

# A Service Middleware that Scales in System Size and Applications

Constantin Adam<sup>1</sup>, Rolf Stadler<sup>1</sup>, Chunqiang Tang<sup>2</sup>, Malgorzata Steinder<sup>2</sup>, Michael Spreitzer<sup>2</sup>

<sup>1</sup> School of Electrical Engineering, Royal Institute of Technology, Stockholm, Sweden

<sup>2</sup> IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

## Abstract

We present a peer-to-peer service management middleware that dynamically allocates system resources to a large set of applications. The system achieves scalability in number of nodes (1000s or more) through three decentralized mechanisms that run on different time scales. First, overlay construction interconnects all nodes in the system for exchanging control and state information. Second, request routing directs requests to nodes that offer the corresponding applications. Third, application placement controls the set of offered applications on each node, in order to achieve efficient operation and service differentiation. The design supports a large number of applications (100s or more) through selective propagation of configuration information needed for request routing. The control load on a node increases linearly with the number of applications in the system. Service differentiation is achieved through assigning a utility to each application, which influences the application placement process. Simulation studies show that the system operates efficiently for different sizes, adapts fast to load changes and failures and effectively differentiates between different applications under overload.

## I. INTRODUCTION

Fundamentally, our aim with this work is to develop engineering principles for large-scale autonomous systems, where individual components self-organize and work together towards a common goal that we can control. In this paper, we do this in the context of application services. We present a design that dynamically allocates CPU and memory resources to a potentially large number of applications offered by a global server cluster. The novel aspect of our design is that all its functions, including resource allocation, are decentralized, which forms the basis for scalability and robustness of the system.

Our design has three features characteristic to many peer-to-peer systems. First, it *scales with the number of nodes*, because each node receives and processes state information only from a small subset of peers that is independent of the system size. Second, the design is *robust*, since all the nodes are functionally identical, and the failure of a node does not affect the availability of the rest of the system. Finally, the design enables a large system to *adapt quickly* to external events. As each node runs its local control mechanisms asynchronously and independently from other nodes, (periodic) local control operations are distributed over time, which lets parts of the systems re-configure shortly after events (such as load changes or node failures) occur.

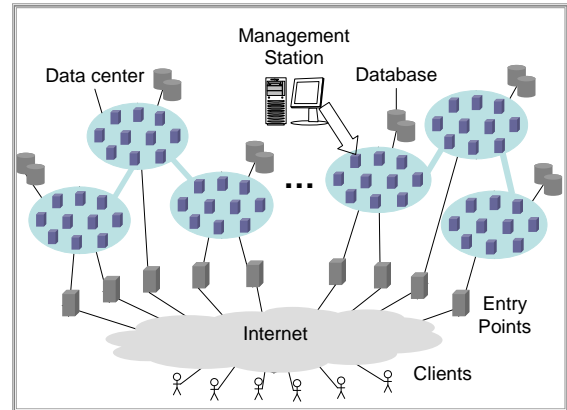


Fig. 1. A possible deployment scenario for the design in this paper. The design includes mechanisms that run on the servers in data centers, the entry points and the management station.

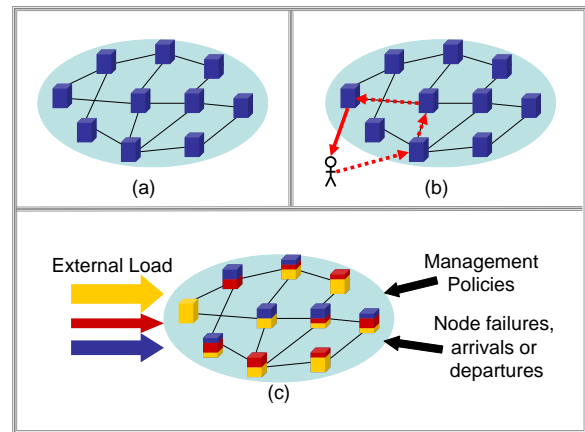


Fig. 2. Three decentralized mechanisms control the system behavior: (a) topology construction, (b) request routing, and (c) application placement.

Fig. 1 shows a possible deployment scenario for the design in this paper. The scenario includes several data centers and many entry points. The system offers a large variety of application services that are accessed through the Internet. We consider computationally intensive services, such as remote computations and services with dynamic content, e.g., e-commerce, or online tax filing. Throughout the paper, we refer to the combined set of nodes in all the data centers as the *cluster*.

This work builds on earlier results by us in the context of engineering scalable middleware for web services. It includes significant extensions and modifications to our earlier designs ([1], [2]), as we address important shortcomings of those designs and conduct a more thorough evaluation of the system. In [1] we introduced a design similar to the one in this paper,

which is though restricted to a small number of applications, as each system application requires its own overlay. The design in this paper uses only a single overlay, which is made possible through the introduction of a forwarding table for request routing and the concept of selective propagation of state to maintain the forwarding tables. In [2] we introduced a decentralized controller for application placement. In this paper, we use a simplified version of this controller, improve its scalability, and extend it to achieve service differentiation.

The rest of the paper is organized as follows: Section II gives an overview of our P2P design. Section III evaluates the algorithm through simulation. Section IV discusses related work. Finally, Section V concludes the paper.

## II. SYSTEM DESIGN

### A. System Model

We consider a set  $\mathcal{N}$  of nodes and a set  $\mathcal{A}$  of applications. A node can run concurrently (instances of several) applications. We assume that the CPU and the memory are bottleneck resources on each node. The CPU is a *load-dependent* resource, as its consumption depends on the offered load. The memory is a *load-independent* resource, as we assume that it is consumed regardless of the offered load, i.e., even if the program processes no requests ([3]).

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 its current load. For example, even in the presence of a low load, memory usage may still be high because of data caching. Third, as 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 be conservative in the estimation of memory usage, i.e., by using the upper limit instead of the average.

### B. The Role of the Entry Point

Our design assumes the entry points have the functionality of a layer-4 switch. While entry points with more advanced functionality, such as layer-7 switching, load balancing, or sophisticated scheduling can provide similar functionality to our design, or fit in our design as well, we advocate using our design together with layer-4 switches, for the following reasons. First, the realization of a layer-4 entry point is generally simpler, more efficient and more economical than that of a layer-7 switch [4]. Second, our approach is potentially more resilient. Since a layer-4 entry point does not hold any state information about load and resource usage, its failure will affect only the pending requests. Third, our solution scales better. Policies such as service differentiation and quality of service objectives are realized through the application placement mechanism, which runs on every node of the system, rather than on the entry point. A single entry point can manage resources for only a limited number of nodes. Increasing the number of entry points makes synchronization among them necessary and adds to the system complexity.

Fundamentally speaking, supporting sophisticated resource allocation strategies on an entry point has all the disadvantages inherent to centralized control schemes, when compared to the decentralized scheme we advocate.

### C. Decentralized Control Mechanisms

Three distributed mechanisms form the core of our design, as shown in Fig. 2. *Topology construction* uses an epidemic protocol and a set of local rules to organize dynamically the nodes into an overlay, through which they disseminate state and control information in a scalable and robust manner. *Request routing* directs service requests towards available resources. This mechanism maintains the forwarding tables of the nodes by disseminating information about which nodes run which applications. *Application placement* dynamically assigns the cluster resources (CPU and memory) to applications. Resource allocation is achieved by continuously maximizing a utility function that takes into account the external load, the operational states of the nodes, and the management policies. All three mechanisms run independently and asynchronously on each node.

1) *Overlay Construction*: Nodes self-organize into a logical overlay network in which the links are application-level (TCP) connections. Nodes use the overlay to disseminate routing information and retrieve states/statistics from their neighbors.

For the purpose of our architecture, we searched for algorithms that produce a bi-directional overlay graph with a constant degree. The topological properties of such an overlay facilitate balancing the control load. They also simplify unbiased estimation of the system state from a subset of nodes, as the state of each node can be sampled the same number of times by other nodes and each node has a neighborhood of the same size for retrieving state information. In addition, we wanted to find algorithms that allow the overlay to regenerate quickly after failures and to optimize it dynamically according to specific criteria.

Our overlay construction mechanism includes two protocols that run independently of each other. First, an epidemic protocol (CYCLON [5]) maintains on each node an up-to-date cache of active nodes that can be chosen as neighbors. Second, a protocol (GoCast [6]) maintains the list of neighbors of a node and picks new neighbors from the cache as needed. (In our implementation, each node has four neighbors, while the size of the cache is 10.)

CYCLON maintains on each node a cache of size  $c$ . Each cache entry has the following format:  $\langle address, timestamp, state information \rangle$ . CYCLON periodically picks the cache entry with the oldest timestamp, and sends the content of the local cache to the node with that address. The remote node replies by sending a copy of its own cache. Both nodes then merge the two caches, sort the merged list according to the timestamps and keep the  $c$  most recent entries. It has been shown in [5] that the caches produced by CYCLON represent adjacency lists of a graph with random-graph properties.

The GoCast protocol [6] selects the neighbors of the local node from the entries of the CYCLON cache. GoCast ensures

TABLE I  
SAMPLE FORWARDING TABLE OF A NODE WITH LENGTH 3.

App ID	Nodes Offering Application		
App 1	10.10.1.9	10.10.1.72	10.10.2.33
App 2	10.10.1.99		
App 3	10.10.1.199	10.10.3.45	10.10.1.79
...	...	...	...
App m	10.10.1.232	10.10.2.40	

that the overlay converges to a graph where each node has approximately the same number of neighbors ( $\pm 1$ ), which is configurable. In addition, the protocol can continuously adjust the overlay graph following a specific optimization objective, such as finding overlay neighbors with lowest latency ([6]).

CYCLON and GoCast, in combination, achieve our design goals of fast regeneration of the overlay after failures and the possibility of dynamically optimizing the overlay topology according to specific criteria. In case of a node failure, a node can replace a failed neighbor with an active node from its cache. By examining the content of the *timestamp* and *state information* fields of the entries in its cache, a node can identify a set of candidate neighbors that improve the optimization criteria for the overlay.

The entry points are connected to the overlay in the following way. Each entry point runs an instance of the CYCLON protocol with a very large cache. It does not run GoCast, however. This ensures that each entry point is connected to a large number of active nodes, which facilitates balancing the load related to request routing. Also, the entry point does not appear as the neighbor of any node in the overlay.

2) *Request Routing*: Service requests enter the cluster through the entry points (see Fig. 1), which function as layer-4 switches. Each entry point is connected via logical links to a large number of nodes. An entry point directs incoming requests to these nodes using a forwarding policy. In our design, we use a round-robin forwarding policy.

Upon receiving a request from an entry point, a node determines its application type. The node processes the request if it runs the corresponding application and if its CPU utilization is below a configurable threshold  $cpu_{max}$ . Otherwise, the node consults its forwarding table and routes the request to a peer that offers the required application. In the absence of overload, a request will likely be routed once inside the cluster, since we believe that a single node will typically offer a small fraction of the applications in the cluster.

Table I shows an example of a node's forwarding table. It contains, for each application offered in the system, a configurable number of nodes that run that particular application. In Table I this number is 3. The forwarding tables are maintained through the mechanism discussed in Section II-C.2.

In order to avoid forwarding loops, a request header contains the addresses of the nodes that have routed the request. In addition, the number of times that a request can be forwarded is limited (to four in our design). The system drops the requests that exceed this limit.

*Selective Propagation of the Routing Updates*: As discussed in Section II-C.3, a node periodically computes the list of

applications that it will be running during the next placement cycle. After each such computation, the node advertises the list of applications to the other nodes, which update their forwarding tables. In this subsection, we propose a scheme for dynamically updating the forwarding tables in an efficient and scalable manner.

A straightforward solution to updating the forwarding tables is for each node to *flood periodically its application list* across the overlay. (This flooding scheme is similar to the propagation of link weight updates in OSPF.) Given the fact that each node generates one update per placement cycle and that the number of neighbors of a node is bounded by the connectivity of the overlay, a node processes  $O(N)$  update messages during each placement cycle, where  $N$  is the system size. Since the size of the application list is bounded, the length of an update message is bounded as well, and the processing load on a node required to update the forwarding table grows with  $O(N)$ .

In [2], we proposed an update scheme where the forwarding table on each node contains the list of applications that are running on every other node in the system. This table is called the *global placement matrix*. Maintaining a copy of the matrix at each node limits the scalability of the system, as the number of update messages increases linearly with the system size. The scheme enforces a maximum message rate on each overlay link, by buffering all the updates received by a node during a time interval  $\tau$  and aggregating them into a single message. Therefore, a node will process at most  $c/\tau$  update messages per second,  $c$  being the connectivity of the overlay. Note, however, that the size of an update message increases with  $O(N)$ , and therefore the processing load on a node increases linearly with  $N$ .

In this paper, we introduce an approach that involves *selective update propagation* to achieve a processing load on a node that is independent of the system size. The basic idea is that a node propagates further only those updates that cause a modification to its forwarding table.

Upon receiving an update message, a node modifies its forwarding table as follows. It extracts from the update four variables: the *sender-id* containing the address of the original sender node of the update, the *timestamp* indicating when the update was created by the sender, the *application-list* of the sender and the *distance* in overlay hops between the sender and the current node. For each application *app* in the list, the node performs the following operations. First, it searches for *app* in its forwarding table. If *app* is not found, the node adds the tuple  $\langle app, \langle sender-id, timestamp, distance \rangle \rangle$  to its table. (For reasons of simplicity, the sample forwarding table in Table I has a different format and does not include timestamp and distance.) Otherwise, if *app* is in the table, the node searches for an entry of the form  $\langle app, \langle sender-id, *, * \rangle \rangle$ . If such an entry exists, but it is less recent than the update, then its timestamp is updated. In case there is no entry of the form  $\langle app, \langle sender-id, *, * \rangle \rangle$  in the forwarding table, the node adds the tuple  $\langle app, \langle sender-id, timestamp, distance \rangle \rangle$  to its table, if there is space in the table. In case there is no space, the node overwrites an existing tuple if the update has

TABLE II  
PERFORMANCE COMPARISON BETWEEN THE SCHEMES FOR DISSEMINATING CONFIGURATION UPDATES.

	number of messages per node	message length	load per node
Flooding	O(N)	O(1)	O(N)
Global Placement Matrix	O(1)	O(N)	O(N)
Selective Propagation	O(A)	O(1)	O(A)

a smaller distance to the sender.

The update message to be further propagated by the node is constructed as follows. The node removes from the *application-list* of the received message each application that does not lead to a modification of its forwarding table. If the application list is empty, the node does not propagate the update any further. Otherwise, it sends the update to its neighbors, with the exception of the node where original the update came from.

For robustness purposes, the entries of the forwarding table are soft state, and each entry expires after a preconfigured timeout. For this reason, each node advertises its complete configuration after each placement cycle.

Regarding the performance of the selective update propagation, we note that the load on each node depends on the connectivity of the overlay  $c$  and on the number of applications  $A$ , but is independent of the system size  $N$ . A node processes  $O(A)$  update messages during a placement cycle. Since the size of the application list is bounded, the length of an update message is bounded as well, and the processing load on a node required to update the forwarding table grows with  $O(A)$ .

Table II compares the performance of the three schemes for propagating updates. The columns indicate the number of update messages that each node processes per placement cycle, the length of these update messages, and the processing load induced on a node by the update propagation, computed as the product of the first two columns.

While in the flooding scheme and the global placement matrix the processing load increases with the system size, in the selective update propagation scheme the load increases with the number of applications, but is independent of the system size. We can therefore conclude that *the selective propagation scheme scales with the system size*.

Note that for certain parameter ranges of the system size and the number of applications, flooding might perform similar to the selective update scheme. As our simulations show, this happens when a system that has a small size runs a large number of applications (see Fig. 6).

An interesting property of the selective dissemination scheme is that it distributes the nodes offering an application uniformly in the forwarding tables of the other nodes. We will study this property in more detail in future work.

3) *Application Placement*: The goal of application placement is to maximize continuously a utility function. The optimization process takes into account, for each application, the external demand for resources, the supply provided by each node and a utility parameter that captures the relative importance of each application. This approach allows the system to adjust dynamically its configuration to changes in the external load, or to a new management policy that changes

the relative importance of an application. Our approach to maximizing a global utility function is to perform a local optimization on each node, based on state information from its neighborhood. Our earlier work shows that such a heuristic can produce a solution that is close to optimal [1], [7].

We model the problem as follows. For a node  $n \in \mathcal{N}$ , let  $\Gamma_n$  and  $\Omega_n$  be its memory and CPU capacities, respectively. Let  $\mathcal{A}$  be the set of applications offered by the system and  $\mathcal{R}_n$  the set of applications that run on node  $n$ . (A node generally runs a small subset of all the applications offered by the system.)

For an application  $a \in \mathcal{A}$ , let  $\gamma_a$  be the memory demand of an instance of  $a$  and  $\omega_a^{demand}$  the total CPU demand for  $a$  in the system. Let  $\omega_{n,a}^{demand}$  be the CPU demand on node  $n$  for  $a$  and  $\omega_{n,a}^{supply}$  the CPU supply that  $n$  allocates to  $a$ . Each application  $a$  has a utility parameter  $u_a$  that defines its relative importance.

(The CPU demand and the CPU supply are measured in CPU cycles/second. We assume that all the nodes have the same CPU architecture and therefore an application request needs the same number of CPU cycles on all the nodes.)

We define the utility provided by a node  $n$  as the weighted sum of the CPU resources supplied to each application:  $utility_n = \sum_{a \in \mathcal{R}_n} \omega_{n,a}^{supply} u_a$ , and the system utility as:  $Utility = \sum_{n \in \mathcal{N}} \sum_{a \in \mathcal{A}} \omega_{n,a}^{supply} u_a$ .

We state the problem of application placement as follows:

$$\max \sum_{n \in \mathcal{N}} \sum_{a \in \mathcal{A}} \omega_{n,a}^{supply} u_a \quad (1)$$

such that

$$\forall n \in \mathcal{N} \quad \Gamma_n \geq \sum_{a \in \mathcal{R}_n} \gamma_a \quad (2)$$

$$\forall n \in \mathcal{N} \quad \Omega_n \geq \sum_{a \in \mathcal{R}_n} \omega_{n,a}^{supply} \quad (3)$$

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.

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 locally, as well as 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. We give the pseudo-code of the algorithm in Fig. 3 and describe each of its phases below.

```

1. class AppInfo {
2.     string app_id;
3.     double cpu_demand, cpu_supply, utility_parameter;
4. }
5. List<AppInfo> active_apps, standby_apps, new_active_apps, all_apps;
6. double max_utility, utility;

7. while(true) {
8.     active_apps=getLocalActiveApps();
9.     all_apps=getApps(forwarding_table);
10.    standby_apps=all_apps-active_apps;
11.    neighbors=getOverlayNeighbors();
12.    standby_apps=getAppStats(neighbors, standby_apps);
13.    active_apps=sortIncreasing(active_apps);
14.    standby_apps=sortDecreasing(standby_apps);
15.    new_active_apps=active_apps;
16.    max_utility=currentUtility()
17.    for(i=0..active_apps.size()) {
18.        utility=transferResources(top i active_apps, standby_apps);
19.        if(utility>(max_utility+change_cost)) {
20.            max_utility=utility;
21.            new_active_apps=active_apps-top_i_active_apps+sel_standby_apps;
22.        }
23.    }
24.    advertise(new_active_apps);
25.    if(new_active_apps!=active_apps)
26.        stopAndStartApps(active_apps, new_active_apps);
27.    wait end of placement cycle;
28. }

```

Fig. 3. The pseudo-code of the application placement mechanism.

*Phase 1: Gathering State Information.* A node retrieves the set of active applications  $\mathcal{R}$  that it is currently offering and it gathers statistics (CPU supply and demand) for each application in  $\mathcal{R}$  (line 8). A node also retrieves from its forwarding table a list of all the applications offered in the system (line 9). Any application that is offered by the system, but is not currently active on the node becomes a potential candidate to be activated on the node during the current placement cycle and is added to a standby set  $\mathcal{S}$  (line 10). A node retrieves from each overlay 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  $(\omega_{x,a_1}^{supply}, \dots, \omega_{x,a_m}^{supply})$  delivered to those applications, and the CPU demands of those applications  $(\omega_{x,a_1}^{demand}, \dots, \omega_{x,a_m}^{demand})$ . In addition, neighbor  $x$  also reports the locally measured demands for applications it could not route, since they are not offered in the system (lines 11-12). (A high demand or utility for these inactive applications might trigger their activation during the next placement cycle.)

*Phase 2: Computing a New Set of Active Applications.* The placement algorithm attempts to replace a subset of applications in  $\mathcal{R}$  with a subset of applications in  $\mathcal{S}$ , so that the utility of the node 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 13-23).

On a node  $n$ , we remove from  $\mathcal{S}$  all the applications for which the supply matches the demand. The active applications in  $\mathcal{R}_n$  are sorted in increasing order of their utility which we define as the load delivered by  $n$  to application  $a$ , multiplied by the utility coefficient of  $a$ , i.e.,  $u_a \omega_{n,a}^{supply}$ . The applications in the standby set  $\mathcal{S}$  are sorted in decreasing order of the amount

that the node would add to its utility in the case when it would fulfill the unsatisfied demand for those applications, i.e.,  $u_a \sum_{n \in neighbors} (\omega_{n,a}^{demand} - \omega_{n,a}^{supply})$ . Intuitively, the placement controller tries to replace low-utility applications from  $\mathcal{R}$  with high-utility applications in  $\mathcal{S}$ , so that the local utility 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.,  $\Gamma_n^{free} > 0$  and  $\Omega_n^{free} > 0$ ), 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 utility.

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  $\{\mathcal{R}^k | k = 0, 1, \dots, |\mathcal{R}|\}$  the configuration that maximizes the

utility of the node.

The algorithm computes the cost of stopping and starting applications (the *change\_cost* variable, defined in line 19) by multiplying the current utility of the node by the time needed to start and stop the applications computed in this phase. In this way, a node that delivers a low utility is more likely to change its configuration.

*Phase 3: Reconfiguring and Advertising the New Configuration.* The algorithm advertises its current configuration (line 24) and switches from the old configuration to the new configuration by stopping and starting applications (line 25).

### III. SYSTEM EVALUATION THROUGH SIMULATION

#### A. Evaluation Setup

We evaluate our system through simulation according to five criteria. *Efficiency* captures the capability of the system to operate exhibiting high performance parameters in a steady state. *Scalability* captures the capability of the system to maintain similar performance characteristics when its size increases. *Adaptability* captures the capability of the system to respond to a change in the operating conditions by reconfiguring and converging to a new steady state. *Robustness* captures the capability of the system to respond to node arrivals, departures and failures, by reconfiguring and converging to a new steady state. In this paper, we understand *manageability* as the capability of the system to adjust its configuration to changes in management policies.

The simulated system is written in Java and uses javaSimulation [8], a package that provides a general framework for process-based discrete event simulation. (We also implemented our design on a testbed using Tomcat [9]. Measurements and experiences from the testbed will be reported elsewhere.)

We simulate a node as having two FIFO message buffers that store the service requests and the control messages, respectively. Every 10 ms the node processes all messages in these two buffers. Therefore, each message experiences a delay between 0 and 10 ms, before being processed. In the case of the request buffer, the node moves a request that it can serve locally into its list of active requests being executed. This list has a capacity of 50, which means that a node can concurrently execute 50 service requests. As a request has an average execution time of 250 ms (see below), the service capacity of a node is 200 reqs/sec. If a service request cannot be executed locally, it is forwarded to another node or dropped. The memory capacity  $\Gamma_n$  of a node is uniformly distributed over the set  $\{1, 2, 3, 4\}$  GB.

For the simulations, the size of the cluster varies between 50 and 800 nodes, and there is one entry point per 200 nodes. If a system has multiple entry points, the request generator distributes the external load equally among them. In all experiments, we set the cache size for CYCLON to 10 and the target number of overlay neighbors for GoCast to 4. A node gathers state information from all the nodes within two overlay hops (a maximum of 17 nodes for a network where each node has 4 neighbors). Each node runs CYCLON every 5 sec, GoCast every 1 sec and application placement every

30 sec. A node limits the size of its forwarding table to four nodes for each application, unless otherwise specified.

We assume that a request for each application has the same average execution time of 250 ms. On a node, the execution times of the requests follow an exponential distribution with  $\lambda = 4sec^{-1}$ . We assume further that the memory requirement  $\gamma_a$  for an instance of application  $a$  is uniformly distributed over the set  $\{0.4, 0.8, 1.2, 1.6\}$  GB.

We have conducted simulations with a load pattern based on the power-law distribution where applications are ranked according to their popularity and the application with rank  $a$  is assigned a value proportional to  $a^{-2.16}$  [10]. We obtain the average external load (measured in CPU cycles/second) for the application with rank  $a = 1, 2, 3 \dots$  by choosing a random weight in the interval  $[0, a^{-2.16}]$ , normalizing the resulting distribution, and multiplying the distribution with the total external load. The arrival of individual service requests is modeled as a Poisson process.

Each simulation run lasts 600 sec. We start measuring after 210 sec, which represents the warm-up phase for the simulation. Each point in Figs. 4-9 represents the average of 20 simulation runs.

For all experiments except manageability, we assign the same utility parameter to all applications, which means that the global utility function that the system attempts to maximize is the utilization of its combined CPU resources.

In all experiments, we use three output metrics to measure the system performance. *Satisfied Demand* is the ratio between the CPU resources supplied by the system to the applications and the total CPU demand (which we also call the external load). Since we assume that a request for each application takes, on average, the same amount of CPU resources, the satisfied demand can be understood as the rate of the executed requests divided by the rate of the incoming requests, i.e., the ratio between service rate and arrival rate. A satisfied demand of 1, for instance, means that the system processes all the incoming requests. One minus the satisfied demand gives the fraction of the requests that are dropped by the system. The *number of configuration changes* is the (average) number of application-start and application-stop operations that a node performs during one placement cycle. The *control load* gives the (average) number of control messages received by a node in one second. The messages are generated by the control mechanisms of the system, namely, overlay maintenance, dissemination of the routing updates, and retrieval of the neighborhood state.

#### B. Evaluation Scenarios

1) *System Efficiency:* We assess the system efficiency for various intensities of the external load. The load intensity is represented using the *CPU load factor* (CLF), as defined in [3]. The CLF is the ratio between the total CPU demand of the applications and the total CPU capacity available in the system:  $CLF = \sum_{a \in \mathcal{A}} \omega_a^{demand} / \sum_{n \in \mathcal{N}} \Omega_n$ .

Fig 4 shows the system performance for different values of CLF up to 2.0, which corresponds to an arrival rate that

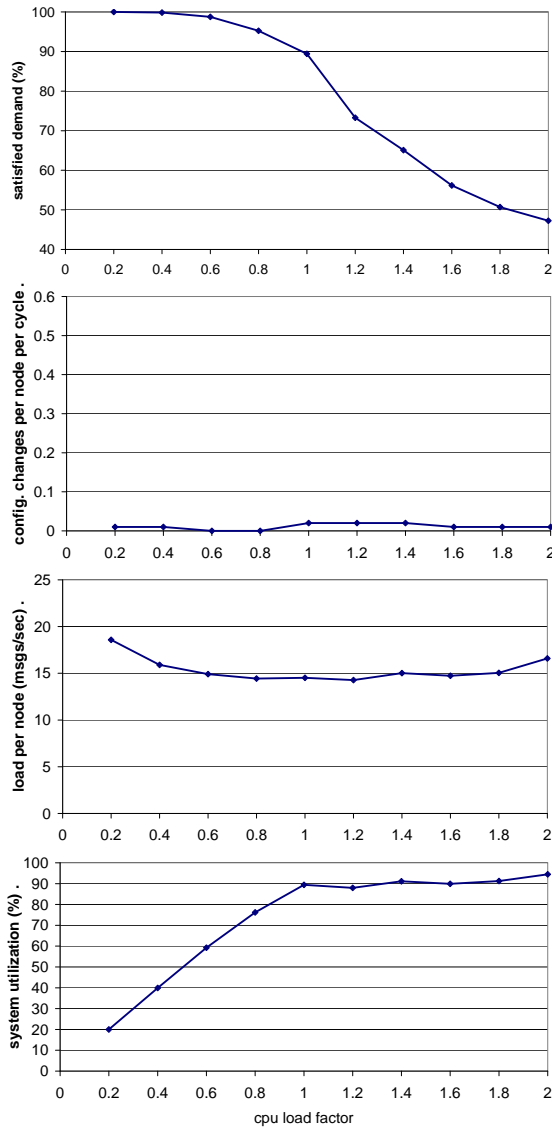


Fig. 4. System efficiency as a function of the CPU load factor (CLF).

is double of the maximum service rate. The system under evaluation has 200 nodes and runs 100 applications.

As expected, the satisfied demand decreases and the number of configuration changes increases when the external load increases. The system utilization, computed as the product of satisfied demand and CLF, increases almost linearly until a CLF of 0.6, then flattens out and becomes almost constant for  $CLF > 1.2$ .

The maximum (average) utilization that our system achieves under these conditions is about 95%. The main limiting factor of achieving higher utilization stems from memory fragmentation. According to our simulation configuration, some applications, which require a large amount of memory (1.2GB or 1.6GB), cannot run on certain nodes (that have only 1GB of memory). This reduces the number of nodes that can run these applications. While an ideal system (i.e., a centralized system that has the combined resources of all the nodes in the simulated system) can generally achieve a utilization of 100%, memory fragmentation limits in a similar way the performance

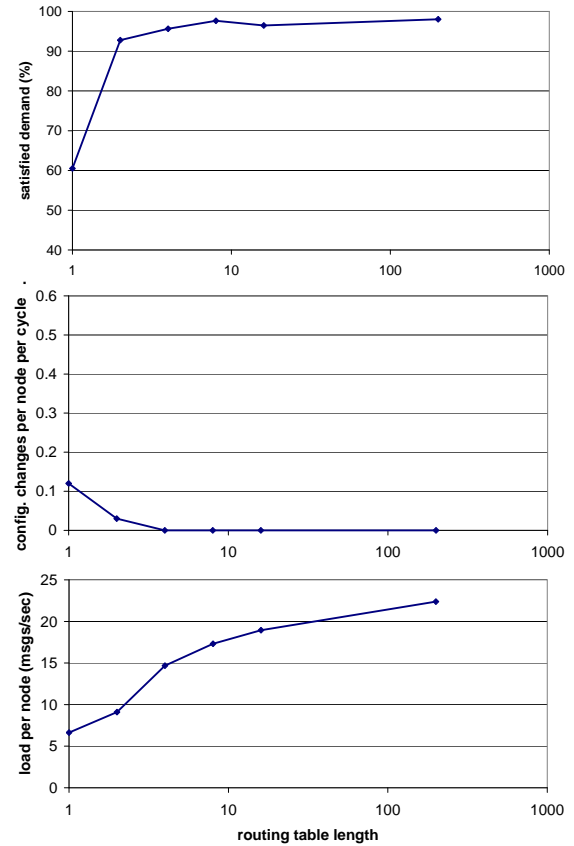


Fig. 5. Impact of the forwarding table size on the system efficiency.

of a centralized resource controller that manages all the system nodes.

An unexpected result is that the control load per node decreases with the external load. The fraction of the control load that changes with the external load stems from routing updates (see Section II-C.2). We explain the measurement result by the way in which the placement algorithm chooses the set of active applications on a node. Since it sorts the applications in the order of their decreasing unsatisfied demand, it favors high-demand applications in its decisions. Consequently, low-demand applications tend to be stopped and, therefore, the number of active applications in the system tends to decrease as the external load increases. The decrease in active applications means a smaller number of routing updates, and, therefore, a lower control load. To avoid starvation of the low-demand applications, a user can increase the utility parameters for these applications, as described in Section III-B.6.

2) *Impact of the Forwarding Table Size on the System Efficiency:* In this set of experiments, we measure the system performance while varying the size of the forwarding table. These measurements allow us to find a tradeoff between the control load for update dissemination on the one hand, and satisfied demand and configuration changes on the other.

We run the experiments for a system with 200 nodes. Fig. 5 gives the results for the case where the length of each row in the forwarding table is 1, 2, 4, 8, 16 and 200.

We observe an initial significant gain in satisfied demand, from a length of 1 to a length of 4, above which the gain in

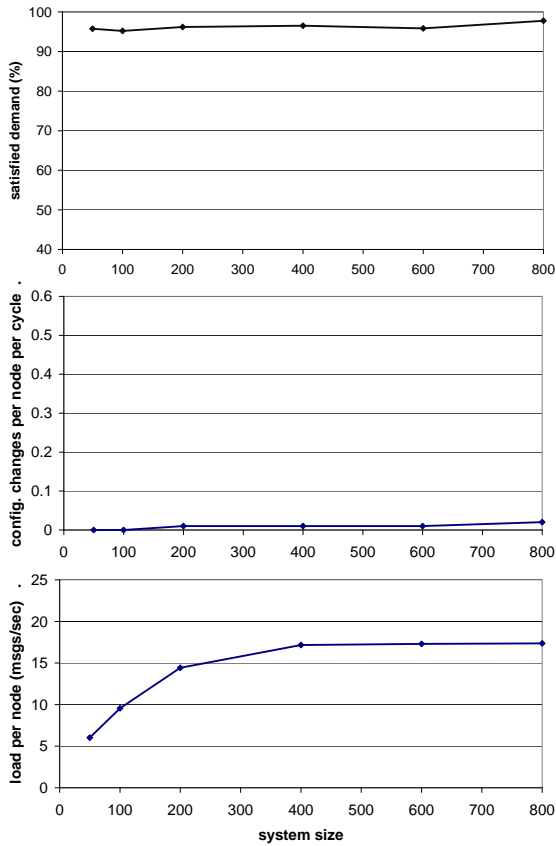


Fig. 6. System scalability.

satisfied demand becomes very small. As expected, the number of configuration changes decreases with the length, while the control load increases with the length. These experiments led us to choose 4 as the length of the rows in the forwarding table.

3) *System Scalability*: To assess the system scalability, we vary the system size from 50 to 800 nodes, while the number of applications is set to 100. Fig 6 shows the results of these experiments.

We observe that the satisfied demand and the (average) number of configuration changes per node do *not* depend on the size of the system, within the parameter range we are measuring. This shows that request routing effectively balances the load for all system sizes considered.

The control load per node increases up to a system size of 200 nodes, then flattens and slightly decreases. We explain the initial increase in load by the fact that our approach, selective update dissemination, performs as well as flooding, as long as the system size is comparable to the number of applications (see Section II-C.2). When the system size increases above 200 nodes, our approach reduces the load compared to flooding. As we argued in Section II-C.2, for a large system size, the control load does not depend on the number of nodes, but only on the number of applications. This leads us to the conjecture that this system can scale to any size regarding the performance metrics considered here, as long as the number of applications remains bounded. This is a useful feature for systems with a large number of machines and only a limited number of

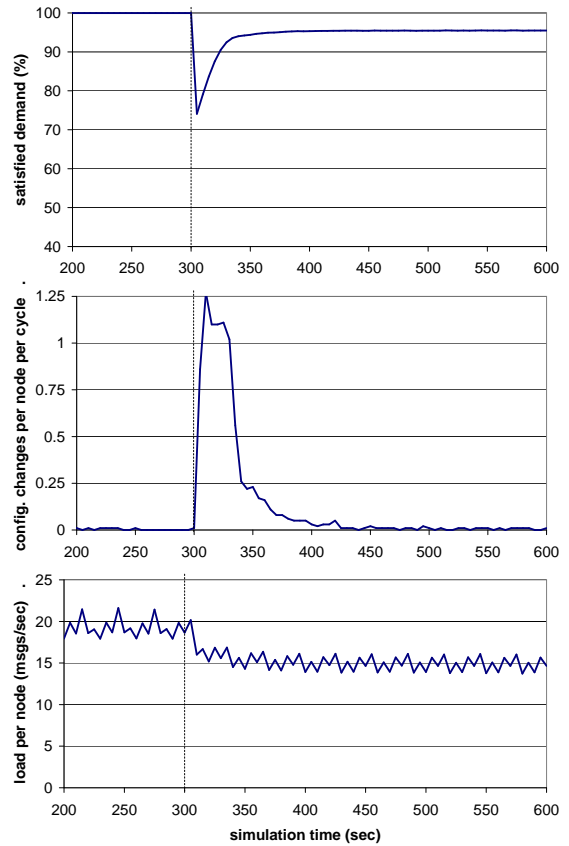


Fig. 7. System adaptability: reaction to addition of new applications.

applications, such as online auction sites.

4) *System Adaptability*: We evaluate the system adaptability for the case where new applications and additional load are added to a system with 400 nodes. Initially, the system is in steady state, offering 50 applications and having an external load of  $CLF=0.4$ . After 300 sec, we add instantly 50 new applications that induce an additional load of  $CLF=0.4$  in the system, which brings the total load to  $CLF=0.8$ . Fig 7 shows the result of this experiment.

As expected from Fig. 4, after adding new load to the system, the satisfied demand decreases, the number of configuration changes increases and the load per node decreases. After the system reaches a steady state, the performance metrics show values that are consistent with the measurements in Fig. 4.

Surprisingly for us, we observe that all three output metrics show different settling times. The satisfied demand converges in about two placement cycles, the number of configuration changes in four, and the control load on a node in about seven. We explain this divergence by the imperfection in the decentralized placement, where nodes continue making changes to their local configuration without improving the satisfied demand.

5) *System Robustness*: We evaluate the system robustness for the case where a subset of the nodes in the system fails. The initial size of the system is 400 nodes and the external load is  $CLF=0.5$ . We run two experiments in which a system in steady state experiences node failures after 300 sec. In the first

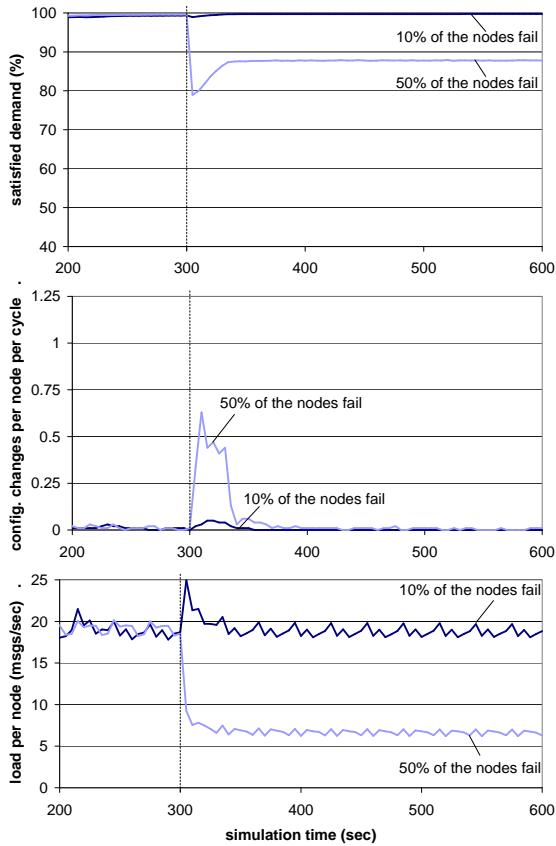


Fig. 8. System robustness: reaction to failures.

experiment, 10% of the nodes fail. As a result, the amount of system resources decreases and the load becomes  $CLF=0.55$ . In the second experiment, 50% of the nodes fail and the load thus becomes  $CLF=1.0$ . Fig. 8 shows the system reaction for both experiments.

After the failures, the system reaches a new steady state. We observe that, if 10% of the nodes fail, the output metrics are approximately the same as before the failures. In the case where 50% of the nodes fail, the output metrics change significantly. We explain the difference in the outcome of these two experiments with the fact that, in the first case, the load increases from  $CLF=0.5$  to  $0.55$ , while, in the second case, the load increases from  $CLF=0.5$  to  $1.0$  (which means overload). Note that these output metrics are consistent with the values in Figs. 4 and 6.

The system recovers surprisingly fast and reaches a steady state, even after massive failures. We explain this behavior by the fact that the overlay construction mechanisms operate on a faster timescale than the application placement (1 sec for Gocast and 5 sec for CYCLON, vs. 30 sec for the application placement). Therefore, with very high probability, the overlay is rebuilt completely within one placement cycle after a failure. The rebuilt overlay enables all nodes to function properly again, specifically, to receive routing updates and state information from their neighborhood. As the overlay recovers fast, the system reacts to a failure similarly as it would to a sudden increase in load.

As in the case of Fig. 7, all the three output metrics

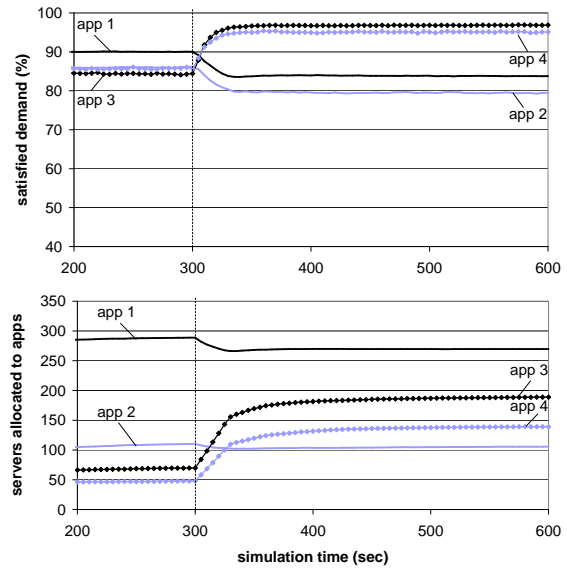


Fig. 9. Service differentiation.

have different convergence times, and the satisfied demand converges first, the number of configuration changes second, and the control load third.

6) *System Manageability*: The management policy in this scenario enables the system administrator to provide differentiated service by changing, at runtime, the utility parameter for each application (see Section II-C.3). Increasing the utility parameter of a specific application results in an increased satisfied demand for that application and its deployment on a larger number of nodes. (We define the satisfied demand of an application as the ratio between the rate of serviced requests divided by the request arrival rate for that application.)

For this experiment, a system with 400 nodes offers 50 applications, and the external load is  $CLF=1$ . Initially, all the applications have the same utility parameter, equal to 1. After 300 sec, the utility parameter of the applications 3 and 4 is increased from 1 to 10.

Fig. 9 illustrates how the system reacts to this change. We show the satisfied demand for the applications 1, 2, 3 and 4 (out of all 50 applications). Applications 1 and 2 do not experience an increase in utility, while applications 3 and 4 do. The figure shows how the changes of the utility parameters affect the satisfied demand and the number of nodes on which each application is active.

We observe that the change in the utility parameters significantly increases the satisfied demand for applications 3 and 4, while the satisfied demand for applications 1 and 2 decreases. (The average satisfied demand in the system is around 87%). At the same time, the number of nodes on which applications 3 and 4 are active increases, while the number of nodes on which applications 1 and 2 are active decreases.

We conclude that our system can effectively provide service differentiation in an overload scenario, and that the system administrator can control the service differentiation through the utility parameters. Increasing the utility parameter for an application usually does not only increase its satisfied demand,

but also its resilience to failures, since the number of nodes on which it is active is increased.

Note that predicting the satisfied demand for a given utility parameter is difficult to achieve, since it depends on the demand for all the applications in the system. The current design allows increasing the satisfied demand for a specific application only up to a certain point that depends on the external load. To achieve a 100% satisfied demand for a specific application, changes to the scheduling mechanism on the nodes are needed.

#### IV. RELATED WORK

The application placement problem, as described in Section II-C.3, is a variant of the class constrained multiple-knapsack problem, which is known to be NP-hard [12]. 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 [3].

Stewart et al. [11] present a centralized method for automatic component placement that maximizes the overall system throughput. In the area of content delivery and stream processing, [13], [14], [15] 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, [16] presents a decentralized placement algorithm that clusters application components according to the amount of data they exchange.

#### V. DISCUSSION AND FUTURE WORK

In this paper, we have presented a peer-to-peer design for a self-organizing middleware in support of application services. Our design manages several types of system resources, including CPU and memory. It scales in terms of nodes and supports a large number of applications. We use three decentralized control mechanisms in our design: overlay construction, request routing and application placement, each of which runs on a different timescale to achieve its own objective.

We highlight here two key results derived from the evaluation of the design through simulation. First, we have argued that the selective propagation of routing updates introduced in this paper allows the system to scale to any size. This is because the message load on a node, induced by all control mechanisms, does not increase, as long as the number of applications remains bounded. We have verified through simulation that a system can scale from 200 to 800 nodes without performance degradation in satisfied demand, number of configuration changes and load per node.

Second, service differentiation in our system can be controlled effectively through changing the values of the utility parameters on the application placement controller. Increasing the utility parameter for a specific application increases the satisfied demand for that application, which means that fewer requests for it are dropped. Further, the resilience of the

application to node failures increases, as more nodes start offering it. We have demonstrated service differentiation under overload conditions (see Fig. 9).

We are in the process of implementing and evaluating the design described in this paper in the Tomcat environment. The request routing mechanism runs as an application filter in Tomcat, following the approach we described in [17]. We are evaluating the performance of the system using the TPC-W and RUBiS benchmarks.

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.

#### VI. ACKNOWLEDGMENTS

This work was supported by the Graduate School of Telecommunications (GST) at KTH and an IBM Faculty Award.

#### REFERENCES

- [1] C. Adam and R. Stadler, "A Middleware Design for Large-scale Clusters Offering Multiple Services", *IEEE eTransactions on Network and Service Management (eTNSM)*, Vol. 3, No. 1, 2006
- [2] C. Adam, G. Pacifici, M. Spreitzer, R. Stadler, M. Steinder, C. Tang, "A Decentralized Application Placement Controller for Web Applications", IBM Tech. Report RC23980, June 2006.
- [3] 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.
- [4] Li Wang: "TCPHA Software", <http://dragon.linux-vs.org/dragon-fly/htm/tcpa.htm>, August 2006.
- [5] S. Voulgaris, D. Gavidia, M. van Steen., "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays", *Journal of Network and Systems Management*, Vol. 13, No. 2, June 2005.
- [6] C. Tang, R. N. Chang, and C. Ward, "GoCast: Gossip-enhanced Overlay Multicast for Fast and Dependable Group Communication", *DSN'05*, Yokohama, Japan, June 2005.
- [7] B. Johansson, C. Adam, M. Johansson, R. Stadler, "Distributed Resource Allocation Strategies for Achieving Quality of Service in Server Clusters", 45-th IEEE Conference on Decision and Control (CDC-2006), San Diego, USA, December 13-15, 2006.
- [8] K. Helsgaun, *Discrete Event Simulation in Java*, Writings on Computer Science, 2000, Roskilde University.
- [9] Apache Tomcat, <http://tomcat.apache.org>, February 2006.
- [10] L.A. Adamic, B.A. Huberman, "Zipf's law and the Internet", *Glottometrics* 3, 2002.
- [11] C. Stewart K. Shen, S. Dwarkadas, M. Scott, *Profile-driven Component Placement for Cluster-based Online Services*, IEEE Distributed Systems Online, Volume 5, Number 10.
- [12] H. Kellerer, U. Pferschy, D. Pisinger, *Knapsack Problems*, Springer-Verlag, 2004.
- [13] S. Buchholz and T. Buchholz, "Replica placement in adaptive content distribution networks", *ACM SAC 2004*, Nicosia, Cyprus, March 2004.
- [14] K. Liu, J. Lui, Z-L Zhang, "Distributed Algorithm for Service Replication in Service Overlay Network", *IFIP Networking 2004*, May, 2004, Athens, Greece.
- [15] 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.
- [16] C. Low, "Decentralized application placement", *Future Generation Computer Systems* 21 (2005) 281-290.
- [17] C. Adam and R. Stadler, "Implementation and Evaluation of a Middleware for Self-Organizing Decentralized Web Services", *IEEE SelfMan 2006*, Dublin, Ireland, June 2006.