RC23980 (W0606-096) June 21, 2006
Computer Science

IBM Research Report

A Decentralized Application Placement Controller for Web Applications

Constantin Adam[1], Giovanni Pacifici[2], Michael Spreitzer[2], Rolf Stadler[1], Malgorzata Steinder[2], Chunqiang Tang[2]

[1]School of Electrical Engineering
Royal Institute of Technology
Stockholm, Sweden

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Decentralized Application Placement Controller for Web Applications*

Constantin Adam[1], Giovanni Pacifici[2], Michael Spreitzer[2], Rolf Stadler[1], Malgorzata Steinder[2], and Chunqiang Tang[2]
[1] *School of Electrical Engineering, Royal Institute of Technology, Stockholm, Sweden*
[2] *IBM T.J. Watson Research Center, Yorktown Heights, NY 10598*

Abstract. This paper addresses the problem of dynamic system reconfiguration and resource sharing for a set of applications in large-scale services environments. It presents a decentralized application placement scheme that dynamically provision enterprise applications with heterogeneous resource requirements. Potential benefits, including improved scalability, resilience, and continuous adaptation to external events, motivate a decentralized approach. In our design, all nodes run a placement controller independently and asynchronously, which periodically reallocates a node's local resources to applications based on state information from a fixed number of neighbors. Compared with a centralized solution, our placement scheme incurs no additional synchronization costs. We show through simulations that decentralized placement can achieve accuracy close to that of state-of-the-art centralized placement schemes (within 4% in a specific scenario). In addition, we report results on scalability and transient behavior of the system.

1 Introduction

The service sector has undergone a rapid expansion in the past several decades. In the United States, services account for approximately three quarters of GDP and eight out of ten jobs. This trend has driven the IT industry to shift its focus from the sales of computer hardware and software toward providing value-added IT services. Another trend in the industry is that many organizations increasingly rely on web applications to deliver critical services to their customers and partners. These organizations typically want to focus on their core competency and avoid unnecessary risks caused by sophisticated IT technologies. IT service providers are therefore encouraged to host the web applications in their data centers at reduced cost and with improved service quality.

An IT service provider that offers comprehensive enterprise computing services may run hundreds of data centers, and a single data center may host hundreds of applications running on thousands of machines. The sheer scale and heterogeneity of hardware and software pose grand challenges on how to manage these environments. Our vision is to engineer a middleware system that dynamically reconfigures in reaction to events, such as changes in the external demand, or node failures. In addition, the system should provide self-management functionality, by adapting to addition and removal of applications or resources, and changes in management policies. Fig. 1(a) is an example of such a system.

The idea of providing middleware support for large-scale web services is not new. Successful research projects in this area include Ninja [1] and Neptune [2], and leading

* This work was mainly done during Constantin's internship at IBM T.J. Watson Research Center in 2005. The authors are in alphabetic order. Contact: ctin@kth.se.

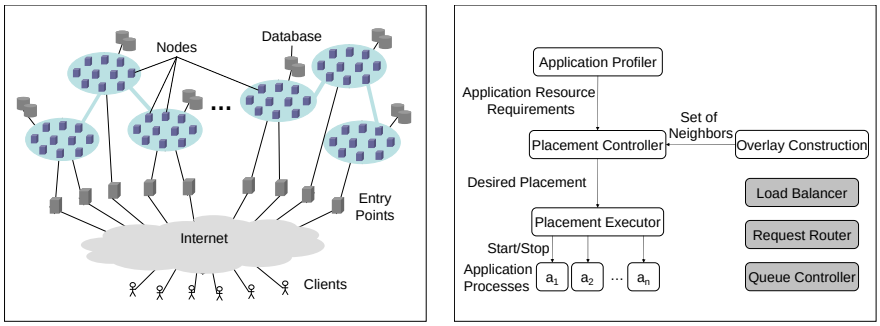

Fig. 1. (a) infrastructure for decentralized placement. (b) A node's components related to application placement.

commercial products include IBM WebSphere and BEA WebLogic. Despite the great success of these systems, the rapidly increasing demand for web services calls for next-generation middleware systems that address existing systems' limitations in resilience, scalability, and manageability.

Inspired by the recent success of peer-to-peer (P2P) systems with millions of users (e.g., KaZaA [3] and Skype [4]), we argue that next-generation middleware systems should leverage P2P technologies that have been proven to be scalable, fault-tolerant, and self-organizing. In P2P architectures, all active computing entities (or *nodes*) are treated equal. Nodes organize themselves into an application-level overlay network and collectively route traffic and process requests. Neither there is a single point of failure, nor the bottlenecks associated with a centralized system.

This paper focuses on a single component of our peer-to-peer middleware system: the *application placement controller*. Other components such as request routing and load balancing mechanisms will be presented in detail elsewhere. The placement controller supports several applications on a single node and therefore enables the deployment of a potentially large number of different applications in the system. Each node runs a placement controller that decides the set of applications that the node is offering. The decision aims at maximizing a global objective function.

The unique feature of our approach for application placement is the *decentralized design of the placement controller*. Each node independently and asynchronously executes the placement algorithm that manages its local resources. Although each node makes its own decisions independently, collectively all nodes work together to meet a global objective.

The problem of dynamic application placement has been studied before, and centralized solutions have been proposed in [8, 9, 13]. We argue that our decentralized approach offers advantages over a centralized design, such as increased scalability, absence of single points of failure, zero configuration complexity, and the capability to adapt continuously to changes in demand and to failures. Compared to a centralized solution, the additional overhead incurred by our approach is small in terms of computational costs and less efficient use of resources.

The rest of the paper is organized as follows. Section 2 gives an overview of our P2P middleware system. Section 3 presents the details of our decentralized application placement algorithm. Section 4 evaluates the algorithm and compares it with state-of-the-art centralized algorithms. Related work is discussed in Section 5. Finally, Section 6 concludes the paper.

2 Overview of Our P2P Middleware System

Fig. 1(a) shows a possible deployment scenario for our system. We consider a large-scale system with several entry points, each of which is logically connected to a large number of nodes, potentially all the nodes in the system. An entry point directs incoming requests to the nodes using a forwarding policy, such as weighted round robin.

The nodes self-organize into a logical overlay network through application-level (TCP) connections. They use the overlay for exchanging state and routing information. Many overlay construction protocols [6, 10–12] have been proposed, and most of them could be used in our system. A comparative assessment of overlay construction mechanisms for our system is beyond the scope of this paper.

Upon receiving a request, a node either processes it locally or forwards it to a peer that offers that application.

2.1 Node Model

As shown in Fig. 1(b), a node runs a single application process for each application $(a_1, a_2 \cdots a_n)$ it offers. The placement mechanism, which manages a set of applications, has three components: the profiler, the placement controller and the placement executor. The profiler gathers local application statistics, such as the request arrival rate, the memory consumed by an application process, and the average number of CPU cycles used by a request. The placement controller runs periodically on each node. It gathers application statistics from the local profiler, as well as from the profilers of the overlay neighbors. Then, it computes a list of application processes to run on the local node during the next placement cycle and sends this list to the executor, which stops and starts the required applications. The complete node model also includes components, such as request router, load balancer, and queue controller, which are not discussed here.

2.2 The Global Application Placement Matrix

If a node cannot process a request locally, it routes the request to another node using a local copy of the placement matrix P, which lists for each application the nodes that run it.

In addition to supporting routing, P provides information, such as the number of instances of a given application that are running in the entire system. This data can be used by the application placement controller to implement policies, such as the minimum or maximum number of application instances in the system.

The local copy of the matrix P on each node is maintained by GoCast, a gossip-enhanced multicast protocol [6]. Whenever a node changes the set of applications running locally, it uses GoCast to disseminate the changes to all other nodes.

Note that maintaining a copy of the global placement matrix at each node potentially limits the scalability of the system, as the number of broadcast operations increases linearly with the system size. However, GoCast enforces a maximum message rate on each overlay link, as it buffers all the updates received by a node during a certain time interval and aggregates them into a single message. The limitation in scalability therefore stems from the fact that the size of an aggregated message and the processing load on a node increases with the system size.

3 Decentralized Application Placement Controller

3.1 The Application Placement Problem

We consider a set $\mathcal{N}$ of nodes and a set $\mathcal{A}$ of applications. Let n be a node in $\mathcal{N}$ and a an application in $\mathcal{A}$. An application's demands for resources can be characterized as either *load-dependent* or *load-independent*. A running application instance's consumption of *load-dependent* resources depends on the offered load. Examples of such resources include CPU cycles and disk bandwidth. A running application instance also consumes some *load-independent* resources regardless of the offered load, even if the program processes no requests. An example of such a resource is the storage space for the program's executable.

Due to practical reasons, we treat memory as a load-independent resource and conservatively estimate the memory usage to ensure that every running application has sufficient memory. Our system includes a component that dynamically estimates the upper limit of an application's near-term memory usage based on a time series of its past memory usage. Because the memory usage estimation is dynamically updated, our placement controller indirectly considers some load-dependent aspects of memory.

We treat memory as a load-independent resource for several reasons. First, a significant amount of memory is consumed by an application instance, even if it receives no requests. Second, memory consumption is often related to prior application usage rather than to its current load. For example, even under low load, memory usage may still be high because of data caching. Third, because an accurate projection of future memory usage is difficult and many applications cannot run when the system is out of memory, it is reasonable to use a conservative estimation of memory usage, i.e., taking the upper limit instead of the average.

Among the many load-dependent and load-independent resources, we choose CPU and memory as the representative ones to be considered by our placement controller, because we observe that they are the most common bottleneck resources. For example, our experience shows that many business J2EE applications require on average 1-2GB real memory. While the description of our algorithm only considers CPU and memory, it can be applied to other types of resources as well. For example, if the system is disk-bounded, we can use disk bandwidth instead of CPU cycles as the load-dependent resource. Regarding measuring the demand and resource consumption of an application, we refer readers to prior work [13–15] on application profiling.

For an application $a \in \mathcal{A}$, let γ_a denote the memory demand of an instance of a, and ω_a^{req} denote the total CPU demand for a throughout the entire system. For a node $n \in \mathcal{N}$, let Γ_n and Ω_n denote its memory and CPU capacities, respectively. The CPU demand of an application is measured in CPU cycles/second (on a standard reference machine). We assume that, because of the memory constraint, a node n can run only a subset $\mathcal{R}_n$ of all applications offered by the system. Let $\omega_{n,a}^{req}$ be the CPU cycles/second needed on node n in order to process the request rate for application a. Let $\omega_{n,a}^{real}$ be the CPU cycles/second that node n allocates to application a. Let ω_a^{real} denote the total number of CPU cycles/second that the entire system allocates to application a, i.e., $\omega_a^{real} = \sum_{n \in \mathcal{N}} \omega_{n,a}^{real}$.

The goal of application placement is to maximize the sum of CPU cycles delivered to all applications.[1] We state the problem as follows:

$$\max \sum_{n \in \mathcal{N}} \sum_{a \in \mathcal{A}} \omega_{n,a}^{real} \tag{1}$$

such that

$$\forall n \in \mathcal{N} \qquad \Gamma_n \geq \sum_{a \in \mathcal{R}_n} \gamma_a \tag{2}$$

$$\forall n \in \mathcal{N} \qquad \Omega_n \geq \sum_{a \in \mathcal{R}_n} \omega_{n,a}^{real} \tag{3}$$

Formulas 2 and 3 stipulate that the allocated CPU and memory resources on each node cannot exceed the node's CPU and memory capacities, respectively.

3.2 The Application Placement Algorithm

Our placement algorithm executes periodically on each node. The time between two executions of the algorithm is called the *placement cycle*.

The placement algorithm has three consecutive phases. In the first phase, a node gathers placement and load information from its neighbors. In the second phase, the node determines a set of applications to run locally during the next placement cycle. In the last phase, the node carries out the placement changes (i.e., start or stop of applications) and advertises its new placement configuration to other nodes using GoCast. We give the pseudo-code of the algorithm and describe each of its phases below.

```
 1.  class AppInfo {
 2.    string app_id;
 3.    double cpu_demand, cpu_supply;
 4.  }
 5.  List<AppInfo> active_apps, standby_apps, new_active_apps;
 6.  List<AppInfo> neighbor_active_apps;
 7.  double max_cpu_supply, cpu_supply;

 8.  while(true) {
 9.    active_apps=getLocalActiveApps();
10.    neighbors=overlay.getNeighbors();
11.    neighbor_active_apps=getAppStats(neighbors);
12.    standby_apps=getIdleAppStats(local_node, neighbors);
13.    for each app in neighbor_active_apps
14.      if(active_apps.contains(app)==false)
15.        if(app.cpu_demand>app.cpu_supply)
16.          standby_apps.add(app);
17.    active_apps=sortIncreasingDensity(active_apps);
18.    standby_apps=sortDecreasingUnmetDensity(standby_apps);
19.    new_active_apps=active_apps;
20.    max_cpu_supply=currentCpuSupply()
21.    num_active_apps=active_apps.size();
22.    for(i=0;i<=num_active_apps;i++) {
```

[1] Our system can also be configured to optimize a certain utility function, but a detailed discussion is beyond the scope of this paper.

```
23.      remove top i apps from active_apps;
24.      cpu_supply=allocateIdleResources(standby_apps);
25.      if((cpu_supply-change_cost)>max_cpu_supply) {
26.        max_cpu_supply=cpu_supply-change_cost;
27.        new_active_apps=active_apps-top_i_active_apps+sel_standby_apps;
28.      }
29.    }
30.    if(new_active_apps!=active_apps) {
31.      advertise(new_active_apps, state=STARTING);
32.      stopAndStartApps(active_apps, new_active_apps);
33.      advertise(new_active_apps, state=ACTIVE);
34.    }
35.    wait until the end of the placement cycle;
36.  }
```

Phase 1: Gathering State Information. A node retrieves from each neighbor x the list of applications $(a_1 \cdots a_m)$ running on x, the memory requirements $(\gamma_{a_1} \cdots \gamma_{a_m})$ of those applications, the CPU cycles/second $(\omega^{real}_{x,a_1}, \cdots, \omega^{real}_{x,a_m})$ delivered to those applications, and the CPU demands of those applications $(\omega^{req}_{x,a_1}, \cdots, \omega^{req}_{x,a_m})$ (lines 10-11). In addition, neighbor x also reports the locally measured demands for applications it could not route, since they are not offered in the system (line 12). (A high demand for these inactive applications might trigger their activation during the next placement cycle.)

Phase 2: Computing a New Set of Active Applications. Using the information gathered in the previous phase, a node builds a set of active applications $\mathcal{R} = \{r_1, \cdots, r_i\}$ (line 9) and a set of standby applications $\mathcal{S} = \{s_1, \cdots, s_j\}$ (lines 12-16). $\mathcal{R}$ contains the applications that are currently active on the local node. $\mathcal{S}$ contains two types of applications: those that run in the neighborhood of the node but not on the node itself, and applications are currently not offered in the system.

The placement algorithm attempts to replace a subset of applications in $\mathcal{R}$ with a subset of applications in $\mathcal{S}$, so that the local CPU utilization is maximized. Since the candidate space for the optimal configuration grows exponentially with $|\mathcal{R} \cup \mathcal{S}|$, we apply a heuristic that reduces the complexity of the problem to $O(|\mathcal{R}| * |\mathcal{S}|)$ (lines 17-29).

On a node n, the active applications in $\mathcal{R}$ are sorted in increasing order of their density d_a, which we define as the load delivered by n to application a, divided by the memory consumption of a, i.e., $d_a = \omega^{real}_{n,a}/\gamma_a$. A high-density application consumes system resources efficiently, in the sense that it causes a relatively high CPU utilization while consuming a relatively little memory. The applications in the standby set $\mathcal{S}$ are sorted in decreasing order of their residual density d^*_a, which we define as the unmet demand of a divided by its memory demand, i.e., $d^*_a = \sum_{n \in neighbors}(\omega^{real}_{n,a} - \omega^{req}_{n,a})/\gamma_a$. The standby applications that have no unmet demands are removed from $\mathcal{S}$, because there is no need to start additional instances for them. Intuitively, the placement controller tries to replace low-density applications in the active set $\mathcal{R}$ with high-density applications in the standby set $\mathcal{S}$, so that the CPU utilization is maximized.

The placement algorithm has $|\mathcal{R}| + 1$ iterations ($k = 0 \cdots |\mathcal{R}|$). During the first iteration ($k = 0$), it does not remove any application from $\mathcal{R}$. If the local node n has available memory and CPU cycles (i.e., $\Gamma^{free}_n > 0$ and $\Omega^{free}_n > 0$), then the algorithm attempts to add one or more applications from $\mathcal{S}$ to $\mathcal{R}$. This is done by selecting applications from the top of $\mathcal{S}$, subtracting the cost for starting the applications, and evaluating the resulting gain in CPU utilization.

During iteration $k > 0$, the algorithm removes the top k applications from $\mathcal{R}$. It then computes the available memory and CPU resources and attempts to assign these resources to applications in $\mathcal{S}$ in the following way. The algorithm attempts to fit the first application $s_1 \in \mathcal{S}$ into the available memory. If this operation succeeds, then the algorithm attempts to allocate the entire unmet CPU demand for s_1. This means that $\min((\omega_{s_1}^{req} - \omega_{s_1}^{real}), \Omega_n^{free})$ CPU cycles/second are allocated to application s_1. If there is not enough free memory to fit s_1, the algorithm continues with the next application $s_2 \in \mathcal{S}$, etc. The iteration k ends when either the free memory or CPU are exhausted, or when all the applications in $\mathcal{S}$ have been considered.

After each iteration k, the placement controller produces a new placement solution $\mathcal{R}^k$ that lists a set of applications to run on the local node during the next placement cycle. At the end of the loop, the placement algorithm returns from the set $\{\mathcal{R}^k | k = 0, 1, \cdots, |\mathcal{R}|\}$ the configuration that maximizes the total number of CPU cycles/second delivered to the active applications.

Note that the algorithm takes into account the cost of stopping and starting an application, as illustrated in the following example. If, for instance, starting application s consumes the local CPU for 9 seconds and the length of the placement cycle is 900 seconds, then the cost of starting s is 1% of the total CPU resources of the node during that placement cycle. The same rule applies for stopping an application. (A more detailed model could take into account the current node utilization and the fact that the start and the stop processes do not take the entire CPU capacity.)

Phase 3: Reconfiguring and Advertising the New Configuration. The algorithm announces the new configuration for the node if the set of active applications has changed (lines 30-31). Next, it switches from the old configuration to the new configuration by stopping and starting applications (line 32). After completing these operations, the algorithm advertises its current configuration (line 33).

The placement controllers process the announcement of a new configuration, whose purpose is to reduce unnecessary configuration changes, such as several nodes activating the same idle application. The announcement of the commitment of the placement operation is used for routing requests.

3.3 Comparison with Centralized Placement Controllers

Our decentralized scheme for application placement has a number of advantages over proposed centralized solutions [8, 9, 13].

First, it *scales* to a large number of nodes. The controller makes decisions based on state information collected from a small subset of the nodes in the system (i.e. its neighbors), which does not grow with the system size. (As we have discussed, there is a limitation to which size the update propagation scheme scales, since it is based on maintaining a copy of the global placement matrix on each node. We are currently working on a decentralized update scheme, where each node knows about a configurable number of providers for each application.)

Second, decentralized application placement contributes to a *robust* system, as failures of single nodes do not affect the availability of the rest of the system. In contrast to a centralized system, all components are functionally identical.

Third, decentralized placement enables a large system to adapt to external events almost *instantly*, while a centralized approach does not have this capability. Consider a centralized controller that periodically gathers data about the state of the system and computes a placement solution. The timescale according to which such a system can adapt is determined by the length of the placement cycle. In the decentralized case, a

large number of controllers reconfigure asynchronously and periodically. The placement operations are distributed over time and parts of the system react almost instantly to external events.

Fourth, decentralization of the application placement does not come at the cost of additional complexity. All communication takes place between pairs of nodes, without further synchronization. Furthermore, one can use the same algorithm for centralized, as well as decentralized placement.

Compared to a centralized solution, decentralized application placement also has disadvantages, as our results in the next section show. Running a controller on each node generally results in a higher processing overhead for application placement. Second, the number of changes in the sense of application starts and stops is usually larger in the decentralized case, since there is no coordination between the decisions of the local controllers.

4 Experimental Results

We implemented our design in Java and studied its behavior through extensive simulations. The implementation is based on the javaSimulation package [16] that provides a general framework for process-based discrete event simulation. (We also implemented our algorithms in a real system based on the Tomcat [17] environment. Measurements from the testbed will be reported elsewhere.)

We have conducted simulations with a large number of system configurations and load patterns. We use several models for synthetic load because we are not aware of any large-scale load traces of J2EE applications. Due to space limitations, we only report results here for one type of load pattern called "power-law" [18], where applications are ranked according to their popularity and the application with rank α is assigned a random weight in the interval $[0, \alpha^{-2.16}]$. To obtain the load for individual applications, these weights are normalized, and multiplied by the total CPU demand for all applications. In our experiments, the load changes periodically. The weights assigned to the applications in a given period are independent of the previous period.

Following our previous work [8], we use two parameters to control the difficulty of an application placement problem: the *CPU load factor* (CLF) and the *memory load factor* (MLF). The CLF is the ratio between the total CPU demand of the applications and the total CPU capacity available in the system: $CLF = \sum_{a \in \mathcal{A}} \omega_a^{req} / \sum_{n \in \mathcal{N}} \Omega_n$. Similarly, the MLF is the ratio between the sum of the memory required by each deployed application and the total memory available in the system: $MLF = \sum_{a \in \mathcal{A}} \gamma_a^{req} / \sum_{n \in \mathcal{N}} \Gamma_n$. In case $MLF = 1$, the system has enough total memory to run one instance of each application. Note that the memory is scattered across the nodes, and it might be not possible to find a placement solution for any value of $MLF > 0$.

We use two performance metrics to evaluate the placement algorithms: the accuracy and the number of placement changes. Accuracy is the ratio between the satisfied CPU demand and the total CPU demand. An accuracy of 1 means that the system can process all the incoming requests. The number of placement changes is the total number of application starts and stops in the entire system during one placement cycle. A good algorithm should exhibit high accuracy and a low number of placement changes.

In the simulation, the memory capacity Γ_n of a node is uniformly distributed over the set $\{1, 2, 3, 4\}$ GB. The memory requirement γ_m of an application is uniformly distributed over the set $\{0.4, 0.8, 1.2, 1.6\}$ GB. The CPU capacity Ω_n is the same for all nodes and is set to 2 GHz.

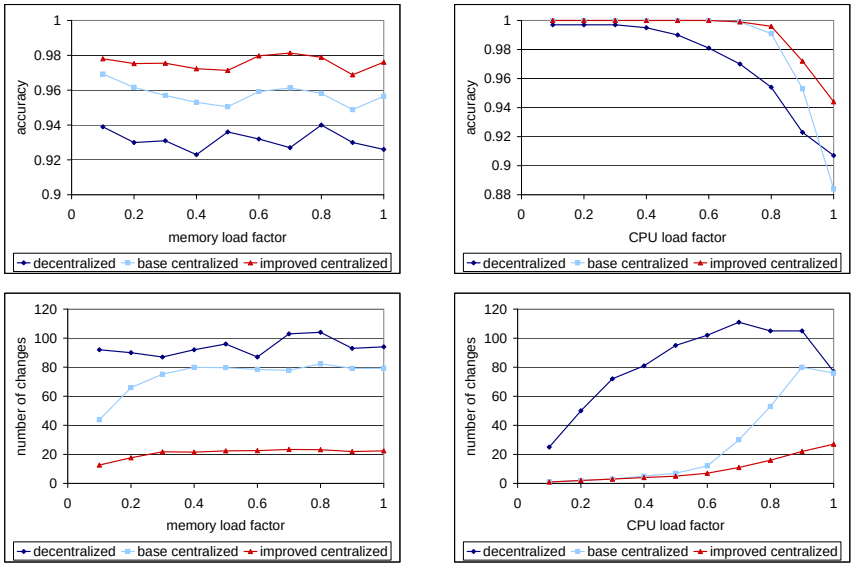

Fig. 2. Comparison between the decentralized and centralized placement algorithms for 100 nodes. Top: Placement Accuracy (left: CLF=0.9, right: MLF=0.4) Bottom: Number of changes (left: CLF=0.9, right: MLF=0.4)

Unless otherwise noted, each placement cycle lasts 900 seconds and each simulation run lasts 3600 seconds. The first cycle (900 seconds) is a warm-up period and the results from this period are discarded. The load pattern changes every 900 seconds and the performance metrics (accuracy and number of placement changes) are measured at the end of each placement cycle. During a placement cycle, nodes independently pick a random time to run the placement algorithm. The default time to start or stop an application is 0 seconds.

The measurement points in the Figs. 2, 3 are averages of 100 simulation runs.

Comparing the Decentralized and Centralized Algorithms. Fig. 2 shows a performance comparison between our decentralized placement controller and two centralized controllers with the same control objectives. The "base centralized" algorithm [8] (previously developed by us) uses a greedy heuristic and two network flows algorithms (max flow and min cost) on a bi-partite graph to find the optimal allocation. The "improved centralized" algorithm is our most recent, enhanced version of the "base centralized" algorithm.

The evaluation is based on a system with 100 nodes. In the decentralized case, each node has 16 neighbors. We report on two series of simulations. First, we set $CLF = 0.9$ and vary MLF from 0.1 to 1. For the second series, we set $MLF = 0.4$ and vary CLF from 0.1 to 1.

Fig. 2 shows the evaluation results. The accuracy of the decentralized algorithm is about 5% lower than the accuracy of the improved centralized algorithm. This cost in CPU resources is paid for a decentralized solution. We further observe that the number of placement changes in the decentralized algorithm is several times higher than that of the improved centralized algorithm. We expect a higher number of changes in the decentralized case, because the nodes make decisions independently and do not coordinate them. We believe that the number of changes could be reduced by applying only those changes that significantly improve the control objective.

As expected, the CLF influences the accuracy significantly. For relatively low CPU utilizations ($CLF \leq 0.5$), the accuracy of the decentralized algorithm is close to that of

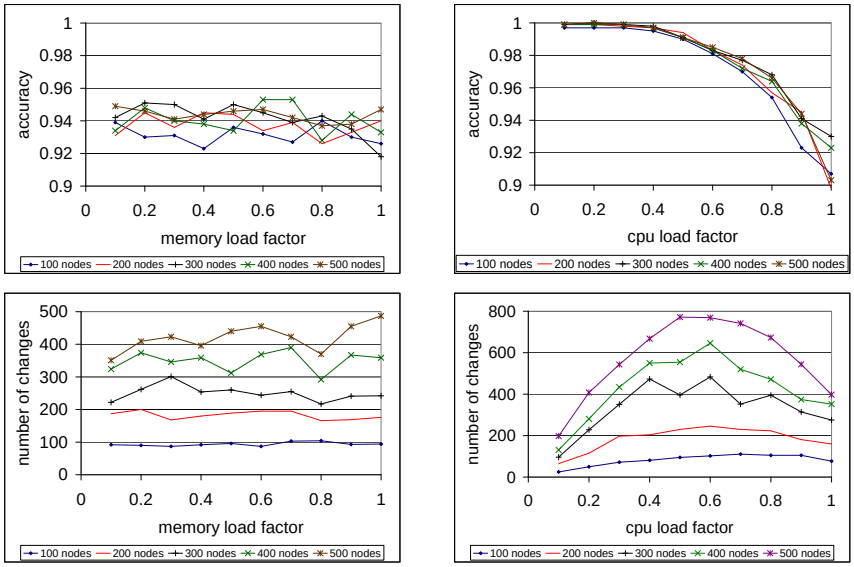

Fig. 3. Scalability of the decentralized placement algorithm. Top: placement accuracy (left: CLF=0.9, right: MLF=0.4). Bottom: number of changes (left: CLF=0.9, right: MLF=0.4).

the centralized algorithm and, furthermore, close to the ideal value of 1. The decrease in accuracy accelerates for $CLF \geq 0.8$. In the decentralized case, the number of placement changes increases with CLF, peaks around $CLF = 0.8$ and decreases afterwards. We explain this decrease by the fact that, for a high CLF, many nodes are loaded close to their capacity, and it is difficult for the algorithm to find placement changes that can further improve their utilization.

Scalability of the Decentralized Placement Algorithm. To investigate the scalability of our algorithm, we consider a system where nodes have on average 16 neighbors. We report on two series of simulations, in which we vary the number of nodes in the system from 100 to 500. First, we set $CLF = 0.9$ and vary MLF from 0.1 to 1. For the second series, we set $MLF = 0.4$ and vary CLF from 0.1 to 1. Fig. 3 shows that the accuracy tends to slightly increase by about 1-2% when the system size increases from 100 to 500 nodes. Fig. 3 also shows that the number of placement changes increases linearly with the size of the system.

Convergence after a Load Change. Fig 4 illustrates the system behavior during transient phases that follow changes in the external load pattern. The system has 200 nodes and each node has on average 16 neighbors. We perform three series of experiments for which the time needed to start or to stop an application is 0, 20, and 100 seconds, respectively. The length of the placement cycle is 900 seconds. The system starts in a random configuration and, at time 0, we generate a load pattern that does not change during the remainder of the simulation. We set $MLF = 0.4$ and $CLF = 0.9$.

Fig. 4 shows that the increase in accuracy and the number of placement changes is initially large and then levels out over time. We observe that the time t needed to start and stop applications significantly impacts the convergence time of the system. Specifically, if t is small (0 seconds or 20 seconds), the system converges during a single placement cycle. For large values of t (100 seconds), the system needs 2-3 placement cycles to converge. When increasing t, the placement accuracy becomes lower, because some CPU resources are used for starting and stopping applications. As expected, we observe changes in the system even in steady state.

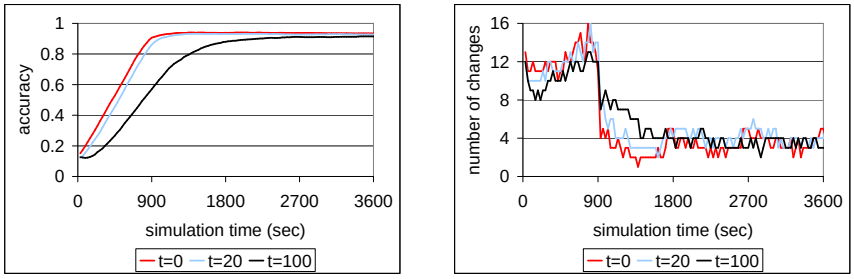

Fig. 4. Convergence after a load change. Left: placement accuracy. Right: number of changes.

5 Related Work

The application placement problem, as described in Section 3.1, is a variant of the class constrained multiple-knapsack problem, which is known to be NP-hard [19]. Variations of this problem have been studied extensively in several contexts. In the area of application placement for web services, this work is closely related to the *centralized* placement algorithm [8] previously developed by us.

Stewart et al. [13] present a centralized method for automatic component placement that maximizes the overall system throughput. In the area of content delivery and stream processing, [20–22] describe methodologies for placing a set of operators in a network, by balancing two objectives: (a) minimizing the delay of the information delivery and (b) minimizing the bandwidth used to send the information. In the context of utility computing, [23] presents a decentralized placement algorithm that clusters application components according to the amount of data they exchange.

6 Conclusion

In this paper, we presented a decentralized control scheme for application placement that attempts to maximize resource utilization. Through simulations, we have shown that decentralized placement can achieve accuracy close to that of state-of-the-art centralized placement schemes (within 4% in a specific scenario). Our decentralized scheme does not include additional synchronization costs compared to a centralized solution. Moreover, conceptual advantages such as scalability, resilience and continuous adaptation to external events motivate the study of decentralized placement as an alternative to a centralized scheme.

A number of issues require further consideration. Our current design does not address the server affinity problem and the concept of the user session. In addition, we did not consider the interaction with the database tier in our design, which limits the use of the current scheme to applications that do not require transactional capabilities or state persistence. Finally, we need to improve further the scalability of our placement scheme. As we have shown, maintaining a copy of the global placement matrix at each node potentially limits scalability, as the processing load on a node increases with the system size. We are currently evaluating an adaptive update mechanism, where each node only knows about a configurable number of providers for each application.

We are in the process of implementing and evaluating the scheme described in this paper in the Tomcat environment. The placement controller runs as an application filter in Tomcat, following the approach described in [24]. We plan to evaluate the performance of our system using the TPC-W and RUBiS benchmarks.

References

1. S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, B. Zhao, "The Ninja Architecture for Robust Internet-Scale Systems and Services," Journal of Computer Networks, vol. 35, no. 4, March 2001.
2. K. Shen, H. Tang, T. Yang, L. Chu, "Integrated Resource Management for Cluster-based Internet Services", *OSDI'02*, December, 2002.
3. KaZaA, http://www.kazaa.com, June 2006.
4. Skype, http://www.skype.com, June 2006.
5. C. Tang, M. J. Buco, R. N. Chang, S. Dwarkadas, L. Z. Luan, E. So, C. Ward, "Low Traffic Overlay Networks with Large Routing Tables,", *ACM SIGMETRICS'05*.
6. C. Tang, R. N. Chang, C. Ward, "GoCast: Gossip-enhanced Overlay Multicast for Fast and Dependable Group Communication", *DSN'05*, Yokohama, Japan, 2005.
7. R. M. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. N. Tantawi, A. Youssef, "Performance Management for Cluster Based Web Services", *IEEE/IFIP IM*, 2003.
8. A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, A. Tantawi, "Dynamic Application Placement for Clustered Web Applications", *the International World Wide Web Conference (WWW)*, 2006.
9. S. Shingal, S. Graupner, A. Sahai, V. Machiraju, J. Pruyne, X. Zhu, J. Rolia, M. Arlitt, C. Santos, D. Beyer, J. Ward., "Quartermaster: A Resource Utility System", *IEEE/IFIP IM 2005*, Nice, France, May 2005.
10. Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H., Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM 2001.*
11. Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S., A scalable content-addressable network. *ACM SIGCOMM 2001.*
12. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, *IFIP/ACM Middleware 2001.*
13. C. Stewart K. Shen, S. Dwarkadas, M. Scott, *Profile-driven Component Placement for Cluster-based Online Services*, IEEE Distributed Systems Online, October 2004.
14. B. Urgaonkar, P. Shenoy, T. Roscoe, "Resource Overbooking and Application Profiling in Shared Hosted Platforms", *USENIX 2002*, Boston, MA, December 2002.
15. G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, A. Youssef, "Managing the response time for multi-tiered web applications", IBM, Tech. Rep. RC 23651, 2005.
16. K. Helsgaun, *Discrete Event Simulation in Java*, Writings on Computer Science, 2000, Roskilde University.
17. Apache Tomcat, http://tomcat.apache.org, June 2006.
18. L.A. Adamic, B.A. Huberman,"Zipf's law and the Internet", *Glottometrics 3*,2002.
19. H. Kellerer, U. Pferschy, D. Pisinger, *Knapsack Problems*, Springer-Verlag, 2004.
20. S. Buchholz and T. Buchholz, "Replica placement in adaptive content distribution networks", *ACM SAC 2004*, Nicosia, Cyprus, March 2004.
21. K. Liu, J. Lui, Z-L Zhang, "Distributed Algorithm for Service Replication in Service Overlay Network", *IFIP Networking 2004*, May, 2004, Athens, Greece.
22. P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer, "Network-Aware Operator Placement for Stream-Processing Systems". *ICDE'06*, Atlanta, GA, April 2006.
23. C. Low, "Decentralized application placement", Future Generation Computer Systems 21 (2005) 281-290.
24. C. Adam and R. Stadler, "Implementation and Evaluation of a Middleware for Self-Organizing Decentralized Web Services", *IEEE SelfMan 2006*, Dublin, Ireland, June 2006.