CYCLON in the overlay manager components of the machines in the cloud. CYCLON provides for P* the function of selecting a random machine for interaction. While we do not change the machines and sites during a simulation run, the load of each site changes dynamically with period d_s and asynchronously.

Evaluation metrics: We run the protocol P* in various scenarios and measure the following metrics. We express the *fairness* of resource allocation through the Coefficient of Variation of site utilities, defined 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 number of new module instances started divided by the number of all module instances running at the end a sampling period, per machine and per sampling period.

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 2. 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, its demand varies according to the formula $\omega_s(t) = \omega(s)(2 + \sin(\frac{2\pi t}{86400} - \phi_s))/2$ where ϕ_s is chosen uniformly at random from $[0..2\pi]$. This generates sinusoidal demand with mean $\omega(s)$, period 24hr and random phase. We choose the distribution for $\omega(s)$ among all sites to be Zipf distributed

with $\alpha = 0.7$, following evidence in [24]. The maximum value for the distribution is $c_\omega \cdot$270G CPU units and the population size used is 20,000. For a module m of site s, we choose a demand factor β_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_γ and c_ω are scaling factors.)

Capacity of machines: A machine has a 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. The probability of selecting a neighbor from the set N_n is chosen to be $p = \frac{|N_n|}{1+|N_n|}$. (The intuition is that p should increase with the size of N_n.)

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_γ and c_ω. In the reported experiments, we use the following parameters unless stated otherwise:

- round period=30 sec
- demand sampling period d_s=15 min
- number of machines=10,000
- number of sites=23,000
- MLF=CLF=0.5

A. Performance of P under varying CPU load factor (CLF) and memory load factor (MLF)*

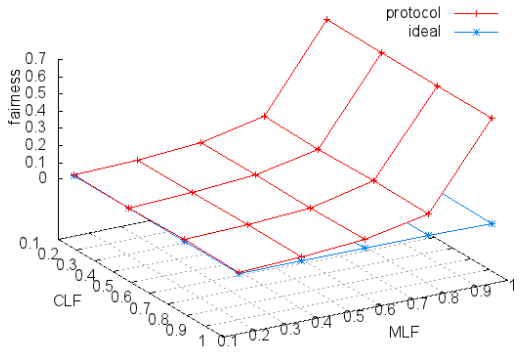
We evaluate the performance of P* for CLF={0.1, 0.4, 0.7, 1.0} and MLF={0.15, 0.35, 0.55, 0.75, 0.95}. (We leave out MLF=1 because there may not exist a feasible solution or Algorithm 1 may not find it.) We compare the performance of our protocol to that of an *ideal system*, which can be thought of as a centralized process making allocation for a load on a single machine with aggregate CPU and aggregate memory capacity of the entire cloud. The performance of the centralized system gives us a bound on the performance P*.

Each point in the graph of Figure 3 relates to the outcome of one simulation run that includes 85 load changes per machine measured after a warm up period. The metrics given are averaged over time. For all values, the 95% confidence interval for the value was found to be less than 10%.

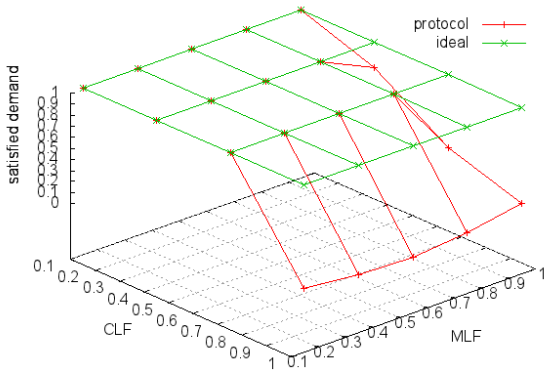
Fairness: First, Figure 3a shows that the fairness metric is independent of CLF. We expect this because P* allocates CPU resources proportional to the demand of a module instance, regardless of the available capacity on the particular machine. Second, the figure shows that resource allocation becomes less fair when MLF is increased. For example, the average deviation of the utilities of sites is about 60% from the average utility for MLF of 95%. However, this value decreases to less than 1% when MLF is at 15%. This behavior is also to be expected, since increasing MLF results in machines being less likely to have sufficient memory for starting new instances. Note that the ideal system always achieves optimal fairness, which means a value of 0.

Satisfied demand: Figure 3b shows that satisfied demand depends on both CLF and MLF. For the ideal system, the satisfied demand depends only on CLF and hence is always equal to 1. Our protocol satisfies more than 99% of all site demands for CLF$\leq$ 70% and MLF$\leq$ 55%. For larger values of MLF, more sites have their demands unsatisfied, even at low CLF levels, due to the unfair CPU allocation. We observe that for CLF=1 and low MLF values, satisfied demand is close to 0. This is the result of the fact that the actual CLF value for the simulation was larger than 1 ($\sim$1.02). In a situation where the CPU allocation is fair (which is the case for small values of MLF), all sites are allocated CPU resources that are less than their demand. For this value of CLF, increasing MLF results in a more 'unfair' CPU allocation, with more and more sites being allocated CPU capacities that satisfy their demand.

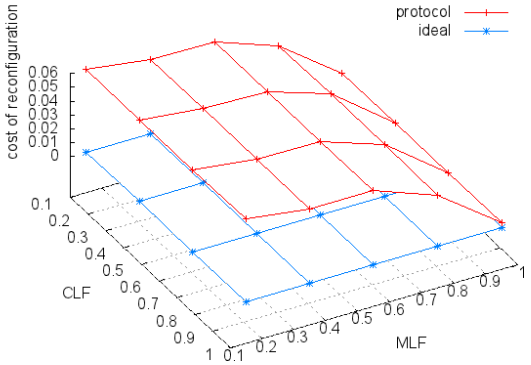
Cost of reconfiguration: Figure 3c suggests that the cost of reconfiguration does not depend on CLF but on MLF. Further, the cost increases with decreasing MLF. For instance, the cost of reconfiguration is less than 1% for MLF of 95%. However, this value increases to about 6% for MLF of 15%. This is attributed to the fact that when there is sufficient memory in the system, the protocol is free to move around modules in order to improve the fairness of resource allocation. The maximum cost of reconfiguration measured is less than 6%. This means that the average time between the start of two module instance is at least 20 minutes. Note that the cost of reconfiguration can be further reduced by (a) controlling the tradeoff between achieving a higher utility vs. increasing the cost of a configuration or by (b) improving the effectiveness of the protocol by tuning parameter p, for instance.



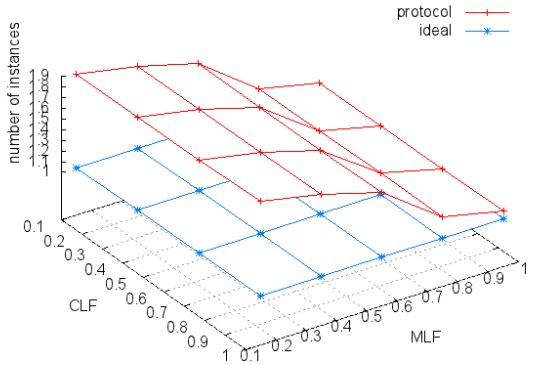
(a) Fairness among sites (0 means optimal fairness).



(b) Fraction of sites with satisfied demand.



(c) Cost of change in configuration over all machines.



(d) Average number of instances per module.

Fig. 3. The performance of the resource allocation protocol P* in function of the CPU load factor (CLF) and the memory load factor (MLF) of the cloud (10,000 machines, 23,000 sites).

Number of instances per module: Figure 3d shows the (average) number of instances running per module. As expected, this metric seems independent on CLF but dependent on MLF. Specifically, the number of instances per module decreases with increasing MLF. For high MLF values (e.g., MLF=0.95), this number is highly influenced by the available system memory (i.e., there simply is not enough memory to run an additional module instance).

B. Scalability

We measure the dependence of our evaluation metrics on the number of machines and the number of sites. To achieve this, we run simulations for a cloud with 5,000, 10,000, 20,000, 40,000, 80,000 and 160,000 machines and 11,500, 23,000, 46,000, 92,000, 184,000 and 368,000 sites respectively (keeping the ratio of sites to machines at 2.3, which ensures that CLF and MLF are kept close to the default value of 0.5).

Figure 4 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.

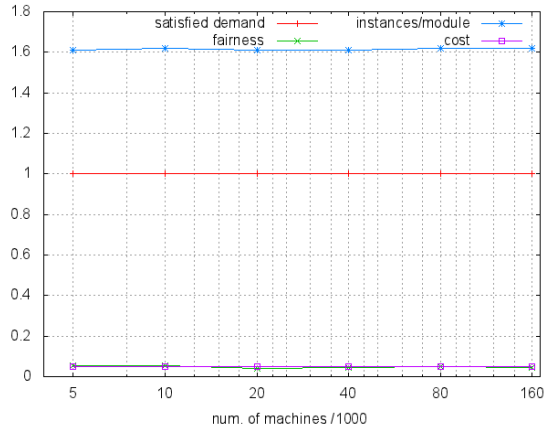


Fig. 4. Scalability with respect to the number of machines and sites.

VII. Related work

The problem of resource management we address in this paper is related to two lines of research, namely, to that of *application placement* and to that of *load balancing* in processor networks.

Application placement in datacenters is often modelled through mapping a set of applications onto a set of machines such that some utility function is maximized under resource constraints. This approach has been taken, e.g., in [18], [21], and solutions from these works have been incorporated in middleware products [6]. While these product solutions, in a similar way as our scheme does, allow for computing an allocation that maximizes the utility, they rely on centralized architectures, which do not scale to system sizes we consider in this paper.

The work in [25], which has been extended by [26] presents a distributed middleware 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 choice of utility functions in that work is such that service differentiation works very well under overload conditions, with the risk of starving unpopular applications. In contrast, our approach guarantees that every module receives its fair share of the CPU resources of the cloud, and that in underload conditions all modules are guaranteed to have satisfied demands. The proposed design in [25], [26] 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.)

In [27] the author presents a distributed algorithm for application placement. The authors consider an environment that hosts applications (equivalent to sites) with a number of components (equivalent to modules). The goal of the work is to minimize the load on the backbone links of the datacenter by moving components of the same application close to one another. This work is complimentary to the one reported in this paper as its primary focus is on resource allocation strategies that minimize the consumption of certain communication resources.

The work in [28] considers the problem of virtual machine placement under CPU and memory constraints. There, the authors use multi-criteria decision analysis to compute the placement of the virtual machines in a decentralized manner. Unlike us, their solution assumes the existence of an oracle with global information regarding the machines and their remaining capacities, which limits the scalability of their approach.

[29] presents a scheme for decentralized utility maximization considering a single type of resource. There, the authors present an optimal solution under the assumption that the demand of an application can be split over several machines. Their notion of utility is different from that of ours. Their solution has limited applicability in our context, since (1) we are considering multiple resources that need to be allocated on the same machine and (2) the demand for memory can not be split between machines.

Distributed load balancing algorithms have been extensively studied for homogeneous (i.e., servers of same capacity) as well as heterogeneous systems, for both divisible and indivisible demands. These algorithms typically fall into two classes: diffusion algorithms (e.g., [30], [31]) and dimension exchange algorithms (e.g., [32]). For both classes of algorithms, convergence results for different network topologies and different norms (that measure the distance between the system state and the optimal state) are reported in the literature. It seems to us that the problem of distributed load balancing, as formulated in the above line of research, is well understood today. The key difference to our work is that the problem we investigate includes local memory constraints, which the above algorithms do not consider.

VIII. Discussion and conclusion

With this paper, we make a significant contribution towards engineering a resource management middleware for cloud environments. 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/applications.

We presented a gossip protocol P* that computes, in a distributed and continuous fashion, a heuristic solution to the resource allocation problem for a dynamically changing resource demand. We evaluated the performance of this protocol through simulation. In all the scenarios we investigated, the protocol achieves the three qualitative design goals given in Section I. 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). By contrast, if we would solve the resource allocation problem expressed in OP(2) through a central periodic controller, then the CPU and memory demand for that controller would increase linearly with the system size.

We formally analysed the convergence property of a protocol P', which can be seen as an idealized version of P* that approximates the execution of P* in an environment where the available memory capacity is significantly larger than the memory demand and where the CPU demand does not change. We prove that the utility generated by a configuration computed by P' converges exponentially to the optimal utility at the rate $\frac{1}{\sqrt{2\sqrt{e}}}$.

The simulation results reported in this paper show that the heuristic protocol P* performs well for the parameter spaces investigated. We believe though that the protocol can be improved in two directions. First, P* is a greedy algorithm whereby interacting machines update their respective configurations, such that their local utility is increased, and it is possible that an execution of P* leads to a suboptimal global state. Consider a system with three machines n_1, n_2 and n_3. Machine n_1 has 1 unit of CPU and 1 unit of memory (which we write as 1/1), n_2 has 10/1, and n_3 has 1/2. Assume that n_1 runs a module m_1 requiring 10/1 amount of resources, n_2 runs m_2 requiring 1/1, and n_3 runs m_3 requiring 1/1. When executing P* on this system, only the interaction between nodes n_1 and n_3 leads to a state change, in which case the cloud utility (i.e., the global utility) increases from 0.1 to 0.18. However, a different protocol may let n_1 and n_2 swap modules (via n_3), in which case the cloud utility would become 1. A question that arises is whether P* can be extended in such a way that the global maximum for the cloud utility can be achieved in all cases. Second, the cost of reconfiguration incurred by P* depends on the exact sequence of interactions between nodes. For instance, consider the case where three identical machines n_1, n_2 and n_3 are running three modules m_1, m_2 and m_3, respectively. Assume that the CPU demands of the three modules are 1, 3 and 5 units of CPU, respectively, while the memory demands are negligible. In such a situation, if machines n_1 and n_3 interact first, the system reaches an optimal state with minimal cost (i.e., 1 new module started). Any other sequence of interactions will incur a higher cost. A question here is whether P* can be extended to improve its performance with respect to the cost of reconfiguration.

We believe that the approach to resource management in clouds outlined in this paper can be applied to other management objectives, to include additional or other constraints, and to include other resource types. For these reasons, we have developed a generic version of the gossip protocol P*, which can be instantiated for various purposes [1].

We view the results in this paper as building blocks for 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. (A mechanism for removing sites is straightforward, since P* will reallocate the freed-up resource.) (2) Extend the middleware design to become robust to various types of failures. (3) Extend the middleware design to span several clusters and several datacenters.

With respect to the protocol P*, we plan to investigate the following issues, in addition to the ones mentioned above. (1) Extend P* with a management control parameter that allows a management system to dynamically tune the CPU allocation to sites/applications. (2) Identify suitable decision functions that control the tradeoff between maximizing utility and minimizing cost of reconfiguration. Assess such decision functions that use only local input vs. functions that decide on implementing a new configuration based on global knowledge of the system state. (3) Investigate how close a configuration computed by P* is to the optimal solution to OP(2). (4) Extend P* to allow the memory demand to change over time. (5) Extend P* to consider additional resource types, such as storage and network resources. Extend P* to observe additional constraints including colocation/anti-colocation, license restrictions, etc.

REFERENCES

[1] R. Yanggratoke, F. Wuhib, and R. Stadler, "Gossip-based resource allocation for green computing in large clouds," in *International Conference on Network and Service Management*, October 2011.

[2] OpenNebula Project Leads, http://www.opennebula.org/, February 2012.

[3] OpenStack LLC, http://www.openstack.org, February 2012.

[4] Eucalyptus Systems, Inc., http://www.eucalyptus.com/, February 2012.

[5] UC Santa Barbara, http://appscale.cs.ucsb.edu/, February 2012.

[6] "IBM WebSphere Application Server," http://www.ibm.com/software/webservers/appserv/extend/virtualenterprise/, February 2012.

[7] VMWare, http://www.cloudfoundry.com/, February 2012.

[8] Amazon Web Services LLC, http://aws.amazon.com/ec2/, February 2012.

[9] Google Inc., http://code.google.com/appengine/, February 2012.

[10] Microsoft Inc., http://www.microsoft.com/windowsazure/, February 2012.

[11] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Transactions on Computer Systems*, vol. 23, no. 3, pp. 219–252, 2005.

[12] ——, "T-Man: Gossip-based fast overlay topology construction," *Computer Networks*, vol. 53, no. 13, pp. 2321–2339, 2009.

[13] F. Wuhib, R. Stadler, and M. Spreitzer, "Gossip-based resource management for cloud environments," in *International Conference on Network and Service Management*, Niagara Falls, Canada, October 2010.

[14] F. Wuhib, M. Dam, R. Stadler, and A. Clem, "Robust monitoring of network-wide aggregates through gossiping," *IEEE Transactions on Network and Service Management*, vol. 6, no. 2, pp. 95 –109, June 2009.

[15] F. Wuhib, M. Dam, and R. Stadler, "A gossiping protocol for detecting global threshold crossings," *IEEE Transactions on Network and Service Management*, vol. 7, no. 1, pp. 42 –57, March 2010.

[16] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "Dynamic estimation of CPU demand of web traffic," in *International Conference on Performance Evaluation Methodolgies and Tools*. New York, NY, USA, October 2006.

[17] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," in *International Conference on Network and Service Management*, Niagara Falls, Canada, October 2010.

[18] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguade, "Utility-based placement of dynamic web applications with fairness goals," in *IEEE Network Operations and Management Symposium*, Salvador, Bahia, Brazil, April 2008.

[19] 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.

[20] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. pp. 416–429, 1969.

[21] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *International Conference on World Wide Web*, Banff, Alberta, Canada, May 2007.

[22] H. Shachnai and T. Tamir, "On two class-constrained versions of the multiple knapsack problem," *Algorithmica*, vol. 29, no. 3, pp. 442–467, December 2001.

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

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

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

[26] 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 *International Conference on Integrated Network Management* Long Island, NY, USA, June 2009.

[27] C. Low, "Decentralised application placement," *Future Generation Computer Systems*, vol. 21, no. 2, pp. 281 – 290, 2005.

[28] Y. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, and Y. Coady, "Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis," in *IEEE International Conference on Cloud Computing*, Miami, FL, USA, July 2010.

[29] E. Loureiro, P. Nixon, and S. Dobson, "Decentralized utility maximization for adaptive management of shared resource pools," in *International Conference on Intelligent Networking and Collaborative Systems*, Washington, DC, USA, November 2009.

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

[31] R. Elssser, B. Monien, and R. Preis, "Diffusion schemes for load balancing on heterogeneous networks," *Theory of Computing Systems*, vol. 35, p. 2002, 2002.

[32] 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.

Fetahi Wuhib (http://www.ee.kth.se/~fzwuhib) is a postdoctoral researcher at KTH Royal Institute of Technology, where he conducts and leads research into decentralized monitoring and configuration of large-scale networked systems. Dr. Wuhib received a B.Sc. degree in Electrical Engineering from Addis Ababa University, Ethiopia, in July 2000. He received an M.Sc. degree in Internetworking, and a Ph.D. in Telecommunication from KTH in July 2005 and December 2010 respectively. His PhD. work has received *best dissertation award* at the 12^{th} IFIP/IEEE International Symposium on Integrated Network Management (IM2011). In 2009 he has been an intern in the Middleware Virtualization Group at the IBM T.J. Watson Research Center, New York.

Rolf Stadler (http://www.ee.kth.se/~stadler) is a professor at the Royal Institute of Technology (KTH) in Stockholm, Sweden. He holds an M.Sc. degree in mathematics and a Ph.D. in computer science from the University of Zurich. In 1991 he was a researcher at the IBM Zurich Research Laboratory. From 1992 to 1994 he was a visiting scholar at Columbia University in New York, which he joined in 1994 as a research scientist. From 1998 to 1999 he was a visiting professor at ETH Zurich. He joined the faculty of KTH in 2001 and is at the School of Electrical Engineering, where he leads research into management of networked systems.

Mike Spreitzer is a researcher at IBMs TJ Watson research center, working on topics in distributed systems including middleware management, autonomic infrastructure, and elastic storage and analytics. In the previous millennium he worked at Xerox PARC on topics in distributed systems, ubiquitous computing, user interfaces, programming environments, and VLSI design.