Self-Organized Service Placement in Ambient Intelligence Environments

KLAUS HERRMANN
University of Stuttgart

Ambient Intelligence (AmI) is an IT concept by which mobile users shall be seamlessly supported in their everyday activities. This includes interactions with remote resources as well as with their current physical environment. We have developed the so-called Ad hoc Service Grid (ASG) infrastructure that supports the latter form of interactions. It allows operators to cover arbitrary locations with ambient services in a drop-and-deploy fashion. An ambient service may autonomously distribute (replicate and migrate) within an ASG network to optimize its availability, response times, and network usage. In this article, we propose a fully decentralized, dynamic, and adaptive service placement algorithm for AmI environments like the ASG. This algorithm achieves a coordinated global placement pattern that minimizes the communication costs without any central controller. It does not even require additional communication among the replicas. Moreover, placement patterns stabilize if no changes occur in the environment while replicas still retain their ability to adapt. Mechanisms for self-organized placement of services are very important for AmI environments in general since they allow for autonomous adaptations to dynamic changes and, thus, remove the need for manual (re)configuration of a running system. We present a detailed evaluation of the algorithm's performance and compare it with three other algorithms to show its competitiveness. Furthermore, we discuss how the desired self-organizing behavior emerges from the interactions of a few simple, local rules that govern the individual placement decisions. In order to do so, we give an in-depth analysis of a series of emergent effects that are not directly encoded into the placement algorithm but stem from its collective dynamics.

Categories and Subject Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed applications*; H.1.0 [**Information Systems**]: Models and Principles—*General*; H.4.3 [**Information Systems Applications**]: Communications Applications

General Terms: Algorithms, Management

Additional Key Words and Phrases: Self-organization, emergence, service placement, ambient intelligence

ACM Reference Format:

Herrmann, K. 2010. Self-Organized service placement in ambient intelligence environments. ACM Trans. Auton. Adapt. Syst. 5, 2, Article 6 (May 2010), 39 pages.
DOI = 10.1145/1740600.1740602 http://doi.acm.org/10.1145/1740600.1740602

Author's address: K. Herrmann, Institute of Parallel and Distributed Systems (IPVS), University of Stuttgart Universitätsstr. 38, D-70569 Stuttgart; email: klaus.herrmann@acm.org.

DOI 10.1145/1740600.1740602 http://doi.acm.org/10.1145/1740600.1740602

1. INTRODUCTION

In recent years, the vision of Ambient Intelligence (AmI) [Ducatel et al. 2001] has produced considerable research efforts. The main idea of AmI is that mobile users may wirelessly access services (anywhere and anytime) that support them in their daily activities without putting any administrative burden on them. Ultimately, AmI shall seamlessly integrate remote access to personalized services and data (e.g., related to workspaces or private homes) with access to services that facilitate the interaction with the immediate physical environment of the mobile user (e.g., an airport or a conference venue). The necessary computing infrastructures shall hide in the background. In this respect, AmI bears some similarities to the more established concept of ubiquitous computing [Weiser 1991].

One essential building block of AmI are infrastructures that allow local access to local, facility-specific services. To enable services that enhance the user's interaction with his current environment, the physical surrounding of the user must be enriched with computing resources that enable service provisioning. As the user enters a physical location, he is able to employ these services to make use of the local resources more effectively and more efficiently. One example scenario is a shopping mall that offers ambient services to customers, enabling them to navigate through the mall, find certain products quickly, and optimize the contents of their shopping cart, for example, according to the overall price or quality.

We have developed the so-called *Ad hoc Service Grid* (ASG) infrastructure [Herrmann et al. 2004] that enables the easy and flexible deployment of this kind of infrastructure. The ASG is based on multipurpose computers (so-called *service cubes*) being distributed in such a way that they can set up a wireless ad hoc network and cover the respective facility (e.g., the shopping mall). Mobile users can access this network via their mobile devices and use its resources. Service cubes represent modular building blocks offering both networking and computing resources for providing ambient services to mobile users. An ASG can quickly be scaled to the desired size by adding, removing, or repositioning individual service cubes. The basic concept that we are trying to realize with the ASG may best be characterized as drop-and-deploy: Apart from the physical setup of the infrastructure (the distribution of service cubes), all other aspects of operating the system shall be organized by the ASG serviceware without requiring excessive manual intervention. One particular challenge in this respect is the adaptive placement of services: Services are injected into the ASG at arbitrary service cubes and shall replicate and position themselves such that the overall network load and the client response times are reduced. Replication is an important aspect of the ASG since it increases availability (even if the network is partitioned), distributes processing load, and reduces network load since client requests can be served by a nearby service replica instead of being routed through the whole network. In order to enable the drop-and-deploy model, creating an appropriate amount of replicas for a service and positioning them adequately must be a self-organizing process that does not require manual control. Moreover, an ASG undergoes different kinds of dynamic changes that require adaptation. Therefore, service placement must be a continuous adaptive process.

In this article, we introduce a self-organizing service placement system that fulfills these requirements. Services are implemented as mobile software agents that are free to replicate (create clones) and migrate (move among service cubes). They inspect local request traffic patterns and take local decisions concerning replications and migrations. As a collective result of these local actions, a cost-efficient global placement pattern evolves that is able to adapt to dynamic changes. We describe the distributed algorithm that is at the heart of the placement system and analyze the key emergent features that drive the adaptation process. Furthermore, we evaluate its effectiveness compared with other algorithms.

This article is partially based on prior publications [Herrmann 2007a, 2007b, 2006; Herrmann et al. 2005, 2004].[1] The major novel aspect is the in-depth presentation of the replica placement algorithms, including a generalized version. Furthermore, we present a detailed quantitative and qualitative analysis. The presentation of the wider context of the work is also novel, and much of the material has been updated to reflect new results gained.

The rest of the article is structured as follows. In Section 2, we describe possible application scenarios to motivate our approach. Section 3 characterizes the basic problem of placing service replicas in an ASG. The proposed placement algorithm is covered in detail in Section 4. Subsequently, we first analyze the emergent collective behavior of the ASG replica placement system in Section 5, before we give a quantitative evaluation of its performance and compare it to other algorithms in Section 6. This is followed by a brief discussion about an improved version of the basic algorithm in Section 7. In Section 8, we discuss related work in the area of service and resource placement. Finally, we state our conclusions and give an outlook on future work in Section 9.

2. PROTOTYPE APPLICATION SCENARIOS

A typical application scenario for the ASG technology is represented by a shopping mall. Modern malls offer many different goods and services for the customer. This ranges from normal shops, to restaurants, cinemas, fitness studios, and beauty salons. They are typically arranged on several floors with an area of several thousand square meters each. Thus, a mall is a complex environment. Electronic services being provided by the shopping mall to customers can help in dealing with this complexity. For example, a *navigation service* may guide customers through the mall. A *shopping guidance service* may match the customers' electronic shopping lists with the information provided by a *product information service* to find the best products according to criteria predefined by the customer (e.g., lowest prices, highest quality, etc.). Together, these services may compute the fastest route through the mall covering all the products on a shopping list. *Reservation services* may allow the customer to reserve a table

[1]Some of the figures and graphs are adopted from these publications without further notice in the text.

at a restaurant or movie tickets from any location in the mall. Thus, the mall environment may be enriched with new value-added services to make it more attractive and enjoyable for different groups of customers.

However, based on the computing devices that already exist in such an environment, the provisioning of complex ambient services is seldomly possible. This becomes even more evident in other possible application scenarios that are of a more temporary and spontaneous nature. For example, a trade fair may be equipped with an ASG to support visitors. Similarly, music concerts and sporting events may benefit from such a system. Therefore, a dedicated computing and communication substrate is required, and the ASG represents an infrastructure for providing such a substrate.

There are two obvious classical alternatives to this kind of service provisioning: WLAN access points and cellular mobile phone networks. IEEE802.11 (WLAN) access points can cover a limited area and let people gain access to the Internet. Cellular phone networks like GSM and UMTS provide ubiquitous access to global resources. However, it turns out that both WLAN and cellular phone networks fail to provide a valid coverage technology for local services at medium-sized locations such as shopping malls, hospitals, and construction sites which span areas of a few thousand square meters. Such locations are too large to be covered efficiently with WLAN. The main reason for this is that, in a standard setup, IEEE802.11 access points need to be wired to some networking infrastructure. Here, the paradox situation occurs that the wiring of several wireless access points costs large amounts of money. This was reported in 2001 by the company lesswire who covered a hall at the CeBIT fair with Bluetooth access points [Kraemer and Schwander 2003]. Other experiments with WLAN also resulted in the costs for wiring exceeding the costs of the wireless equipment by far. Moreover, wired access points limit the flexibility of the network since extending or restructuring the network is cumbersome.

The alternative of using mobile phone networks for service provisioning introduces another problem. The services that are subject to the coverage are in most cases local by nature. For instance, in a shopping mall a shop owner wants to provide a product information service, or a music store offers excerpts of current chart hits for download and lets the customer compose his own CD. Using these services requires communication between two parties in close physical proximity at the same location (the shopping mall). However, using mobile phone networks implies global communication at a relatively high price and at a comparably low data rate (even with UMTS). Therefore, they seem inappropriate too.

In the proposed ASG system, mobile ad hoc networking (MANET) technology is used for the basic communication, as indicated in Section 1. Apart from the fact that this technology scales well to the envisioned location sizes, it can be provided at zero cost and with relatively high bandwidth. Furthermore, it fits the spontaneous nature of the drop-and-deploy vision perfectly. Since the core network of service cubes is relatively static, established and robust routing and transport mechanisms for MANETs can be easily employed as detailed in Section 3.2.

3. THE PLACEMENT PROBLEM

In this section, we introduce the basic placement problem. First, we state the requirements posed by the ASG environment before we present some central assumptions of our system. Then, we investigate a closely related NP-hard problem from graph theory to motivate our heuristic approach. Finally, we present the cost model that is used later on for the evaluation.

3.1 Requirements

The basic requirements for a replica placement algorithm in the ASG context are the following.

—*No External Control*. The replica placement system must not require any control actions from the outside. Users as well as operators must be relieved from the burden of taking placement decisions.

—*Decentralization*. The ASG does not possess a central control entity. This ensures its robustness, availability, and scalability. Therefore, the placement system must also operate in a decentralized fashion.

—*No Global Knowledge*. The lack of a central controller implies that any form of global knowledge (e.g., about the network topology or client behavior) is hard to obtain and maintain. Additionally, the system must operate online, that is, it has only knowledge about past events.

—*Creation of a Coherent Global Placement Pattern*. Despite the lack of external control, centralized components, and global knowledge, the system must achieve an overall placement pattern that optimizes costs on a global scale.

—*Adaptivity*. Whenever relevant changes happen in the ASG, the placement system must be able to adapt by replicating and repositioning replicas as needed in order to optimize the fitness of the overall system.

—*Stability*. The system must not run into some form of unstable behavior (e.g., oscillations) due to its continuous effort to adapt.

—*Optimize Communication Costs*. As the fundamental optimization criterion, we define the overall communication cost produced in an ASG. Since the shared wireless network medium is the most critical resource in the ASG, the primary objective of service placement is to reduce the network traffic as much as possible. This frees network resources and increases the overall service provisioning capacity. Moreover, client response times are potentially reduced since they tend to depend heavily on the wireless transmission medium.

3.2 System Model

As a model for the network connecting the service cubes, we assume a simple unit disk graph $G = (N, E)$. N is the set of service cubes, and E is the set of bidirectional wireless links connecting the cubes. This assumption does not hold for arbitrary wireless ad hoc networks due to antenna characteristics and interference. However, the ASG network is set up purposefully, and service cubes are moved only infrequently. Therefore, we assume that the setup is done in a way such that most links are bidirectional. The remaining ones can be detected and sorted out.

A *service* or *service type* is represented by one or more replicas in the system. Thus, the service is only a type information (e.g., “Navigation Service”). The physical representation is given by the replicas. A specific service is brought into the ASG network by injecting a single initial replica at some arbitrary node. Then, this replica uses successive replication and migration to produce an appropriate service distribution. The initial replica is never removed from the system as long as the service as such is active. Thus, if a service is not used for an extended period of time and idle replicas remove themselves from the system, there is always at least the initial replica that is still active.

Clients are assumed to be mobile and equipped with handheld devices (e.g., PDAs). These devices have the same transmission range as service cubes and establish a connection to the nearest service cube automatically. This implies that a client is always connected to at least one service cube.

A client finds an adequate replica by querying a specifically designed lookup service [Herrmann et al. 2005]. The lookup service itself is distributed and keeps track of the locations of all active replicas. The way in which lookup service instances are distributed guarantees that each client has an instance in its one-hop neighborhood. Upon a client query, a lookup service instance returns the closest (in terms of network hops) running replica. The client may then send requests to this replica which are routed using the standard DSDV routing protocol [Perkins and Bhagwat 1994].

In regular intervals T_a, each replica is allowed to *adapt* based on the information it has gathered locally over the time window T_h. To do this, it can execute one of the following actions.

—*migrate*(v) causes the replica to migrate from its current node to node v.
—*replicate*(v, w) causes the replica r (running on some node u) to be cloned, and the replicas are migrated to node v and w, respectively.
—*dissolve*() causes the replica to remove itself from the system.

T_a is called the *adaptation interval*, and T_h is referred to as the *history window*. In the following, we assume that $T_a = T_h$, and that time is discrete. In the current implementation, T_a and T_h are global system parameters that are defined at deploy time. Both influence the system's reactivity: If they are high, the adaptivity is low and vice versa. In practice, they would be set to some value between several minutes and a day depending on the application-related reactivity required.

The essential part of the placement algorithm is concerned with when to invoke the respective action and with the selection of adequate nodes as parameters.

ASG services can be stateful. Each replica of a stateful service accepts write operations from local clients. Thus, the local *data stores* of the replicas become inconsistent over time and have to be synchronized by means of a consistency protocol. Statefulness and consistency are important factors in any replication mechanism because it effectively limits the number of replicas that may be produced and the distance between them: With stateless services, the ideal solution to our placement problem would be to put a replica on each node. With stateful services, however, this would introduce a vast message overhead since the replicas have to communicate to achieve consistency. The adaptation algorithm implicitly takes into account the messages sent through the consistency protocol and, thus, prevents replicas from migrating too far apart. Moreover, it is completely independent of the concrete consistency protocol as it only analyzes the message flows in the system. We have designed and implemented a consistency protocol for the ASG [Herrmann 2007a] that can efficiently update whole groups of replicas in a single reconciliation process. This protocol is a variant of the well-known Bayou anti-entropy protocol [Terry et al. 1995].

3.3 Cost Model

The cost model that we use to evaluate our solution contains three distinct cost components.

—The *request cost* C_q is incurred by client requests. C_q is dependent on the rate at which clients produce requests and on the distance between clients and the replicas they use.

—The *migration cost* C_m that is caused by the transmission of the replicas' state as they migrate.

—The *replication cost* C_r is incurred by the messages being exchanged between the replicas of a service in order to achieve consistency among their states.

The time-dependent overall cost function is defined as

$$C(t) = C_q(t) + C_r(t) + C_m(t). \tag{1}$$

This cost function only contains the communication costs. We are not trying to optimize aspects like CPU or memory usage. The cost in our model is measured in number of average client messages. We do not use raw bytes as our unit since we want to be independent of any concrete application. Our simplifying assumption is that there are basic data entities that are subject to requests and responses and that are stored in the replicas' data stores. The concrete (average) size of such an entity is depending on the application. Each request and response has size 1 (one data entity) and the data store has size $\sigma \gg 1$ (σ data entities). All components of the overall cost (Eq. (1)) have the same weight since this *data unit model* ensures that all components contribute equally to the communication cost.

Let $v_t(x)$ be the node (service cube) in G at which the program x is running at time t. x is either a client or a replica. Let $p_t(c)$ be the replica that serves the requests issued by client c at time t. The system allocates replicas to clients in a way that is transparent to the client. Let $q_t(c)$ be the current number of requests issued by client c at time t, and let $d(u, v)$ be the number of network hops on the shortest route between nodes u and v. K is the set of clients in the system. The request cost component at time t is defined as follows.

$$C_q(t) = 2 \cdot \sum_{c \in K} q_t(c) \cdot d(v_t(c), v_t(p_t(c))) \tag{2}$$

We use a factor of 2 since we assume that a response is generated for each request. Thus, we have two messages being transmitted for each request.

Let w be the write rate exhibited by clients. w is the fraction of all client requests that contain write operations, and let R_t be the set of all replicas of a specific service at time t. The *replication distance* $d_t^R(r)$ for replica r is defined as follows.

$$d_t^R(r) = \sum_{r' \in R_t} d(v_t(r), v_t(r')). \tag{3}$$

We assume that a replica has to transmit each write operation to all of its fellow replicas in order to achieve consistency among the replicas. Note that this is a very conservative assumption as, in reality, the consistency protocol that we implemented in the ASG is somewhat more efficient. The replication distance is simply the sum of all shortest paths between r and its fellow replicas in R_t. Based on these definitions, we define the replication cost as

$$C_r(t) = w \cdot \sum_{c \in K} q_t(c) \cdot d_t^R(v_t(p_t(c))). \tag{4}$$

Let $\{t_1, t_2, t_3, \ldots\}$ be the points in time at which adaptations happen with $t_{i+1} - t_i = T_a$. Let $M_t \subset N \times N$ be the set of all migrations at time t where a migration is expressed as a pair of source and destination nodes (u, v). We define the *migration cost* as follows.

$$\forall t \in [t_{i-1}, t_i] : C_m(t) = \frac{1}{T_a} \cdot \sum_{(u,v) \in M_{t_{i-1}}} \sigma \cdot d(u, v) \tag{5}$$

In other words, the migration cost that is in reality incurred at times t_1, t_2, t_3, etc., is distributed evenly over the time intervals between the adaptations. This is done because the migrations cost is fundamentally different from the request and replication cost. While the latter are associated with a specific configuration (positions of replicas and clients) of the system, the former is associated with the transition from one configuration to another. Thus, it must be handled like an *investment* that has to pay off through decreased request and replication costs over the following interval. σ is the average size of a replica.

The three cost components represent opposing optimization criteria: $C_q(t)$ is minimal if $|R_t| = |K|$ and one replica resides on each node that hosts a client, such that each client request can be serviced at the client's location. For $w > 0$, $C_r(t)$ is minimal if $|R_t| = 1$ since no reconciliation is necessary in this case. For $1 < |R_t| < |K|$, $C_r(t)$ is minimized by having replicas stay close together as this minimizes the replication distance, and $C_q(t)$ is minimized by distributing replicas to request hotspots. $C_m(t)$ is minimal if adaptations are avoided completely and replicas stay at the same nodes all the time. However, this would imply that neither $C_q(t)$ nor $C_r(t)$ can be actively minimized. Therefore, there are dependencies between all three cost components, and minimizing the overall cost $C(t)$ is nontrivial.

3.4 The Hardness of the Problem

The placement problem that we have to solve in the ASG is related to the well-known *Uncapacitated Facility Location Problem* (UFLP) [Cornuéjols et al. 1990] which is defined as follows: Given a set of locations N, a set of facilities has to be built on these locations such that a given set of clients (also residing on locations in N and using the closest available facility) produces the lowest possible cost when using these facilities. The cost produced by a client j connected to a facility i (denoted as $j \to i$) is defined as $d_j c_{ij}$, where d_j is the client's demand, and c_{ij} denotes the distance between i and j. In addition to these costs, building a new location i incurs a cost of f_i. c_{ij} is given by the topology of the underlying graph $G = (N, E)$. The objective is to find a number of facilities and their locations such that the overall cost $\sum_{j,i|j\to i} d_j c_{ij} + \sum_i f_i$ is minimal.

It has been shown that the UFLP is NP-hard [Cornuéjols et al. 1990]. To show that the replica placement problem (called RPP hereafter) is also NP-hard, we can map UFLP to RPP in polynomial time such that an optimal solution to RPP gives an optimal solution to UFLP. We assume that the clients in the RPP do not move (like in UFLP) and produce requests at a constant rate. Facilities in UFLP are mapped to replicas in RPP. We map the client-induced cost of UFLP ($d_j c_{ij}$) to the request cost C_q of RPP. The facility cost f_i of a facility i is mapped to the migration cost produced by the series of migrations necessary to eventually place a replica on the desired node. The initial replica r is placed on a node v at the beginning. Each link (u, w) in the network is assigned a migration cost of $f_w - f_u$. Thus, the sum of all migration costs produced by r on its way from v to w is equal to the value assigned to f_w. In order to do this, migration costs are also allowed to take negative values. However, this does not restrict generality in any way. The replication cost is set to 0, assuming the case of stateless services. With this mapping, an optimal final configuration produced by the migrations and replications in the RPP instance also represents an optimal solution to the corresponding UFLP instance. Therefore, $UFLP \preceq_p RPP$ and RPP is NP-hard.

The fact that we have to solve this problem continuously in a dynamic environment (mobile clients and changing request flows) with local information only makes the task of any online algorithm for this problem even harder. We conclude that an optimal online solution is not efficiently computable in our case. Therefore, our algorithms use a heuristic approach.

4. THE PLACEMENT ALGORITHM

We propose a fully decentralized, usage-driven placement algorithm that is run by each replica. It inspects the flows of messages received by the replica and takes a local decision about an adequate adaptation. No central controller and no additional communication among the replicas is required in order to achieve a coordinated global placement pattern that minimizes the cost $C(t)$. Moreover, this placement pattern stabilizes if no changes occur in the environment (e.g., in client request patterns), and, at the same time, it retains its adaptivity to be able to react to relevant changes. The means by which this is achieved are

```
procedure ADAPT($r \in R$)
begin
  if IDLE($r$) then $r$.DISSOLVE()
  else
    $(v, w)$ = GETREPLICATIONTARGETS($r$)
    if $v \neq \overline{v} \wedge w \neq \overline{v}$ then $r$.REPLICATE($v, w$)
    else
      $v$ = GETMIGRATIONTARGET($r$)
      if $v \neq \overline{v}$ then $r$.MIGRATE($v$)
      endif
    endif
  endif
end
```

Fig. 1. The core adaptation algorithm.

elegant and easy to implement. By employing this algorithm, the collection of replicas is able to self-organize its placement without any external intervention.

4.1 Basic Feedback Rules

The whole placement mechanism is based on two basic feedback rules, described next.

(1) *Clients always send their requests to the closest replica.* This rule is enforced by the ASG lookup service [Herrmann et al. 2005]. It is not part of the placement algorithm's function.
(2) *Replicas try to get closer to their peers.* A replica's peers are those clients and fellow replicas that communicate with it. This rule is enforced by the placement algorithm we propose.

These two rules interact in a positive feedback loop: As a replica moves closer to its clients (rule 2), it attracts even more clients from the respective area (rule 1). This, in turn, increases the area's attractiveness for the replica (rule 2), and so on. As the replica moves into the group of its clients, a negative feedback sets in and lets the replica converge to a stable position. We will investigate these positive and negative feedback processes in more detail later on.

4.2 Core Adaptation Algorithm

Figure 1 depicts the core adaptation algorithm that is run by each replica. It uses three rules (implemented as functions) to decide about the action that shall be invoked on a given replica r. In the following, $\overline{v}$ represents the *invalid node*. Whenever a function shall return a node fulfilling a specific condition but no such node can be found, it returns the value $\overline{v}$.

ADAPT defines a general precedence for the rules applied in the adaptation process. If a replica is idle (receives no or very few requests) it is removed. Otherwise, we first try to replicate it. GETREPLICATIONTARGETS() selects target nodes for the replication according to some strategy that is defined shortly. If the algorithm selects valid target nodes, the REPLICATE action is executed on the replica. We try to replicate first since this produces the necessary number of

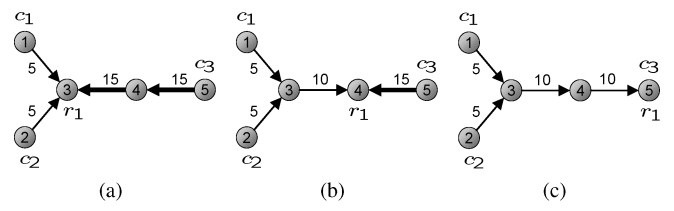

Fig. 2. Exploiting dominating flows for incremental cost optimization.

replicas quickly. If we would opt to migrate first, then the single initial replica would tend to move towards the center of the overall request traffic, and then it would produce replicas that move back towards the *outskirts* of the network to serve the different request hotspots. This causes unnecessary migrations and, thus, costs. If we replicate early, the individual replicas may move more directly to locations that minimize network traffic. If the algorithm cannot replicate, it tries to migrate by calling the GETMIGRATIONTARGET(). If the selected node is valid, the algorithm issues a MIGRATE operation on the replica. Note that if neither a pair of replication targets nor a migration target is selected, the core algorithm exits without issuing any adaptation. The implementation of IDLE(), GETMIGRATIONTARGET(), and GETREPLICATIONTARGETS() specifies the strategy applied by the service distribution algorithm. We introduce this strategy in the following.

4.3 Idle Rule

A replica is considered to be idle if it has not received more than α (*idle threshold*) requests per time unit on average during the past m adaptation intervals (*idle grace time*). Thus, α specifies which level of load (requests per time unit) is necessary to justify the existence of a replica. m defines the length of time we are willing to tolerate load levels that are too low. It is beneficial to wait before dissolving a replica since its low request rate may only be temporary. A replica that is requested to dissolve initiates a reconciliation process with its fellow replicas before it leaves to make sure that no data is lost. Any incoming client requests are adequately handled by the software layer hosting the replica to ensure that the effect on clients is minimized. This is also true for migrations and replications. For brevity, we will not go into the details of the respective mechanisms.

4.4 Migration Rule

The migration rule exploits a basic property of message flows in an ASG network. Figure 2 depicts a simple network with three clients and one replica. Clients c_1 and c_2 send 5 requests per time unit each while c_3 sends 15 requests per time unit. Replica r_1 can minimize the number of transmissions per time unit in the network (denoted as C_{qr}) simply by moving towards c_3 because the amount of transmissions flowing towards r_1 from c_3 (15) is greater than the sum of all other message flows $(5 + 5)$. If a flow $\mathcal{F}$ (series of incoming requests) is larger than the sum of the remaining flows, we also say that $\mathcal{F}$ is *dominating*.

The basic strategy followed by the migration rule is to move towards the source of a dominating message flow if such a flow exists.

More formally, let $N_v = \{w|(v, w) \in E\} \cup \{v\}$ be the set of v's neighbors in $G = (N, E)$, including v itself. Let $m_v(r, w, t)$ be the number of requests that replica r, running on node v, receives via neighbor $w \in N_v$ at time t. Then

$$M_v(r, w, t_0, \Delta t) = \sum_{t=t_0-\Delta t}^{t_0-1} m_v(r, w, t) \tag{6}$$

is the sum of these requests in the time interval $[t_0 - \Delta t, t_0 - 1]$. t_0 denotes the current time index at which the migration rule is evaluated. The fact that at some time index t_0, the request flow received via a neighbor u in the history window T_h is dominating, is expressed by the *simple migration condition*.

$$M_v(r, u, t_0, T_h) > \sum_{w_i \in N_v \setminus \{u\}} M_v(r, w_i, t_0, T_h). \tag{7}$$

For brevity, we will also use the notation $M_v(w)$ instead of $M_v(r, w, t_0, T_h)$. We could let the algorithm migrate replica r to node u if condition (7) is true. However, this would eventually result in a behavior that is inherently unstable: Suppose that r receives two message flows of size m from u and w with $N_v = \{u, w, v\}$, and both flows are subject to small random fluctuations $\varepsilon_u, \varepsilon_w \ll m$. In this case, the random variation in the difference between ε_u and ε_w lets r migrate randomly between u, v, and w. In order to avoid such an undesirable behavior, we apply a stability analysis to $M_v(r, u, t_0, T_h)$ and require that condition (7) must be stable. To analyze the stability, we divide the history window $[t_0 - T_h, t_0 - 1]$ into smaller subintervals of length Δt with $\Delta t \ll T_h$ and $(T_h \bmod \Delta t) = 0$ and apply condition (7) to each of these intervals. The condition is stable iff it holds for more than an η-fraction of all subintervals. η is called the *stability threshold*. To formalize this, we first define a function that evaluates to 1 if condition (7) holds in a given interval and to 0 otherwise.

$$b_v(r, u, t_0, \Delta t, I) = \begin{cases} 1 : M_v(r, u, t_0, \Delta t) > \sum_{w_i \in N_v \setminus (\{u\} \cup I)} M_v(r, w_i, t_0, \Delta t) \\ 0 : \text{otherwise.} \end{cases} \tag{8}$$

I is a set of neighbor nodes that shall be ignored in the analysis. We will use this feature later on in the replication rule (refer to Figure 4). Based on this function, we can formulate the *stability condition*.

$$\frac{\Delta t}{T_h} \cdot \sum_{i=0}^{\frac{T_h}{\Delta t}-1} b_v(r, u, t_0 - i \cdot \Delta t, \Delta t, I) > \eta. \tag{9}$$

The *advanced migration condition* is defined as a conjunction of (7) and (9). Let $P_m(r, u)$ be the migration predicate that is true iff condition (7) is fulfilled for r and u. Likewise, let $P_s(r, u, I)$ be the stability predicate that is true iff condition (9) is fulfilled for r, u, and I. The advanced migration condition is

$$P_m(r, u) \wedge P_s(r, u, \emptyset). \tag{10}$$

With a sufficiently high value for η, the stability condition (9) ensures that random fluctuations do not cause a migration. In our simulations, we typically

```
procedure getMigrationTarget($r \in R$)
begin
    select $u$ from $N_v$ with $M_v(u) = max_{w \in N_v} M_v(w)$
    if $P_m(r,u) \wedge P_s(r,u,\emptyset)$ then return $u$
    else return $\overline{v}$
    endif
end
```

Fig. 3. Implementation of the migration rule.

use 20 subintervals and $\eta = 0.7$ (refer to Figure 15(a)) for the stability analysis. We assume that when random fluctuations happen in an unstable state they result in the stability condition being fulfilled in about half of all subintervals Δt. Thus, appropriate values for the stability threshold η are $\frac{1}{2} < \eta < 1$. r is migrated to node $u \in N_v$ at time t_0 iff condition (10) holds. In this case, we also say that u is stably dominating in $N_v \setminus I$.

Unfortunately, it is nontrivial to derive the best stability threshold automatically. This would require knowledge about the dynamics of flows in the system and a new system parameter describing the desired adaptivity. The former is very hard to capture with local information only, while the latter is very hard to define in meaningful terms. Therefore, we can only offer an experimental analysis of the adequate stability threshold (see Section 6.3.3).

The complete migration rule is implemented in the function getMigrationTarget (used in the core algorithm depicted in Figure 1) as shown in Figure 3.

4.4.1 *Unpredictable Changes Induced by Migrations.* In the simple example in Figure 2, the requests generated by c_1 and c_2 flow through the path taken by r_1 in its migrations (nodes 3 and 4). However, this is not necessarily true in the general case. Since a replica has no knowledge about the routing of requests and the global mapping of clients to replicas, it cannot predict how the requests will flow through the network after its migration. It seems as if such unpredictable changes in flows could void our assumptions that (1) the request traffic is generally reduced through a migration and (2) that the whole process converges to a stable low-cost configuration. However, in the following, we will show that this is not the case if we assume that the routing algorithm succeeds in finding the shortest path (in terms of hop distance) between two nodes.

Two general cases may occur after a migration.

(1) A client c_1 is attracted to another replica r_2 that has become the closest replica due to the migration of r_1. Let us assume that the distance between c_1 and r_1 before the migration of r_1 is $d(c_1, r_1) = l$. After the migration, $d(c_1, r_1) \leq l + 1$ holds since a migration covers only one hop. If c_1 switches to r_2, $d(c_1, r_2) = l$ must hold due to the first feedback rule (see Section 4.1). Therefore, the requests of c_1 travel the same distance as before while the dominating flow travels one hop less due to the migration. This results in a reduction of the overall request traffic.

(2) A client c_1 continues using r_1. Again, $d(c_1, r_1) \leq l + 1$ holds after the migration. $l + 1$ is the length of the route before the migration plus the additional

hop. If the requests from c_1 are routed through a different path due to the migration, this path cannot be longer than $l+1$ hops because the routing algorithm always chooses the shortest route. Therefore, the requests from c_1 travel at most one hop more than before while the dominating flow travels one hop less. Again, the result is a reduction of the overall request traffic.

As a consequence, in a system where clients do not exhibit any dynamics, the request traffic will be reduced with every migration until a configuration is reached where there is no dominating request flow. In this situation, the stability criterion (see preceding) takes over and avoids futile migrations. If clients are mobile and change the volume of requests they are sending, these changes may lead to migrations that are eventually reverted, but that is a normal phenomenon in adaptive systems.

An additional situation that may occur is the simultaneous migration of two (or more) replicas of the same service type to the same node. This has no negative effects, neither on the achievement of the optimization goal nor on the stability of the system. Such a case may appear if both replicas receive requests from disjoint sets of clients (no client sends requests to both replicas)[2] via the target node of the migration. These flows will still be associated with the respective replicas after the migration. Thus, there is no interference between the two replicas other than the fact that requests share the same network links. Note that the probability of two replicas clumping together like this for an extended period of time is very low. Due to the two feedback rules, differences in request flows will soon lead to different forces being applied to the replicas. This asymmetry will be amplified quickly, and both will be pulled into different directions.

4.5 Replication Rule

Replications are issued based on the average distance between the replica and all entities in the system from which it receives messages (clients and fellow replicas). For simplicity, these entities will be denoted as "clients"' in the following. We assume that the distance (number of network hops) between the replica and a client can be extracted from the client's request messages. This defines a basic requirement for the system (e.g., middleware platform) on which our algorithm is implemented.

The parameter ρ defines the so-called *replication radius*. Informally, the replication rule can be stated as follows.

> For $|N_v| \geq 2$, we replicate if the messages received from neighbor u, contributing the highest request flow, have an average path length greater than ρ. The destinations of the replication are u and the neighbor contributing the second highest stable request flow.

In order to achieve stability also in the replication process, we need to avoid that insignificant request flows with a high average path length can trigger a

[2]Note that client sets are always disjunct since a client sticks to a replica unless another one becomes the closest.

```
procedure getReplicationTargets($r \in R$)
begin
  if $|N_v| < 2$ then return $(\overline{v}, \overline{v})$
  else
    calculate $U_{\widehat{d}}$ and $U_{\widehat{m}}$
    forall $u \in U_{\widehat{d}}$ do
      if $\overline{d(u)} > \rho \wedge u \in U_{\widehat{m}}$ then
        select $u' \in N_v \setminus \{u\} \mid \forall w \in N_v \setminus \{u\} : M_v(u') \geq M_v(w)$
        if $\neg P_s(r, u', \{u\})$ then $u' = v$ //u' not stable!
        return $(u, u')$
      endif
    endforall
    return $(\overline{v}, \overline{v})$ //No node u found!
  endif
end
```

Fig. 4. Implementation of the replication rule.

replication. Otherwise, a single client at the edges of the network may cause a replication into a region that does not provide sufficient volumes of requests to justify the existence of a replica. We do this simply by requiring that a replication can only be triggered by the neighbor node with the highest request flow.

Let $\overline{d(u)}$ be the average distance (number of hops) traveled by requests that were received via neighbor node u. Let $U_{\widehat{d}}$ be the set of neighbor nodes with the highest average distance in N_v, and let $U_{\widehat{m}}$ be the set of neighbor nodes with the highest message flow. The *replication condition* is defined as follows.

$$|N_v| \geq 2 \ \wedge \ \exists u \in U_{\widehat{m}} \cap U_{\widehat{d}} \mid \overline{d(u)} > \rho. \tag{11}$$

The algorithm chooses a node u such that condition (11) is fulfilled. If such a node exists, the algorithm chooses a second node u' that is stably dominating in $N_v \setminus \{u\}$ and uses u and u' as replication targets. If such a node u' does not exist, the original replica stays at its current location, v. The replication rule is implemented by the function getReplicationTargets (used in the core algorithm) as shown in Figure 4.

In principle, it would be possible to produce more than one additional replica in a replication and send them to those neighbors that contribute the most significant request flows. We chose to keep this number to the minimum of one since a higher number always implies a higher risk of overshooting the optimization goal. The algorithm introduced before is nothing else than an online optimization algorithm that proceeds stepwise as many well-known optimization techniques do (e.g., gradient descent). Such algorithms have parameters defining the size of the steps taken towards the minimum. The number of replicas created in a replication is such a parameter in our case. Producing more than the minimum number of replicas may help, but it may just as well be too big a step and lead to undesirable effects (e.g., oscillation). After all, replication introduces potential exponential growth. Keeping the base of this growth as small as possible helps in controlling the overall process.

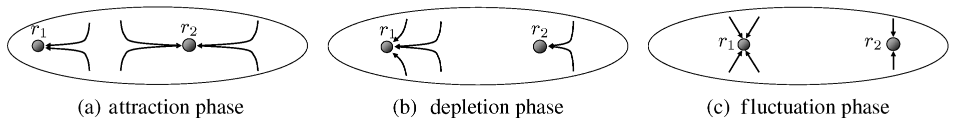

Fig. 5. The three feedback phases.

5. EMERGENT FEATURES

Before we investigate the performance and the quantitative properties of the algorithm presented earlier, we will first analyze those features that occur as a larger number of replicas execute the algorithm in parallel. The overall process, by which the repetitive application of the three rules finds a stable, cost-efficient location for a replica, is *emergent*. That is, it bears mechanisms that are not coded directly into the rules. Instead, they are a product of the interactions between replicas. The key to employing the proposed mechanism effectively is an understanding of this emergent behavior since it represents the link between the simple local rules presented before and the complex global performance of the overall system of indirectly coupled replicas [Herrmann 2007b].

In this qualitative evaluation, we will investigate the emergent properties of a collection of replicas belonging to a single service type. These replicas experience coupling through client requests while replicas belonging to different service types do not experience such a coupling and act largely independently from each other. Note that we only explore message flows as coupling mechanism here. Other factors like resource consumption (e.g., memory and CPU usage) may couple the behaviors of replicas belonging to different service types in a real environment. However, we chose not to investigate this effect for now in order to preserve the simplicity necessary for doing an in-depth analysis.

5.1 Collective Migration

The advanced migration condition (10) implements an essential part of the feedback mechanism discussed in Section 4.1. The behavior of a single replica is straightforward: It migrates until it finds a location that produces an equilibrium (absence of a dominating node) in the message flows received from all neighbors. However, the collective behavior of many replicas is less obvious. The key to the analysis of this behavior are the interactions between the replicas. From the three rules, it is obvious that there is no direct interaction since there is no exchange of messages and the replicas are completely ignorant of each other. Instead, the replicas are coupled indirectly through the message flows they receive: Firstly, when a flow towards one replica increases, the flow towards some other replica must decrease since requests are only sent to one replica. Secondly, whenever a replica r migrates, it may change the assignment of clients to replicas in the system since it moves away from some clients and towards other clients such that the former may choose another replica while the latter may switch to r.

The process can be divided into three phases (refer to Figure 5). For the sake of simplicity, we assume that there is only very little dynamics in the system.

5.1.1 *Attraction Phase (Positive Feedback).* Let us assume that, at the start of the process, replica r_1 is placed in a suboptimal location with respect to the client positions and the request patterns. In such a position, the flows received via the neighbors of v tend to differ in strength very widely since all requesting clients lie in the same general direction, and the network routing mechanism gathers the requests on one route such that they are received by r_1 via the same neighbor. In this phase, r_1 is attracted very quickly by the group of requesting clients (see Figure 5(a)). Due to first feedback rule, this tends to attract more clients which strengthens the attraction even more. This is depicted in Figure 5(b) where clients that were previously sending requests to replica r_2 are now switching to replica r_1.

5.1.2 *Depletion Phase (Negative Feedback).* After some adaptations, r_1 crosses a point at which the directions of requests start getting more diverse (see Figure 5(b)). As r_1 gets closer to the group of clients, their requests come in via more than one neighbor node since the distance is too small for the gathering effect of the routing algorithm to occur. As this directional diversity sets in, the solution to condition (7) becomes less obvious, which causes the strength of the attraction present in phase 1 to decrease. This is the point at which the negative feedback sets in because the force that caused the quick migration towards the group of requesting clients is progressively depleted by the migration itself. The speed of adaptation is reduced, and r_1 slowly approaches a good position around the center of the group of requesting clients. This mechanism has a strong stabilizing effect.

5.1.3 *Fluctuation Phase.* As r_1 settles down, the differences in the incoming flows are evened out (refer to Figure 5(c)). When this has happened, the system is more or less in a stable state, that is, there is no distinct force that drives r_1 into further adaptations. However, the degree of stability may vary in this phase. At one extreme end, r_1 may constantly be at the edge of yet another migration, and at the other extreme, it may be far from migrating. For the ability of the system to react to dynamic changes by readapting, this phase is the most important one. The system must not execute random, unproductive migrations. However, it must still be able to adapt quickly if significant changes occur. The stability condition (9) solves this problem by checking the statistical significance of a flow difference that fulfills the migration condition. Thus, the algorithm is able to recognize stable changes and react to them while unstable changes are ignored.

5.2 Collective Replication

The replication rule forces replicas to distribute in such a way that the main stream of client requests for each replica travels at most ρ hops on average. If an average request comes from inside the replication radius, the algorithm will not replicate and try to migrate instead to optimize its distance to its current clients. Effectively, the replication radius ρ controls the overall number of replicas $|R|$. As soon as each replica covers a network portion of radius ρ, the stimulus to replicate vanishes. If ρ is large, then a small number of

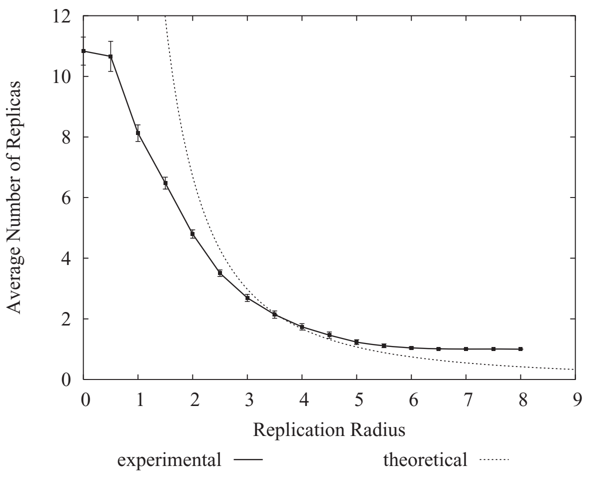

Fig. 6. Overall number of replicas $|R|$ against ρ.

replicas suffices to cover the whole ASG network. And, conversely, if ρ is small, many replicas are required. This behavior is also a consequence of the indirect coupling via the message flows. This time the information that is indirectly transported via the flows is related to network coverage. A message flow from outside a radius of ρ indicates that further away in the respective direction there are not enough replicas to cover the network. On the other hand, the absence of such a message flow indicates that every client has a replica within ρ hops.

Figure 6 shows how the average number of replicas over a complete simulation run decreases as the replication radius increases (error bars indicate 95% confidence intervals). This data has been measured for networks of 50 nodes with an average diameter D of 10.35 hops. The function that was acquired experimentally is roughly proportional to ρ^{-2}. It converges to 1 since there is always at least one replica maintained in the system. For $\rho \to 0$, $|R|$ enters a state of saturation.

In order to explain this behavior, we adopt an idealized view of an ASG network and assume that it has a circular shape.[3] In this scenario, each replica covers a circle with an area of $\frac{\pi}{4} \cdot (2\rho)^2$ hops. The complete network may be regarded as a circle with an area of $\frac{\pi}{4} \cdot D^2$ hops. The following approximation for the number of replicas ($|R|$) needed to cover the ideal network holds in this scenario.

$$\frac{\pi}{4} D^2 \approx |R| \cdot \frac{\pi}{4} (2\rho)^2 \tag{12}$$

Simple transformations for $D = 10.35$ yield

$$|R| \approx \frac{1}{4} \left(\frac{D}{\rho} \right)^2 \approx 26.78 \cdot \rho^{-2}. \tag{13}$$

[3]For an average network that is generated according to our network creation model, this is indeed true. Note that this is not a requirement but only helps in explaining the result shown in Figure 6.

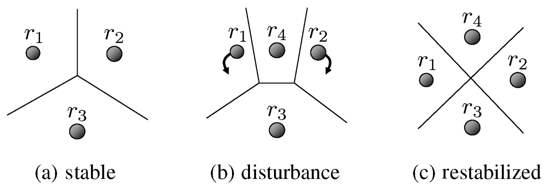

Fig. 7. Adaptation of the cell structure.

This theoretical dependency between $|R|$ and ρ is also shown in Figure 6 (dashed line).

An important issue with respect to the replication rule is its relationship to the optimization of the replication cost. In general, any replication raises the replication cost since an additional replica needs to exchange messages with its fellow replicas in order to preserve consistency. As we have described in the discussion at the end of Section 3.3, the overall goal is to achieve a balance between the three cost components. Decreasing the request cost implies an increase of the replication cost. However, since the replication radius limits the total number of replicas, as shown before, it also effectively limits the replication cost component. In addition, our replication rule implicitly enforces that replicas are evenly distributed which represents a favorable configuration as it minimizes the replication distance on which the replication cost directly depends. Therefore, collective replication in the ASG increases the replication cost only minimally and in a controlled way.

5.3 Adaptive Cell Structuring

The combination and the dynamics of collective migrations and replications creates another effect that is vital for the ability of the overall system to adapt to changes. Due to the fact that clients always use the closest replica, the collective adaptation process of all replicas creates a global pattern consisting of the cells of a Voronoi decomposition. In each cell, a replica r serves all requests being issued by clients inside the cell (clients for which r is the closest replica available). The replica placement algorithm replicates and migrates the set of replicas such that a stable configuration of Voronoi cells is created.

Figure 7 depicts how the cell structure changes if the replica configuration is changed. For this simplified example, we assume that the plane is densely populated with nodes and that client requests are issued uniformly from all nodes. Figure 7(a) shows a stable Voronoi cell pattern with three replicas: Each replica is placed at about the center of its cell and, thus, the forces applied by client requests are about the same in all directions. Let us assume that, due to some disturbance in the request patterns, a new replica r_4 is created between r_1 and r_2. The resulting Voronoi diagram is shown in Figure 7(b). This disturbance changes the cells of r_1 and r_2 such that they are close to the borders of their cells. Since the requests coming from the upper part of the plane are now redirected to r_4, they start receiving an unbalanced flow of requests. According to the migration rule, r_1 and r_2 are forced to move to a position where the flows are in balance again. The resulting four-cell setup is depicted in Figure 7(c). In a

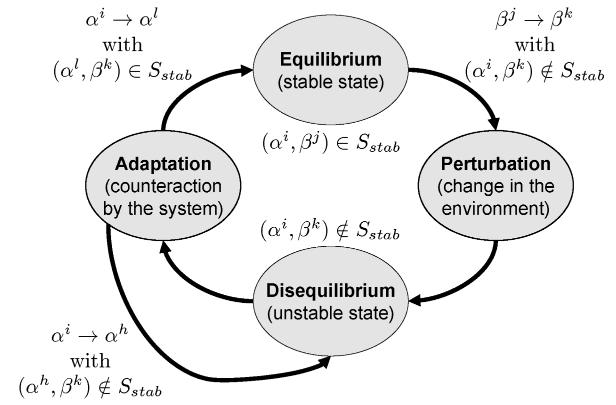
$\alpha^i \to \alpha^l$ with $(\alpha^l, \beta^k) \in S_{stab}$
Equilibrium (stable state)
$\beta^j \to \beta^k$ with $(\alpha^i, \beta^k) \notin S_{stab}$
$(\alpha^i, \beta^j) \in S_{stab}$
Adaptation (counteraction by the system)
Perturbation (change in the environment)
$(\alpha^i, \beta^k) \notin S_{stab}$
Disequilibrium (unstable state)
$\alpha^i \to \alpha^h$ with $(\alpha^h, \beta^k) \notin S_{stab}$

Fig. 8. Adaptation cycle.

setup with more cells, this effect propagates like a wave since each replica that moves creates a similar disequilibrium for other replicas that are positioned in the direction of its movement. Thus, a reconfiguration on a global scale is caused by a local disturbance without any direct communication taking place between the replicas.

5.4 Adaptation Cycle and Attractors

The nature of the three rules (migration rule, replication rule, and idle rule) that govern the adaptations in a running system is such that in the absence of a relevant stimulus, a replica does not adapt. From this simple fact, we can derive the following stability condition for the system.

Definition 1 (*Stability Condition*). The state of the overall replica system is stable if and only if

(1) the incoming message flows of all replicas are balanced (i.e., there is no dominating node),
(2) no replica serves requests with an average path length of more than ρ hops, and
(3) each replica receives a sufficient amount of requests per time unit.

In such a state, none of the adaptation rules of any replica is triggered and, thus, the current replica placement is retained.

Perturbations, for instance, due to changes in client behavior or network topology, can push the system into an *unstable state* in which the stability condition is violated. This automatically leads to a series of adaptations through the respective rules of the distributed placement algorithm until a new stable state is reached (see Figure 8). This resembles a classical *control loop* that counters every deviation from the prescribed *set point*. However, in our case, the set point is no particular placement or any geometrical characteristics thereof, as one may expect. Instead, the set point is *stability*. In complex systems terminology, this set point is also called an *attractor* [Milnor 1985]: Let α be a replica placement and let β be a state of the environment that the replica system has to adapt to (client request pattern, network topology, etc.). Among the set S of all possible system states (α, β), there is a subset $S_{stab} \subseteq S$ that satisfies the stability conditions.

$$S_{stab} = \{(\alpha, \beta) \in S \mid \alpha \text{ fulfills Definition 1 under } \beta\}.$$

Through the adaptation cycle, the replica system is constantly attracted to some placement α^i such that $(\alpha^i, \beta^j) \in S_{stab}$ for the current environment state β^j. If, through some sort of dynamic change, the environment enters a different state β^k such that $(\alpha^i, \beta^k) \notin S_{stab}$, then this instability automatically triggers a *distributed adaptation* and pushes the replica system into a new state α^l such that $(\alpha^l, \beta^k) \in S_{stab}$ holds again until the next environmental change. Note that there may well be a series of intermediate replica placements $\alpha^h, \dots, \alpha^x$ that lead the system through a series of disequilibrium states on its way to stability (refer to lower left part of Figure 8). However, if we assume that, during this series of adaptations, β^k remains unchanged, each of these adaptations reduces the stimulus for further adaptations. For migrations, we have shown this in Section 5.1: The dominance that causes a migration is reduced by the migration until it vanishes. The same is true for replications and dissolves. Therefore, the whole system is successively driven towards a new equilibrium.

But how is stability related to our original goal of reducing the communication cost in the overall system? For each individual replica, a stable state implies that the local costs cannot be immediately reduced by an adaptation. As indicated in Figure 2, a migration towards a dominant node has a potential for reducing the local costs since the sum of request flows received from the remaining nodes ($\mathcal{F}_{rest}$) may have to travel one hop further while the hop count of the dominant flow ($\mathcal{F}_{dom}$) is reduced by one. Due to the definition of dominance, we have $\mathcal{F}_{rest} < \mathcal{F}_{dom}$, and the number of transmissions per time unit required to transport the client requests to the migrated replica is reduced by $\mathcal{F}_{dom} - \mathcal{F}_{rest} > 0$. In a stable state, a replica does not have this immediate possibility of reducing the local costs. Thus, the costs are the lowest that are currently achievable by the replica. There may well be possible migrations that eventually lead to a reduction. However, the replica may only speculate about them. Therefore, the stable state represents a local cost minimum for this particular replica. The communication cost in the overall system is the sum of all local costs. Thus, the overall costs are at a local minimum if all replicas are in a stable state. In other words, the system is constantly forced into some cost-efficient attractor. However, this attractor may represent a local optimum that is some way off the global optimum due to the heuristic nature of the algorithm.

The existence of the adaptation cycle already provides a basic mechanism for avoiding oscillations and for achieving convergence. However, these phenomena reside at different scales: On the large scale (considering the overall network and gross message flows) the mechanisms introduced by the adaptation cycle result in good convergence (as will be shown in Section 6) and low dynamics under static conditions (oscillations). On the small scale, however, it may well be that a replica tends to execute oscillatory migrations between two neighboring service cubes and fails to converge to an absolutely stable state. The stability condition introduced in the migration condition in Eq. (9) in Section 4.4 prevents this to a great extent and achieves good convergence and almost oscillation-free behavior. Evidence for this is given in Figure 11 in Section 6.3.1. However, it has to be noted that oscillations can only be reduced and not completely avoided in such a dynamic system, as a useful balance has to be achieved between stability and agility.

6. PERFORMANCE EVALUATION

In this section, we explore the behavior of our algorithm under different settings of key system parameters. We also compare the performance of the algorithm with three other approaches for distributing service replicas. These approaches include online algorithms that are applicable in practice requiring different degrees of global knowledge as well as offline algorithms that serve as a benchmark and give a good indication for the best (lowest) cost that is achievable. The comparison is based on the cost measure introduced in Section 3.3. Furthermore, we investigate under which conditions our placement approach is feasible. One important variable in this context is the replica size. It is intuitively clear that any mechanism that migrates programs at runtime will only be beneficial if these programs are small enough. We will show what "small enough" means in the context of the ASG by comparing the performance of our algorithms with the performance of other practical approaches for increasing replica sizes. In the following, we will use the label "online" to refer to our algorithm.

While the qualitative investigation in Section 5 focused on replicas belonging to a single service type (due to the independence of service types), this section studies the performance of the full system running different service types as explained in Section 6.2.

6.1 Benchmark Algorithms

Figures 15(c) to 15(f) show comparisons of our online algorithm with three other approaches that are explained in the following.

6.1.1 *Optimal Static Placement.* The ultimate benchmark is the optimal algorithm. However, even with global knowledge in time and space, finding the optimal solution is too difficult due to the computational complexity of our problem. If we consider networks with 50 nodes, a replica set of size 5, and 20 different placements in each simulation, then the solution space has a magnitude of 10^{126}. Therefore, the "optimal" algorithm used to get a lower bound for the cost achievable by our algorithms only calculates an approximation of this lower bound rather than the "real optimum." We constrain our approximation to a single placement that remains static over a complete simulation interval. Thus, our optimal algorithm is not adaptive.

To find a single static placement solution for a given simulation run, we run a simulated annealing algorithm 100 times starting with randomly chosen replica placements and select the best of the 100 solutions. This process is run iteratively, and the number of replicas is increased after each iteration. First, the overall costs decrease when $|R|$ is increased since more replicas can reduce the request cost C_q. At some point, the replication cost C_r starts to dominate and the overall costs go up again. The best placement with the replica number m_{opt} that produces the lowest overall cost is used to compare our algorithm against.

6.1.2 *Average Random Placement.* The average random placement algorithm also generates a single static replica placement. It distributes m_{opt} replicas (found by the optimal algorithm) randomly inside the network 100 times and averages the overall costs for all trials. This algorithm mimics an ad hoc approach that would be taken in practice if no knowledge about the network and the behavior of the clients can be assumed. Since the algorithm knows m_{opt}, it has an advantage over a completely random placement that has to guess m_{opt}. Moreover, a single random solution as it would be applied in practice may be considerably worse than the average case. The random algorithm represents the main opponent of our online algorithms. Our aim is not to beat the static optimal solution, but the average random algorithm.

6.1.3 *Greedy Heuristics.* Qiu et al. [2001] present a widely cited algorithm for placing M Web server replicas in a *Content Distribution Network* (CDN) of N nodes such that the cost of all requests is minimized. This algorithm takes a greedy approach by choosing one replica at a time and placing it at an appropriate node. More precisely, in the first iteration, each of the N potential sites is evaluated individually to determine its suitability for hosting a replica. The costs of this placement are computed under the assumption that all requests are directed to that replica. The second replica is placed by evaluating the costs in conjunction with this first replica assuming that each client chooses its closest replica. This process is iterated until all M replicas are placed. The greedy algorithm by Qiu et al. [2001] has been shown to produce very good placements and, thus, is also used by other researchers as a benchmark for evaluating their work (e.g., [Szymaniak et al. 2005; Karlsson and Karamanolis 2004; Radoslavov et al. 2001]) and has inspired subsequent development of similar approaches (e.g., [Chandra et al. 2004]).

It should be noted that unlike our algorithm, this greedy algorithm assumes global knowledge. It is an online algorithm since it operates on a window of past data (e.g., the last 24 hours) and has no knowledge about the future. However, it requires knowledge about all requests posted throughout the network in the given time window and about the network topology. Furthermore, the algorithm is run on a central node. Finally, it operates with a prescribed number M of replicas. We allow the greedy algorithm to find the best value for M in the same way as the optimal algorithm does. In practice, it would have to guess a value for M in some way. Unlike our online algorithm, it has no means for finding it on its own. Therefore, the cost values presented for the greedy algorithm represent lower bounds.

In our simulation, the greedy algorithm is run at the same intervals as the online algorithm and has the same history window. We assume that a

Table I. Properties of Randomly Generated Networks

	Average Value	95% Confidence Interval
node degree	3.85	0.0125
shortest path length	4.57	0.0174
diameter D	10.35	0.0646

Table II. Default Values Chosen for the System Parameters in the Experiments

Parameter	Description	Default Value
σ	replica size	10000 data units
η	stability threshold	0.7
client ratio	—	0.5
ρ	replication radius	3 hops
ω	write ratio	0.1
#nodes	—	50
T_a	adaptation interval	10000 ticks
T_h	history window	10000 ticks
m	idle grace time	3 adaptation intervals
α	idle threshold	0.1 requests per tick

replica placement at time t_i is produced from the previous allocation at t_{i-1} by migrating existing replicas and (possibly) replicating new ones such that the overall number of hops needed for all migrations is minimal.

6.2 Experimental Setup

The following experimental results were obtained through simulations using the agent-based Repast simulation toolkit [Repast 2009]. For each measurement, we generated 100 random connected networks of 50 service cubes each to obtain average values of statistical significance. The basic characteristics of these networks are given in Table I. Three services were instantiated in each network and a variable number of mobile clients were simulated. Since we are interested in evaluating the adaptivity of the the system, we enforced sudden changes in request patterns. For this purpose, we employed a *hotspot model* for the requests issued by clients: We randomly chose between 3 and 5 hotspots for each service. A hotspot is a service cube k with a high probability p_s that a mobile client connected to k sends a request for service s. In order to set up the request probabilities throughout the network, they were diffused over all the nodes. That is, each node received a value for p_s that is inversely proportional to its distance from the nearest hotspot of type s. In order to test the system's reaction to dynamics, the hotspots were redefined periodically with an interval of 50,000 ticks.

A single initial replica was injected for each service at a randomly chosen service cube at the start of each simulation run. The following measurements were conducted for systems with three different services. Each of these services ran the algorithm and produced its own replicas. For simplicity, we assumed that the services are independent of each other. Unless explicitly stated otherwise, all experiments use the default parameter values given in Table II.

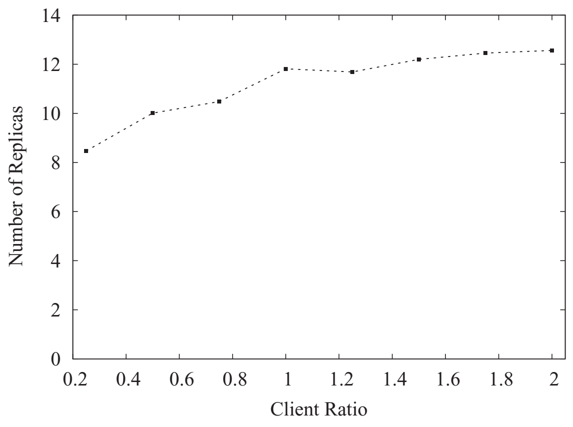

Fig. 9. Average number of replicas with increasing number of clients in the system.

6.3 Results

6.3.1 *Adaptations.* Figure 9 depicts the increase in the number of replicas as the number of clients is raised. The client ratio is the ratio between the overall number of clients in the system and the number of service cubes in the ASG network. At a client ratio of 2, 100 clients are moving through a network of 50 nodes. The figure shows the average overall number of replicas (for all three services) in the system measured over the entire simulation time of 500,000 ticks. The number increases only by 48% as the number of clients is increased by 800%. Moreover, it increases sublinearly. Due to the nature of the replication rule, the limiting factor for the number of replicas is the network diameter and the replication radius (see Section 5.2). The graph shows that the number of replicas is effectively limited by these two parameters.

Figure 10 shows the number of adaptations of each adaptation type (migrations, replications, dissolves) that is executed on average per adaptation interval and replica. Thus, a value of 0.1 migrations means that, on average, 10% of all replicas migrate when the adaptation algorithm is run. Again, the values represent averages over the whole simulation time. The first important fact shown by the figure is the very low number of adaptations. All the figures are in the sub-10% range. This provides evidence that adaptations are not executed unnecessarily. The second implication of the diagram is the fact that the number of adaptations does not increase significantly as the number of clients in the system grows. Migrations are executed more often than replications and dissolves. This indicates that the system tends to employ migrations in order to adapt to changes. The fact that the number of replications lies above the number of dissolves consistently simply means that there are more replicas at the end of a simulation than at the start, which is what we would expect. Note that there is only about a 2%–4% chance for a replica to dissolve in any adaptation interval. This underlines the stability of the system: A set of replicas is created and positioned in the network such that this setup does not require major reconfigurations, and the majority of changes can be handled simply by migrating replicas.

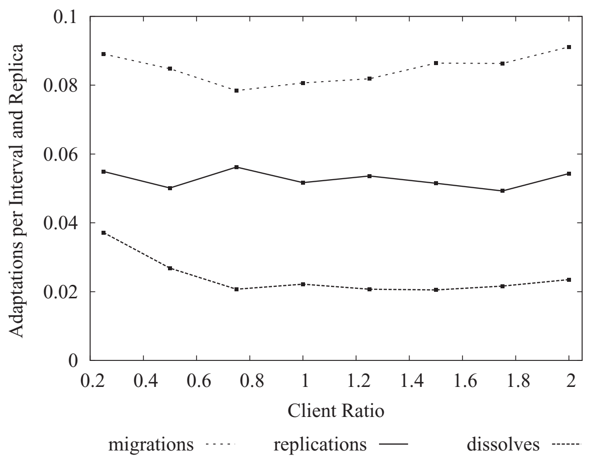

Fig. 10. Average number of adaptations per replica and adaptation interval with increasing number of clients.

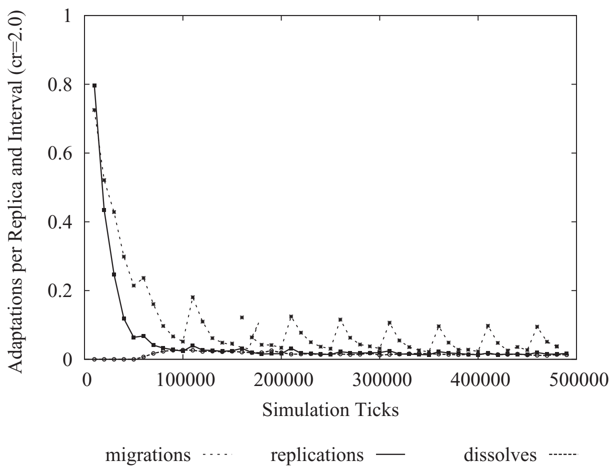

Fig. 11. Number of adaptations per replica and interval with increasing simulation time.

Figure 11 depicts the number of adaptations for a fixed client ratio of 2 against the simulation time. Note that the values are again normalized by the number of replicas that were running at the specific point in time. Thus, a value of 1.0 means that 100% of all active replicas executed the respective adaptation. The figure shows that initially, many replications and migrations take place. In this phase, the system tries to create enough replicas and to distribute them throughout the network. The number of replications and migrations drops significantly and rapidly. The periodic peaks in migrations reflect the points at which the usage patterns are changed. These changes are primarily countered by migrations while the number of replications and dissolves drop to values between 1 and 4%. Figure 12 shows a magnified version of Figure 11, depicting only the replications and the dissolves. After the system reaches a steady state

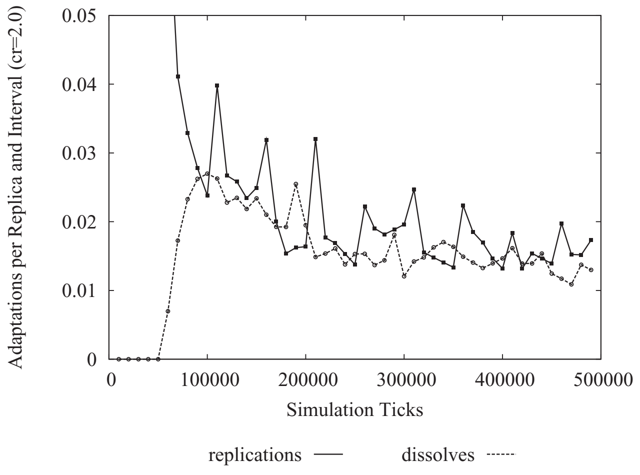

Fig. 12. Number of replications and dissolves per replica and interval (zoomed in).

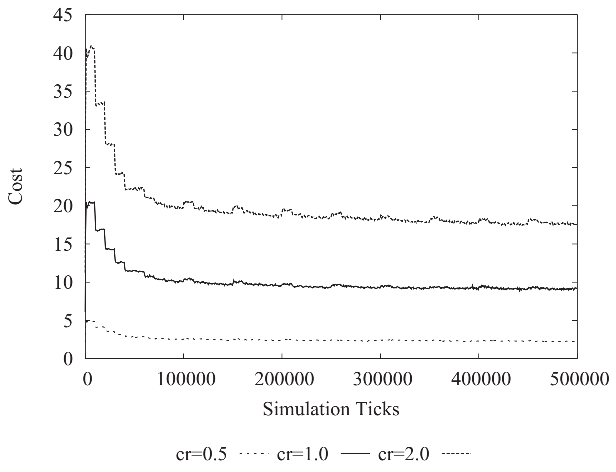

Fig. 13. Cost over simulation time for different client ratios.

(at about 100,000 ticks), both values fluctuate around the same region. This shows that an equilibrium is established between the creation of new replicas and the removal of existing ones. As a consequence, the number of replicas stays at about the same level. Moreover, Figure 12 indicates a continuous downward trend: As the system is running, less and less fluctuations in the number of replicas is experienced, even though there are periodic changes.

Figure 13 depicts the average cost over the course of a simulation run for three different client ratios. It becomes evident that the adaptations shown in Figure 11 lead to a quick initial decrease in communication cost to about half the initial value. After 10,000 ticks, the changes in request patterns are countered and the cost is steadily decreased. This shows that the adaptations are indeed effective.

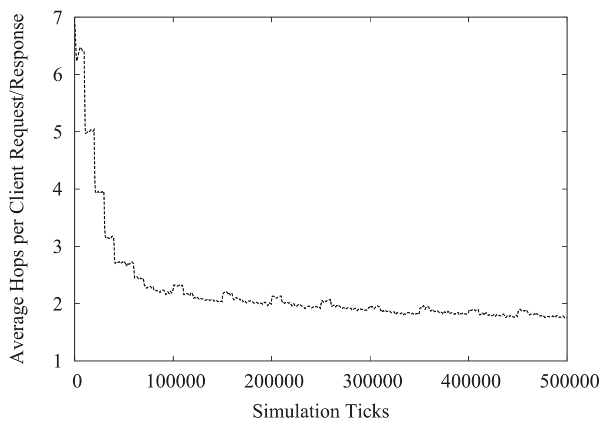

Fig. 14. Average hops per client request/response.

6.3.2 *Client Response Times.* We assume that the response times perceived by clients are foremost dependent on the wireless communication medium. In our model, the wireless bandwidth is limited while the processing capacity on the service cubes is not. Therefore, the algorithm does not optimize the replica placement with respect to the available processing capacity on the nodes. In practice, this is unrealistic. However, we argue that in most scenarios, the capacity of the individual service cubes can be chosen large enough while the bandwidth depends on the networking technology and diverse lower-level protocols (e.g., MAC). Today's technologies still put more or less severe limits to the achievable bandwidth. In any case, the next step in our future work will be to integrate other factors like available processing capacity and free memory into the model.

Under the preceding assumptions, however, the client response time is proportional to the number of hops required to transport requests and responses between clients and replicas. Figure 14 shows the number of hops required on average to transport a message (request or response) between a client and a replica. The graph has the same general shape as the cost graph in Figure 13 because the reduction in cost is achieved by moving replicas closer to clients and, thus, minimizing the hop count. It shows that the average hop count is reduced to a value of around 2, which corresponds well with the chosen replication radius of 3: If clients with a distance of 1, 2, or 3 send requests to each replica, the average distance will be 2. Based on our bandwidth-oriented model, the reduction from almost 7 hops to under 2 hops also implies a reduction in response times of about 70%.

6.3.3 *Stability Threshold.* Figure 15(a) displays the behavior of the online algorithm as the stability threshold η is increased. At $\eta = 0$, the stability check (Eq. (9)) is effectively turned off. At $\eta = 1$, migrations are only triggered if the migration condition holds for all subintervals of T_h. Without any stability check, our algorithm produces more adaptations and tends to oscillate between a small set of acceptable replica placements as request patterns fluctuate randomly. If

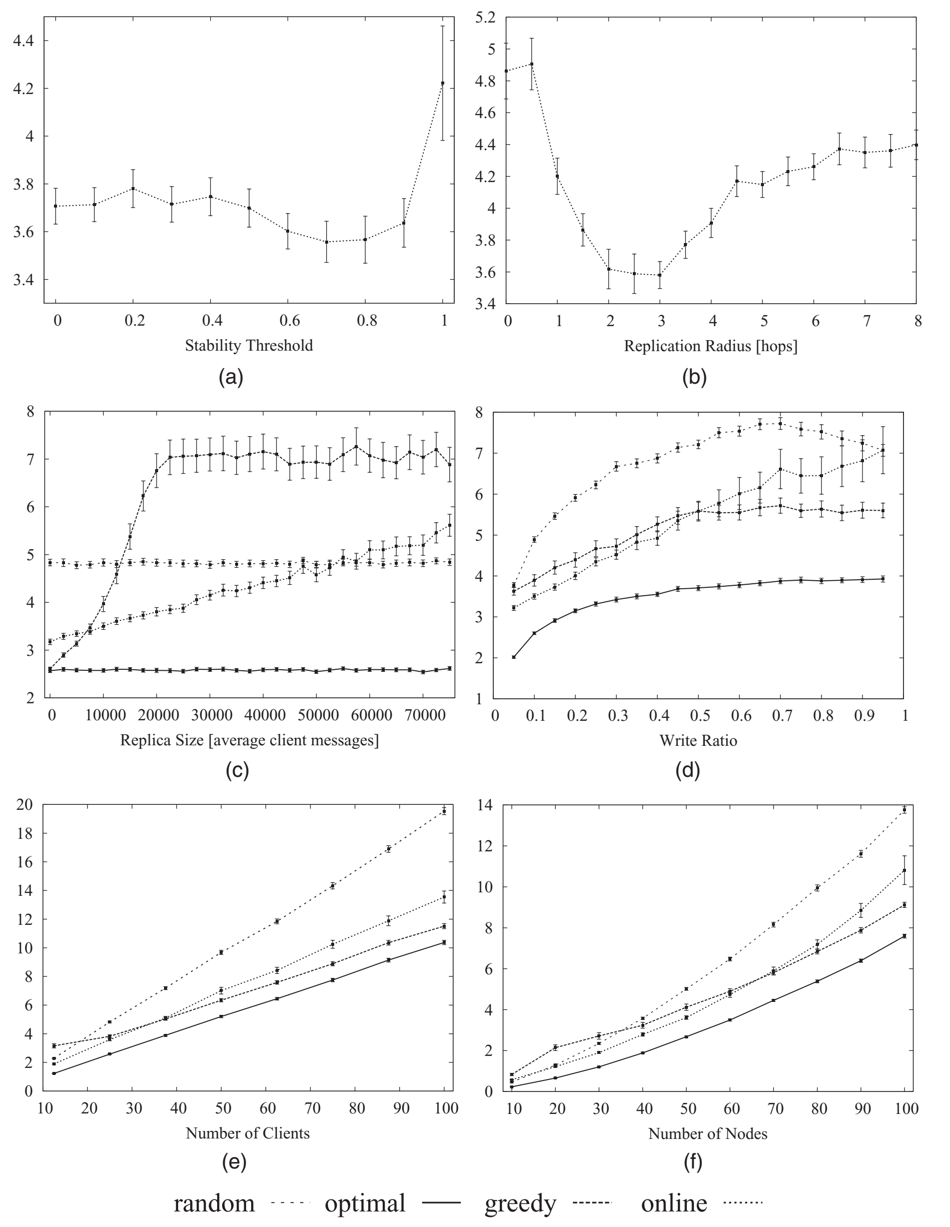

Fig. 15. Experimental results: All diagrams show the placement cost (refer to Section 3.3) for different algorithms and against different parameters. Error bars indicate 95% confidence intervals.

we require complete stability, many useful adaptations are prevented because request flows are rarely absolutely stable. In both extreme cases, the cost produced by the online algorithm is higher than the optimum which occurs at about $\eta = 0.7$. Our algorithm already has an inherent stabilization mechanism since it may only migrate replicas to neighbor nodes. For other variants of the algorithm that are allowed to take larger steps (not covered in this article) the effect of explicit stabilization is larger.

6.3.4 *Replication Radius.* If ρ is close to 0, then a large number of replicas is distributed in the network. As ρ grows, this number converges to the minimum of 1 replica. At both extremes, the cost produced by the online algorithm is suboptimal. Figure 15(b) shows that, for the setup chosen in this experiment, the best value for ρ is between 2.5 and 3.0. To the left of this point, the number of replicas gets too high, the replication cost grows and starts to dominate the overall cost. At the same time, the request cost is not reduced significantly beyond a certain number of replicas. To the right of the optimum, the number of replicas gets smaller. The request cost starts to dominate while the replication costs vanishes as the minimum of one replica is reached (around $\rho = 6$). Thus, the overall cost converges.

6.3.5 *Replica Size.* Figure 15(c) shows the behavior of all algorithms as the replica size σ is increased. The optimal and the random algorithm both remain at a constant cost level since they do not migrate any replicas. The important result that can be derived from the graph is the point at which the cost of our online algorithm grows beyond that of random placement. This is the case at a replica size of about 50,000. What does this mean in practice? Following the rational presented in Section 3.3, a replica may store up to this many data items, and still the online algorithm outperforms the random algorithm even though it executes migrations. This value seems adequate to allow a number of useful services in an ASG environment.

Keep in mind that this is a rather pessimistic approximation since the random algorithm knows the optimal number of replicas m_{opt}. A random algorithm that has to guess the best number of replicas in some way would produce an even higher cost. The cost produced by the online algorithm grows at a rate of approximately 0.074 points per 2,500 points increase in replica size. This data indicates that even if a more realistic random algorithm would increase the cost only by 10% over the one used in this measurement, the point up to which dynamic replica distribution is beneficial raises by approximately 32% to an absolute value of about 66,000.

The second important result presented in Figure 15(c) is the fact that the growth in cost of the greedy algorithm is superlinear at first and reaches a state of saturation at about the 20,000 mark. The greedy algorithm is superior to our online algorithm only for rather small replica sizes. The basic approach of the greedy algorithm is to position each replica optimally with respect to the previous allocation and keep all previous positions fixed. Therefore, the placement of the second and all following replicas has only limited benefits since it only shortens the request paths for the clients in parts of the network. The online algorithm is able to move existing replicas in reaction to newly placed ones. Thus, a new replica improves the efficiency throughout the network. At a replica size of 20,000, the greedy algorithm stops replicating completely and converges to a configuration with a single replica.

6.3.6 *Write Ratio.* Figure 15(d) shows how the algorithms perform under an increasing write ratio w. At $w = 0$, no write operations (also called *writes*) are produced by clients and, thus, no significant replication costs are produced by replicas. At a write ratio of 1, every operation submitted by a client is a Write and the replication cost reaches its maximum.

As one might expect, the cost produced by the online algorithm is considerably higher than that of the optimal algorithm. However, up to a write ratio of 0.5, it is better than the greedy and the random algorithm. Only at extremely high ratios ($w > 0.5$), the greedy algorithm is better since it sticks to a single replica from there on. The optimal algorithm also converges to a single-replica setup. The random algorithm, being agnostic of the consequences of high ratios, performs worst. Only as it is forced to use a single replica (due to the fact that it uses the replica number found by the optimal algorithm) its cost reduces again and approaches that of the online algorithm. It should be noted, though, that ratios above 0.2 are rather unusual for normal applications.

6.3.7 *Scalability.* Figures 15(e) and 15(f) depict the results of two scalability experiments. In Figure 15(e), we used networks of 50 nodes and increased the number of clients. All algorithms scale linearly with the online algorithm being close to the greedy one. The optimal algorithm scales best and the random algorithm scales worst. In Figure 15(f), we increased the number of nodes and, proportionally, we also increased the number of clients. Thus, the whole scenario was scaled up in this experiment. Since the average distance between replicas and the number of write operations are both proportional to the number of nodes in this experiment, the resulting cost is growing quadratically. The same general picture unfolds as in the previous experiment: The greedy and the online algorithm are relatively close together with the greedy one growing less rapidly. Regarding that the greedy algorithm was designed for large-scale infrastructures and that the online algorithm is intended to work at a medium scale, the scalability of our online algorithm is surprisingly good.

6.4 Applicability Assessment

An important issue with respect to the applicability of our placement algorithm is its appropriateness for different classes of services. Since we propose a system that replicates and migrates services at runtime, it is intuitively clear that this sets some limits to its applicability. The key factors in this context are:

(1) the requirement to keep a number of replicas consistent with each other, and
(2) the fact that we have to transmit a replica's state for a migration.

A consistency protocol that is used to transfer updates between the replicas induces more costs if the write ratio increases. There is a point at which the creation of replicas is not beneficial anymore because the high reconciliation traffic between them outweighs the reduction in request traffic. A similar argument can be made for the migration cost induced by the size of a replica's state: Depending on the choice of the adaptation interval, there is a certain value for the state size at which migrations do not amortize anymore over the next interval.

Thus, the set of conditions under which runtime replication and migration is beneficial critically depends on the write ratio w and on the average replica size σ. But what does "beneficial" actually mean? As we stated in the discussion on the benchmark algorithms, the random algorithm is really the only realistic contestant. The other algorithms use global knowledge, either in space (greedy algorithm) or in time and space (optimal algorithm). Therefore, they do not match the requirements defined for the ASG environment (refer to Section 3.1). Furthermore, the random algorithm, as we have defined it, has an advantage over a real random algorithm in that it knows the optimal number of replicas. Based on these facts, we choose two algorithms as a yardstick for measuring the applicability of our approach depending on w and σ:

(1) the random algorithm as we have defined it (called $\mathcal{R}$ hereafter), and
(2) an additional algorithm that we call *random single* (called $\mathcal{Q}$ hereafter).

The *random single algorithm* has no information at all. It does not know the optimal number of replicas. Therefore, it is very conservative and chooses to place a single replica only. Since it also has no information on the request patterns in the network, it places its sole replica randomly.

6.4.1 *Defining Applicability.* We say that our system is applicable for a pair (w, σ), if the cost $C^{\mathcal{A}}(w, \sigma)$ produced by online placement algorithm $\mathcal{A}$ lies below the cost $C^{\mathcal{R}}(w, \sigma)$ of the random algorithm and $C^{\mathcal{Q}}(w, \sigma)$ of the random single algorithm respectively.

We evaluate the applicability of $C^{\mathcal{A}}$ based on the data presented in Figures 15(c) and 15(d). These graphs show that $C^{\mathcal{A}}$ grows linearly with w and with σ. A linear regression yields the following linear equations:

$$C^{\mathcal{A}}(w) = a_1 + b_1 \cdot w \tag{14}$$

$$C^{\mathcal{A}}(\sigma) = a_2 + b_2 \cdot \sigma \tag{15}$$

with

$$a_1 = 3.229,\ b_1 = 4.282,\ a_2 = 3.189,\ \text{and}\ b_2 = 3.048 \cdot 10^{-5}. \tag{16}$$

To simplify the derivation of $C^{\mathcal{A}}(w, \sigma)$, we assume that both y-intercepts are equal to their arithmetic mean: $\overline{a} = a_1 = a_2 = 3.21$. This introduces only a very small offset of about 0.02 and -0.02 to the actual curves. Now, the cost produced by $\mathcal{A}$ for any combination (w, σ) is given by

$$C^{\mathcal{A}}(w, \sigma) = \overline{a} + b_1 \cdot w + b_2 \cdot \sigma. \tag{17}$$

Since $C^{\mathcal{R}}(w, \sigma)$ is independent of σ (no migrations in the random algorithm), we simply have $C^{\mathcal{R}}(w, \sigma) = C^{\mathcal{R}}(w)$. Since the random single algorithm is independent of the replica size σ (no migrations) and also of w (no replication), $C^{\mathcal{Q}}(w, \sigma)$ has a constant value. Experiments yielded $C^{\mathcal{Q}}(w, \sigma) = 6.97$

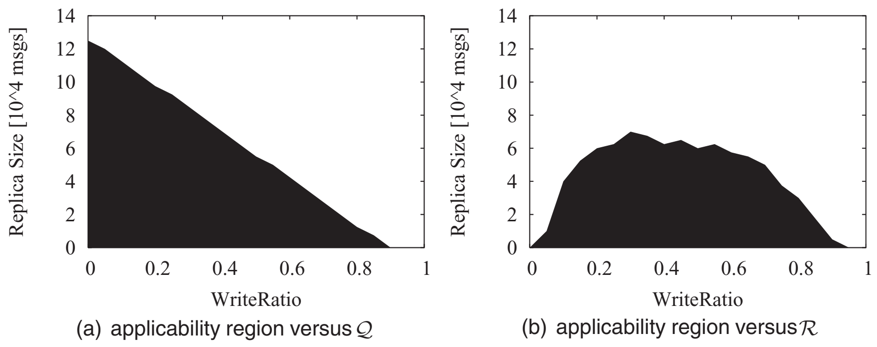

Fig. 16. Applicability regions in the space spanned by w and σ.

with a 95% confidence interval of 0.02. $\mathcal{A}$ is applicable versus $\mathcal{R}$ for (w, σ) iff $C^{\mathcal{A}}(w, \sigma) \leq C^{\mathcal{R}}(w, \sigma)$. The same statement can be made for $\mathcal{Q}$. We call this the *applicability condition*.

6.4.2 *Applicability Analysis.* Figure 16 depicts the results of the applicability analysis. These graphs show the regions of the parameter space spanned by w (x-axis) and σ (y-axis) that represent combinations of w and σ for which our applicability condition holds with respect to $\mathcal{Q}$ (Figure 16(a)) and $\mathcal{R}$ (Figure 16(b)).

In Figure 16(a) the points on the cost plane defined by Eq. (17) are separated linearly into the ones that have a cost smaller than the constant value of $C^{\mathcal{Q}}(w, \sigma) = 6.97$ and those whose cost is higher. $C^{\mathcal{R}}(w, \sigma)$ is not constant, as can be verified in Figure 15(d). As the write ratio increases, $C^{\mathcal{R}}(w, \sigma)$ first increases and then, for values higher than $w = 0.7$, it decreases again. As a result, we have no simple linear separation of the parameter plane. Due to the low cost of the random algorithm for a low write ratio, the acceptable replica size is small for small values of w. Then, it increases and decreases again analogously to $C^{\mathcal{R}}(w, \sigma)$ in Figure 15(d).

We conclude that a possible replica size in the order of 10^4 times the size of an average client request message and achievable write ratios of well beyond 20% should cover some useful services.

Note that the classification depends on the choice of system parameters. If, for example, the adaptation interval is increased, then the general level of replica sizes that are acceptable rises since the migration cost has more time to amortize. As a result, both classification regions in Figure 16 are shifted upwards and the class of settings for which our algorithm is applicable is increased. Of course, a larger adaptation interval also means that the algorithm needs longer to converge to a stable, low-cost configuration. In general, the classification regions change if system parameters are changed from their default values (see Table II). However, the applicability analysis tool introduced here can be used to analyze applicability under any setting of these parameters. Therefore, if there is a specific setting that is implied by a specific application or deployment of the ASG system, it can be tested with this

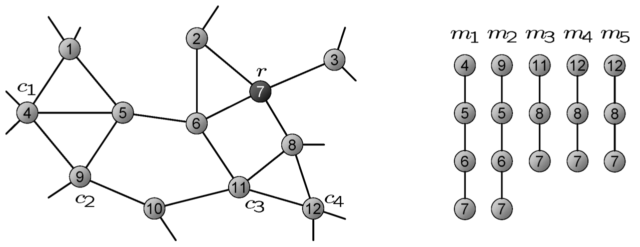

Fig. 17. Flow tree update: example network and client request paths.

analysis tool to assess whether our placement algorithm is beneficial under these conditions.

7. FLOW TREE ALGORITHM

The fact that our algorithm is inherently restricted to one-hop adaptations limits the speed at which it can approximate the optimal configuration starting from the initial configuration. The question arises as to whether an algorithm that may apply multihop adaptations can achieve a faster convergence while preserving the approximation quality, the adaptability, and the stability of the *one-hop algorithm*.

In this section, we will briefly introduce a variant of the algorithm that is allowed to migrate replicas over more than one network hop in a single adaptation. For this purpose, we introduce the concept of a *flow tree*. We assume that in each message, its path through the network is recorded as a list of node IDs. For a replica r, the paths of all messages it receives end with v (the node that r is running on). Therefore, the combination of all paths results in a tree rooted at v. Each replica records all message paths and maintains such a flow tree. Each node in the flow tree data structure additionally holds a counter. When a new path is added to the tree, the counters of all nodes that already exist in the tree are incremented while new nodes are added and initialized with a counter of one. This counter is also called the *weight* of the node in the tree. The product of the depth and the weight of a node

$$\mathcal{F}(u) = d(u) \cdot w(u) \tag{18}$$

is denoted as its *flow measure*. This value is equal to the number of message transmissions contributed by the node u since the $w(u)$ messages had to traverse $d(u)$ hops to get from u to v.

Figure 17 shows a sample network with one replica and four clients. Additionally, the paths of five request messages are depicted (right side).

Figure 18 depicts how a flow tree is incrementally created from the message paths. The numbers in parentheses are the node weights.

The flow tree algorithm applies exactly the same rules as the one-hop algorithm. However, the selection of target nodes is now based on the flow tree. For a migration target, this means that instead of simply choosing the dominant neighbor node u, the flow tree algorithm inspects the subtree rooted at u and chooses the nodes with the highest flow measure in this tree. For example, in

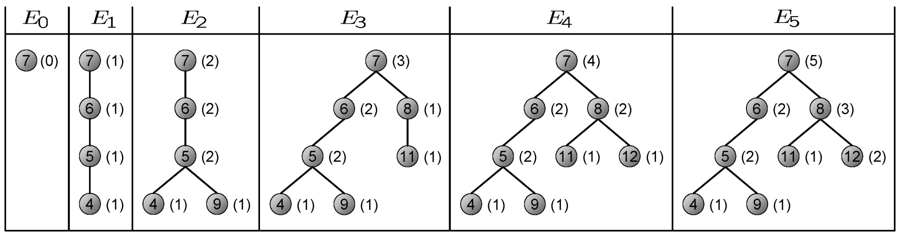

Fig. 18. Flow tree update procedure.

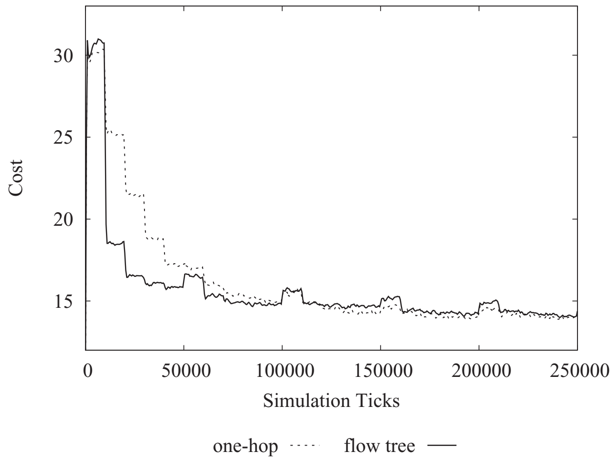

Fig. 19. Speed of convergence for one-hop and flow tree algorithm.

the rightmost tree in Figure 18, node 8 is the dominant node and node 12 is the node with the highest flow measure ($\mathcal{F}(u) = 2 \cdot 2 = 4$) within this subtree. Thus, instead of migrating the node 8, the replica would jump directly to node 12. The same principle is applied in case of a replication: If the one-hop algorithm would have chosen nodes u and w as migration targets, the flow tree algorithm chooses the nodes with the highest flow measure within the two subtrees rooted at u and w.

Figure 19 shows how the cost is reduced under both algorithms over time. It demonstrates that the flow tree algorithm indeed does improve the speed at which the cost is reduced. It also shows that the same general level of cost is obtained. Thus, the ability to make jumps does not effect the stability of the algorithm to a great extent. Otherwise, the excessive adaptations would produce major additional costs that would show up in the diagram. In fact, the costs exhibited by the flow tree algorithm are a bit higher than those of the one-hop algorithm as a steady state is reached (after 100,000 ticks). This can be attributed to the ability of the one-hop algorithm to approach the minimum-cost configuration without the danger of overshooting the optimum. Similar results related to different step widths have been obtained for general optimization techniques (e.g., gradient descent).

8. RELATED WORK

The optimal placement of objects in a computer network according to some optimality criterion has been studied in different contexts. Existing approaches can broadly be characterized into two categories: *service placement* and *resource placement*.

Existing algorithms for the placement of services are mostly targeted towards the Internet. The preferred application domain in this area is the Web. There are numerous approaches to placing Web servers or Web proxies such that some performance measure is optimized [Jia et al. 2001a, 2001b; Li et al. 1998; Li and Shen 2004; Tenzakhti et al. 2004; Qiu et al. 2001; Kalpakis et al. 2001; Tang et al. 2007]. The placement algorithms applied in all of these cases use global knowledge about the topology of the network and about client requests. Other approaches are presented in the domains of *service grids* [Lee and Weissman 2001; Andrzejak et al. 2002] and *Service Overlay Networks* (SON) [Liu et al. 2004; Choi and Shavitt 2001]. These systems also assume a global view on the network and some centralized decision entity that solves the NP-hard problem heuristically. Moreover, most of these approaches do not consider stateful services that require specific measures for preserving consistency. Kalpakis et al. [2001] and Tang et al. [2007] do investigate replica state and the associated consistency problem. However, they still assume a global view.

The problem of resource placement is closely related to the service placement problem. However, a service placement algorithm assumes a rather coarse-grained view by placing a whole set of resources (data, documents, etc.) provided by a single service. In contrast to that, we use the term resource placement to denote the placement of individual resources like, for example, Web pages. Centralized algorithms have been proposed for placing resources inside content distribution networks [Kangasharju et al. 2002] and on download mirrors [Cronin et al. 2002; Radoslavov et al. 2001]. Coppens et al. [2005] present an architecture and algorithms for a semi-centralized content delivery service that achieves a better distribution of processing load but still suffers from the problems of most centralized approaches. Ko and Rubenstein [2004] introduce a fully decentralized resource placement system that is based on the coloring of network nodes. However, their approach assumes that clients are static and that the distances between clients and resource servers do not change frequently. Moreover, the periodic exchange of color information implies considerable levels of network load. Rabinovich et al. propose the RaDaR architecture for monitoring a large pool of servers and migrating and replicating resources among them. RaDaR employs routing information extracted from the OSPF (Open Shortest Path First) routing system to inspect the paths of resource requests and uses them for placement decisions. The system is tailored to immutable resources like Web pages. Therefore, its restrictions on the number of replicas are much weaker than in the ASG, and the cost of consistency is not an issue.

We conclude that the existing approaches to object placement do not meet the requirements defined in Section 3.1. More specifically, none of them is decentralized and considers stateful objects at the same time.

9. CONCLUSIONS AND FUTURE WORK

We have presented an algorithm for the self-organizing service replica placement in an Ambient Intelligence environment. Existing algorithms are either centralized and assume global knowledge of the network topology and the client behavior, or they ignore the fact that services may be stateful. Our algorithm is fully decentralized and explicitly supports stateful services that require a consistency protocol to be run between the replicas. We have shown that two simple, interacting feedback rules suffice to let an arbitrary set of replicas distribute to cover an entire ASG network in a cost-efficient way. The rules for triggering replicas to migrate and replicate are comparably simple. However, the emergent effects produced by their repetitive application in a number of replicas lead to a complex and feature-rich behavior and eventually produce the desired result. We have discussed these effects in detail. Coordinated service placements are achieved without any central coordinator, based on local knowledge and local actions, without requiring external control, and without any extra communication. Replicas simply inspect incoming message flows and adapt by migrating, replicating, or dissolving. This results in a system that finds a stable, low-cost replica placement fast while still being able to adapt quickly if relevant changes occur in the environment.

Our evaluation shows that our system is applicable up to replica sizes and write ratios that enable numerous useful services to be run in an ASG. Over a wide range of parameter settings, it outperforms both the well-known greedy placement algorithm by Qiu et al. [2001] as well as different random placement algorithms that would be applied in an ad hoc fashion. Furthermore, it scales acceptably well. Another big advantage of our algorithm over its rivals is its ability to find the adequate number of replicas autonomously. By employing the replication radius parameter, it divides the network into cells that are covered by individual replicas. Within each cell, the replica can react to changes by migrating. A further improvement of the algorithm in terms of convergence speed has been achieved by enabling replicas to migrate over distances of more than one hop in a single adaptation.

We conclude that our placement system fulfills all requirements defined in Section 3.1. It represents an important contribution for the realization of AmI infrastructures that enhance the user's interaction with his current physical environment in an unobtrusive and self-organizing way. Furthermore, the simplicity of the applied rules and the way in which useful complex dynamics emerge from their interaction may serve as an example for the development of similar systems that require autonomous operation.

Until now, we have evaluated the system for an artificial application that reflects our basic assumptions. This has two reasons: (1) it allows us to argue independently of any concrete application and its idiosyncrasies, and (2) the efficiency of the test application enabled us to run a large number of simulations in order to produce statistically relevant results. The next step would be to implement a number of real services in the ASG to verify our results. Furthermore, we are in the process of generalizing the self-organizing placement concept to other application domains. We believe that it has great potential for improving replication and replica placement, for example, in content delivery networks and peer-to-peer systems.

REFERENCES

Andrzejak, A., Graupner, S., Kotov, V., and Trinks, H. 2002. Algorithms for self-organization and adaptive service placement in dynamic distributed systems. Tech. rep. HPL-2002-259, HP Laboratories Palo Alto.

Chandra, R., Qiu, L., Jain, K., and Mahdian, M. 2004. Optimizing the placement of integration points in multihop wireless networks. In *Proceedings of the 12th IEEE International Conference on Network Protocols*.

Choi, S. and Shavitt, Y. 2001. Placing servers for session-oriented services. Tech. rep. WUCS-2001-41, Department of Computer Science, Washington University.

Coppens, J., Wauters, T., Turck, F. D., Dhoedt, B., and Demeester, P. 2005. Evaluation of replica placement and retrieval algorithms in self-organizing CDNs. In *Proceeding of the IFIP/IEEE International Workshop on Self-Managed Systems & Services (SelfMan'05)*.

Cornuéjols, G. P., Nemhauser, G. L., and Wolsey, L. A. 1990. The uncapacitated facility location problem. In *Discrete Location Theory*, Wiley, 119–171.

Cronin, E., Jamin, S., Jin, C., Kurc, A., Raz, D., and Shavitt, Y. 2002. Constrained mirror placement on the Internet. *IEEE J. Select. Areas Comm. 20*, 7, 1369–1382.

Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., and Burgelman, J.-C. 2001. Scenarios for ambient intelligence in 2010. Tech. rep., The IST Advisory Group (ISTAG).

Herrmann, K. 2006. Self-Organizing infrastructures for ambient services. Ph.D. thesis, Berlin University of Technology.

Herrmann, K. 2007a. Group anti-entropy—Achieving eventual consistency in mobile service environments. In *Proceedings of the 8th IEEE International Conference on Mobile Data Management (MDM'07)*. IEEE Computer Society Press.

Herrmann, K. 2007b. Self-Organizing replica placement—A case study on emergence. In *Proceedings of the 1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society Press.

Herrmann, K., Geihs, K., and Mühl, G. 2004. Ad hoc service grid—A self-organizing infrastructure for mobile commerce. In *Proceedings of the IFIP TC8 Working Conference on Mobile Information Systems (MOBIS'04)*. IFIP—International Federation for Information Processing, Springer-Verlag.

Herrmann, K., Mühl, G., and Jaeger, M. A. 2005. A self-organizing lookup service for dynamic ambient services. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE Computer Society Press, 707–716.

Jia, X., Li, D., Hu, X., and Du, D. 2001a. Optimal placement of Web proxies for replicated Web servers in the Internet. *The Comput. J. 44*, 5, 329–339.

Jia, X., Li, D., Hu, X., and Du, D. 2001b. Placement of read-write Web proxies in the Internet. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*. 687–690.

Kalpakis, K., Dasgupta, K., and Wolfson, O. 2001. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. Parallel Distrib. Syst. 12*, 6, 628–637.

Kangasharju, J., Roberts, J., and Ross, K. W. 2002. Object replication strategies in content distribution networks. *Comput. Comm. 25*, 4, 367–383.

Karlsson, M. and Karamanolis, C. 2004. Choosing replica placement heuristics for wide-area systems. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. 350–359.

Ko, B.-J. and Rubenstein, D. 2004. Distributed server replication in large scale networks. In *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM Press, New York, 127–132.

Kraemer, R. and Schwander, P. 2003. Bluetooth based wireless Internet applications for indoor hot spots: Experience of a successful experiment during CeBIT 2001. *Comput. Netw. 41*, 3, 303–312.

LEE, B.-D. AND WEISSMAN, J. B. 2001. Dynamic replica management in the service grid. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC'01)*.

LI, B., DENG, X., GOLIN, M. J., AND SOHRABY, K. 1998. On the optimal placement of Web proxies in the Internet. In *Proceedings of the IFIP TC-6 8th International Conference on High Performance Networking (HPN'98)*. Kluwer Academic Publishers, 485–495.

LI, K. AND SHEN, H. 2004. Optimal placement of Web proxies for tree networks. In *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'04)*. 479–486.

LIU, K. Y., LUI, J. C., AND ZHANG, Z.-L. 2004. Distributed algorithm for service replication in service overlay network. In *Proceedings of the 3rd IEEE/IFIP TC-6 Networking Conference (Networking'04)*.

MILNOR, J. 1985. On the concept of attractor. *Comm. Math. Phys. 99*, 2, 177–195.

PERKINS, C. E. AND BHAGWAT, P. 1994. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of the ACM SIGCOMM*. 234–244.

QIU, L., PADMANABHAN, N. V., AND VOELKER, M. G. 2001. On the placement of Web server replicas. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (InfoCom)*. 12–22.

RADOSLAVOV, P., GOVINDAN, R., AND ESTRIN, D. 2001. Topology-Informed Internet replica placement. In *Proceedings of the 6th International Workshop on Web Caching and Content Distribution*.

REPAST. 2009. The RePast website. http://repast.sourceforge.net/.

SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. 2005. Latency-Driven replica placement. In *Proceedings of the International Symposium on Applications and the Internet (SAINT)*. 399–405.

TANG, X., CHI, H., AND CHANSON, S. T. 2007. Optimal replica placement under TTL-based consistency. *IEEE Trans. Parallel Distrib. Syst. 18*, 3, 351–363.

TENZAKHTI, F., DAY, K., OULD-KHAOUA, M., AND OBADIAT, M. S. 2004. Placement of Web proxies with server capacity constraints. In *Proceedings of High Performance Computing Symposium (HPC'04)*. Kluwer Academic Publishers, 101–106.

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM Press, New York, 172–182.

WEISER, M. 1991. The computer for the 21st century. *Sci. Amer. 265*, 66–75.

Received August 2007; revised April 2008; accepted December 2009