Self-Managing Federated Services[*]

Francisco Matias Cuenca-Acuna and Thu D. Nguyen
{*mcuenca, tdnguyen*}*@cs.rutgers.edu*

Technical Report DCS-TR-541
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854
December 22, 2003

Abstract

We consider the problem of deploying and managing replicated highly available services in *federated systems* that span multiple collaborative organizations. In particular, we present a peer-to-peer framework that supports the construction of self-managing replicated services that automatically adjust the number of replicas and their placement in response to changes in the system or client loads. Our framework is novel in that self-management is completely decentralized, depending only on a modest amount of loosely synchronized global state. More specifically, each replica is wrapped with a management agent that monitors the state of the service and the underlying system. Each agent occasionally searches for configurations that may be better than the current one with respect to an application model and explicitly stated performance and availability targets. On finding a better configuration, an agent will enact the new configuration after a random delay to avoid possible collisions. We evaluate our framework by studying a prototype UDDI service subjected to various system changes as well as client load. We show that while agents act autonomously, the service rapidly reaches a stable and appropriate configuration in response to system dynamics.

1 Introduction

We consider the problem of deploying and managing replicated highly available services in *federated systems* that span multiple collaborative organizations. Rising Internet connectivity and emerging web service standards and middleware are enabling a new federated computing model, where computing systems will be comprised of multiple components distributed across multiple collaborative organizations. The emergence of federated computing is already evident at several levels of collaboration, including peer-to-peer (P2P) information sharing, scientific computing grids, and e-commerce services.

While federated computing can revolutionize collaboration across the Internet, currently, deploying and managing federated services can be an arduous task. In particular, applications must be able to execute in heterogeneous and potentially highly dynamic environments that are not under the control or management of any single entity. Consider services such as the infrastructural Universal Description, Discovery and Integration (UDDI) service that must be replicated across multiple sites to ensure high performance and availability. Such multi-site services must currently be configured and maintained by hand, placing a considerable burden on

[*]This work was supported in part by NSF grants EIA-0103722 and EIA-9986046.

system operators [18]. Worse, it is quite difficult to understand and properly manage the overall performance and availability of such a service because the components are spread across multiple administrative domains. A concrete example is the European Data Grid [1], which has around 160 machines (some of them clusters) federated across Europe. This system relies on a continental scale directory service comprised of more than 100 data providers. This directory service is vital to the operation of the grid yet its topology and redundancy must be manually configured. Any adjustments or changes must also be performed manually.

In this paper, we describe a distributed resource management framework that can be used to build self-managing multi-site services that dynamically adapt to changing system configuration and client load. Our goal is to reduce the burden of deploying and managing a multi-site service to three tasks: (1) defining an application model for the framework to choose appropriate runtime configurations and specifying the desired availability and performance goals, (2) deciding the set of machines that can host instances of the service, and (3) providing maintenance support to repair these machines when they fail. Given a set of hosting machines, our framework will start an appropriate number of service replicas to meet the desired quality of service. Further, it will monitor the application as well as the hosting infrastructure; as the membership of the hosting set, processing capabilities and availability profiles of hosting machines, and client load change, our framework will adjust the number and placement of replicas to maintain the target quality of service.

Our management framework is novel in that it is completely distributed; each replica of a service is wrapped with a management agent that makes autonomous decisions about whether new replicas should be spawned and whether the current replica should be stopped or migrated to a better node. Agents rely on a set of loosely synchronized data to make their decisions, including the client load each replica is currently serving and load information about potential host machines. Critically, this information is only loosely synchronized with weak consistency using a gossip-based publish/subscribe infrastructure; the global state maintained at each node can be out-of-date by minutes. This autonomy and dependence only on loosely synchronized global state make our framework scalable and highly robust to the volatility inherent within federated systems. Despite autonomous actions based on weakly consistent data, we show that our management framework reaches a stable and appropriate configuration rapidly in response to system dynamics.

As a proof of concept, we have implemented a prototype self-managing replicated UDDI service using our framework. This test application is challenging to manage because increasing the number of replicas also increases the overheads of writes into the replicated database. Thus, our manager cannot take the conservative

[1]http://www.eu-datagrid.org/

approach of over-replicating to meet the desired quality of service. Also, as already stated, a UDDI service is a critical infrastructural service that can determine the availability of the overall federated system and so makes a good test case for our framework. Modifying this application to work with our framework was simply a matter of wrapping the back-end database with 270 lines of commented Java code.

A significant contribution that is not described in detail here is the infrastructure that we have built to ease the task of remotely deploying and managing federated services. For example, in addition to the self-management, operators can use our framework to remotely manage and monitor services running on multiple sites.

We evaluate our prototype by studying its behavior in response to a simulated workload on two distinct testbeds. The first is a LAN-connected cluster environment that allows us to easily understand the behavior of our framework in a controlled environment. The second is the PlanetLab testbed[2], which consists of nodes widely distributed across the WAN. PlanetLab presents an extremely challenging environment for our framework because it is heavily loaded and very volatile in addition to being widely distributed. In fact, we expect that PlanetLab currently presents a more challenging environment than what would reasonably be expected of real federated systems hosting highly available services. Our results show that the self-management framework is quite stable despite its decentralized nature, yet adapts quickly and effectively in response to changes in load.

2 PlanetP

We begin by briefly describing PlanetP [6, 7], a gossiping-based publish/subscribe infrastructure, since we assume its use for maintaining the loosely synchronized global data needed for our self-management approach. Members of a PlanetP community publish documents to PlanetP when they wish to share these documents with the community. PlanetP indexes the published documents, along with any user-supplied attributes, and maintains a detailed inverted index *local* to each peer to support content searches. (We call this the *local index*.)

In addition, PlanetP implements two major components to enable community-wide sharing: (1) an infrastructural gossiping layer that enables the replication of shared data structures across groups of peers, and (2) a content search, ranking, and retrieval service. The latter requires two data structures to be replicated on every peer: a membership directory and a content index. The directory contains the name, address, and a set of application-defined properties for each current community member. The global content index contains term-to-peer mappings, where the mapping $t \rightarrow p$ is in the index if and only if peer p has published one or more documents containing term t. The global index is currently implemented as a set of Bloom filters [3], one per

[2]http://www.planet-lab.org/

peer that summarizes the set of terms in that peer's local index. All members agree to periodically gossip about changes to keep these shared data structures weakly consistent.

To answer a query posed at a specific node, PlanetP uses the local copy of the global content index to identify the subset of peers that contains terms relevant to the query and passes the query to these peers. The targeted peers evaluate the query against their local indexes and return URLs for relevant documents to the querier. PlanetP can contact all targets in order to retrieve an exhaustive list or a ranked subset to retrieve only the most relevant documents.

In order to use PlanetP's indexing capabilities to locate services, we have extended the concept of a document. Java objects can also advertise themselves as dynamic documents. If a client searches for a dynamic document it will receive a URL that can be used to perform an RMI call to the Java object.

Using simulation and measurements obtained from a prototype, we have shown that PlanetP can easily scale to community sizes of several thousands [7]. The time required to propagate updates, which determines the window of inaccuracy in peers' local copies of the global state, is on order of several minutes for communities of several thousand members when the gossiping interval is 30 seconds. Critically, PlanetP dynamically throttles the gossiping rate so that the bandwidth used in the absence of changes quickly becomes negligible.

3 Self Management Approach and Algorithms

In our self-management approach, each self-managing application must have a model that maps possible configurations, i.e., number of replicas and their placement, to "fitness" values. We call this the application fitness function (or model). Each replica of a service is then wrapped with a management agent. Each management agent independently monitors the service and adapts the number of replicas and their placement as needed to meet overall performance and availability goals as defined by the fitness function.

Broadly, our self-management algorithm works as follows:

- The set of nodes that can host services managed by our framework forms a PlanetP community, called the hosting community. Each node runs a monitoring agent that updates the node's processing capacity, availability, and current load when there are significant changes (these are defined as properties of a node and so changes are propagated via gossiping throughout the community).

- Each agent publishes the fact that a replica is running, and where it is running, when the replica (and the agent) is first started.

- Periodically, each agent searches for a better configuration using the application model and current information about the hosting community. If it finds a configuration with a fitness value that exceeds the current one by a threshold then it tries to deploy it. Before deploying a new configuration, however, the agent will wait a random delay period to avoid collisions. If, after the delay period, the agent observes that the service is still in the original configuration, then it proceeds with the deployment. Otherwise, it aborts the deployment.

Wrapping each replica of a service with a management agent is a fundamental decision of our framework. This decision implies that our framework must not limit the scalability of the service if it is to be applicable to a wide range of services. An alternative that avoids this coupling would have been to build a separate monitoring and management framework. We believe, however, that integrating the management framework with the application has two significant advantages. First, in the presence of failures, there will always be a management agent wherever a replica of the service is running. This means that, even when a network partition divides a service into several pieces, each piece can manage itself according to the semantics of the application, as defined by the fitness model, and the resource available. Second, our framework is unstructured and completely decentralized, making it highly robust even in very volatile environments.

In the remainder of this section, we discuss three critical components of our approach. First, we discuss the fitness model that we have developed for our example UDDI service; while this model was develop specifically for this application, the UDDI service is representative of a broad class of replicated applications to which this model may be applicable. Further, we explain the reasoning behind the model so that it can be applied to the construction of other models. Second, we describe the optimization algorithm that our framework uses to find good configurations. Finally, we describe how our decentralized decision-making algorithm works toward a coherent whole by choosing appropriate random delay times to avoid collisions.

3.1 An Example Application Model

We build our example model using availability and a single dimension of performance for simplicity. In particular, we focus on CPU-bound applications and build a model around CPU load and idleness. In general, for other applications, creating models based on other performance aspects such as I/O and networking would likely be necessary. We leave this as future work.

Each node's availability is tracked as the average fraction of time that that node is a member of the hosting community. Each node's processing capacity is estimated using a metric called bogomips. Bogomips is a portable metric used by the Linux kernel to assess any node's processing capacity. Typically, any machine running Linux has an entry estimating its bogomips in the file */proc/cpuinfo*. The processing capacity of a machine at any point in time is then defined as the estimated idleness of the CPU times the bogomips rating of the machine.

Given the above metrics for availability and performance and assuming that nodes' availability are not correlated, we then estimate the expected processing capacity of a set of nodes, called a configuration, using the following equation:

$$C(c) = \sum_{i=1}^{|c|} \begin{pmatrix} |c| \\ i \end{pmatrix} A_{avg}^{i} \left(1 - A_{avg}\right)^{|c|-i} i I_{wavg} \qquad (1)$$

where c is a configuration, $|c|$ is the number of nodes in c, A_{avg} is average availability of the nodes in c, and I_{wavg} is the weighted average of the current idle bogomips rating of all nodes in c. I_{wavg} is computed by weighing the idle bogomips of a node by its availability; this is necessary to prevent configurations with high performance but low availability machines from appearing overly attractive. Clearly, equation 1 is an approximation; in general, each node has a distinct availability and so an exact computation would require a sum across all possible combinations of nodes being up and down. This computation becomes expensive, however, as configurations grow in size and so motivates our approximation. We have found that a similar approximation is generally conservative and works well in driving replication in highly heterogeneous P2P environments [5].

In essence, equation 1 gives the approximate expected capacity if the service is required to stay in the same configuration through node failures. This may be the case if, for example, the hosting community is under high utilization. If excess capacity is available, however, and the startup time of a new replica is less than nodes' MTTRs, then equation 1 is pessimistic in that our framework will spawn additional replicas as needed to satisfy the offered load. This pessimism is desirable because, in general, we would prefer to exceed the availability target when resources are available rather than leaving resources idle and missing the availability target.

Given the above approach for estimating the capacity of a configuration, we then build a model that is generally suitable for a class of applications where:

- increasing the number of replicas increases overall throughput but can degrade the response time of a subset of the requests. Specific to the UDDI service, increasing the number of replicas will increase throughput because the workload is expected to be mostly reads, which only requires contacting one replica. The response time for writes will degrade, however, because writes must be applied to all replicas of the database.

- the service can only support a specific maximum number of replicas because of either administrative or scalability constraints. Specific to the UDDI service, the replicated database that we use is experimental and so does not work well beyond a certain number of replicas.

- the number of replicas should be adjusted according to the current load, increasing when the load increases and decreasing when the load decreases. There are two reasons to reduce the number of replicas under low load. First, it reduces the response time of the operations sensitive to the number of replicas. Second, it releases resources so that other services on the hosting community can use them if needed.

Given the above properties, we derive the following rules that our fitness model should obey. As before, let $C(c)$ be the expected capacity of configuration c as computed by equation 1, l the current client load, a the target availability level, $f(c, l, a)$ be the fitness of c given l and a, and $|c|$ be the number of nodes (equivalently replicas) in c, then for two different configurations c_1 and c_2:

1. if $C(c_1) \geq la$ and $C(c_2) < la$ then $f(c_1, l, a) > f(c_2, l, a)$;

2. if $C(c_1) \geq la$ and $C(c_2) \geq la$ but $|c_1| < |c_2|$ then $f(c_1, la) > f(c_2, la)$;

3. if $C(c_1), C(c_2) < la$ or $C(c_1), C(c_2) \geq la$ and $C(c_1) > C(c_2)$ then $f(c_1, l, a) > f(c_2, l, a)$;

4. if $C(c_1) < la$, $C(c_2) < la$ but $|c_1| > |c_2|$ then $f(c_1, l, a) > f(c_2, l, a)$.

One subtlety in the above rules is that, when computing $C(c)$ to compare against la, we must cap the idleness of each node by l. This is because once all client load has been directed to a single node because of failures, the excess capacity is no longer useful in serving that load. Thus, if the capped idleness gives a lower expected capacity than la, then over time, this configuration cannot meet our expected load at the target availability.

Finally, we assume that a maximum capacity is specified for a service when it is run. Figure 1(a) shows one possible fitness function that fits our criteria, where l is the current offered load, a the target availability, and l_{max} is the maximum expected offered load. To compute the fitness of a configuration c, we perform two computations. First, we compute $C(c)$ where the idle capacity of each node is capped by l. If this computation gives a bogomips value less than la, then we return the corresponding fitness value. If this computation gives a bogomips value greater than la, however, then we recompute $C(c)$ without capping the idle capacity of each node and return the corresponding fitness value. This differentiates between configurations that can meet the current load at the target availability yet have different amount of excess capacity that can be devoted to respond to increases in client load.

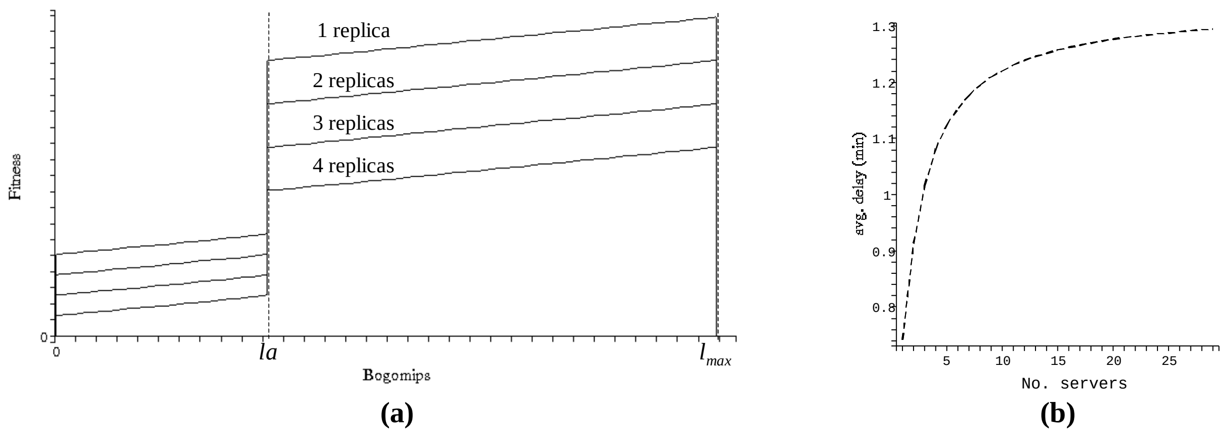


Figure 1: *(a) Fitness function used for the UDDI service. l is the current offered load, a the target availability, and l_{max} is the maximum expected offered load. Each curve represents a different number of replicas. (b) Average delay plotted against the number of agents for a community of 200 nodes running PlanetP with a gossip interval of 1 second. The probability of collision was set to 0.1.*

3.2 Finding the Right Configuration

Given a fitness function, we must still solve the optimization problem of finding the optimal (or a close to optimal) configuration. We have chosen to use a genetic algorithm because: (1) as explained below, the manner in which this algorithm evolves toward a good configuration seems to fit our domain well, and (2) if needed in the future, the algorithm can be easily parallelized with minimal inter-replica communication.

A typical genetic algorithm consists of three steps [9]. First, generate an initial population—in our case, this can just be a set of random configurations. Second, randomly select a pair of individuals, where the probability of choosing particular individuals is weighted by their fitness according to some fitness function. Third, produce the next generation by applying genetic operators to the selected pairs. Steps 2 and 3 are then repeated until a satisfactory offspring is found or a maximum number of generations has passed. In the latter case, the individual with the best fitness is chosen as the solution.

Genetic algorithms try to mimic the natural selection process by probabilistically allowing the most fit individuals to reproduce. The genetic operators most commonly used for reproduction are crossover and mutation. Crossover consists of taking a pair of individuals and crossing their chromosomes at a randomly selected point. For example, if the individual's chromosomes were represented as bit arrays this would be a valid crossover **11001**011+11011**111** = **11001111**. The idea behind crossover is that the offspring of two genetically fit individuals can inherit the best attributes of both parents, becoming an even more fit individual. Using the same example, mutation would be represented as the flipping of random bits. The purpose of mutation is to prevent populations from becoming overly specific, limiting the selection process to some local optimum.

Genetic algorithms seem well suited to our application domain. Similar to our previous example, we can use bit vectors to represent service configurations where each bit indicates the placement of a replica at a particular node. Then, since a good configuration will likely include highly available, powerful nodes that are lightly loaded, a crossover of two good configurations is likely to produce another good configuration. Further, mutations prevent the search from collapsing around a local optimum.

In our current implementation, each management agent periodically searches for a better configuration according to its view of the community (provided through PlanetP). We use a publicly available library called JGAP [3] that provides standard tools for running genetic algorithms. Because of the simplicity of our fitness model, each search starts from scratch and runs for a fixed number of generations.

More complicated models might benefit in the future by starting from the previous population since it will likely shorten the search. Further, as already stated, genetic algorithms can be parallelized by randomly exchanging individuals between nodes. In our infrastructure, this can be done easily by publishing promising individuals to PlanetP.

3.3 Realization of a Configuration

Because each manager agent in our framework operates autonomously, it is possible that several agents might try to deploy new configurations simultaneously. Concurrent deployments may interfere with each other, leading to too many or too few replicas. Particularly undesirable is when all agents conclude that the number of replicas should be reduced and simultaneously stop the local replica, leading to a service failure.

To address this problem, we adopt a randomized back off approach reminiscent of Ethernet-style back off to avoid transmission collisions. When an agent decides that a new configuration should be adopted, it waits a random delay time, t_d, chosen with a uniform density function from the interval $[0, T)$, before deploying the new configuration. After t_d, the agent rechecks the current configuration. If the configuration has changed in any way, it cancels its intended deployment. Otherwise, it proceeds with its deployment.

In essence, our randomized back off approach provides a probabilistic serialization of actions across the agents. Multiple agents reacting to the same changes in the community use the random delays to avoid colliding with each other.

[3] http://jgap.sourceforge.net/

Given a fixed T, what is the probability that two or more agents take concurrent actions if all of them simultaneously decide to deploy a new configuration? To answer this question, let N be the number of agents, v be the bound on information propagation time in PlanetP (also called the *period of vulnerability* since members may have inconsistent views of the community during this time), and t_0 be the minimum chosen delay time. As explained in Section 2, v is only a probabilistic bound but all of our experimentations have shown this bound to be quite accurate except in very overloaded conditions [7]. Since all agents choose delay times independently and each time within the interval $[0, T)$ is equally likely to be chosen, all outcomes are equally likely. Then, assuming that the delay times are discrete, we can compute the probability of concurrent action as follows:

$$p(N, T, v) = 1 - \frac{\mathit{NoCollision}(N, T, v)}{\mathit{AllOutcomes}(N, T, v)} \tag{2}$$

where *AllOutcomes* is the set of all possible outcomes and *NoCollision* is the number of outcomes where $\forall j, 1 \leq j \leq N-1, t_j \geq t_0 + v$, such that all agents except the one choosing delay time t_0 will cancel their planned deployment.

The number of outcomes that satisfy the *NoCollision* condition can then be counted as:

$$\mathit{NoCollision}(N, T, v) = N \sum_{i=0}^{T-v-1} (T - i - v)^{N-1} \tag{3}$$

where $T > v$. Basically, $(T - i - v)^{N-1}$ is the number of possible non-colliding outcomes when $t_0 = i$. The N factor arises from N possible agents that can choose t_0.

At runtime, we fix a probability of concurrent action, which is set to 0.1 in the experiments reported here, and vary T appropriately depending on the number of agents currently executing. Because Equation 2 does not have a closed form inverse, we have used the secant method to numerically compute T as the community changes.

To understand the delay that we might see in moving to a better configuration, we computed the average delay before a redeployment is enacted. In essence, this bounds the responsiveness of our system to changes, either in the client load or the server infrastructure. Figure 1(b) plots the results vs. the number of agents for a community of 200 nodes running PlanetP with a gossip interval of 1 second. This gives a v of 7 seconds. The probability of collision was set to 0.1. Observe that delays are under 2 minutes even for large numbers of replicas. Of course, delays heavily depend on the gossiping interval; if we back off of the somewhat aggressive interval of 1 second, the delays will increase.

Finally, observe that we also need to keep the redeployment time short to avoid collisions. The longer a

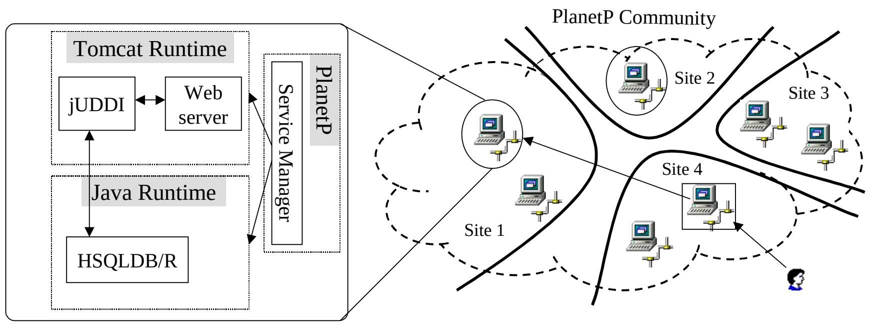


Figure 2: *This figure shows four sites sharing resources to support a federated UDDI service. The shared machines runs PlanetP to form the hosting community. Number and placement of replicas are automatically controlled by our framework; here, two replicas are shown running on the circled machines. Clients access the service through proxies that use PlanetP to locate a running instance.*

node takes to instantiate a new configuration, the more likely that other nodes will start to concurrently deploy different configurations.

A possible solution to this problem is to modify the fitness function to favor new configurations that are not too different from the current one. The disadvantage of this approach is that could limit the search, leading to the selection of sub-optimal solutions.

We instead introduce a deployment planning component to limit redeployment times without affecting the overall optimization algorithm. Specifically, our deployment planner will select a single step to enact per re-deployment in order to bring the current configuration closer to the new one. The assumption is that if the new configuration is significantly better than the existing one then the optimizer will find it in every search and therefore the system as a whole will eventually get to the new configuration.

In order the find the best step, we first compare the number of server nodes on the existing configuration against the new one. If one is greater than the other then we do a linear search to find which single node addition or deletion will produce the best result according to the fitness function. If the two configurations have the same size then we search for the best migration (i.e. adding and then deleting a node).

4 Architecture

In this Section, we describe our prototype implementation. Figure 2 shows the architecture of our prototype in the context of our UDDI test application: each instance of the service runs a multi-tier application comprised of the jUDDI [4] front-end, HSQLDB/R [5] (a replicated database) and PlanetP.

[4]http://www.juddi.org/
[5]http://www.javagroups.com/javagroupsnew/docs/hsqldbr.html

In this section, first briefly describe the runtime environments that we have developed for deploying PlanetP-enabled services. Then, we describe how we implement node monitoring, our self-management framework, and the UDDI service and client as PlanetP-enabled services.

4.1 Runtime Environments

Using ideas from Tomcat [6] and The Globus Toolkit [7], we package PlanetP-enabled applications inside jar files that include instructions on how they should be installed and executed. To run an application remotely, first a jar file is uploaded to the target node. Second, a loader service verifies that all the dependencies are fulfilled and downloads additional files if needed. Third, depending on the nature of the application a runtime environment is located and the binary code is handed to it. Optionally, an application can include a fitness model that will be handed to a local management agent for its execution.

Currently, we have implemented two different runtime environments. One is based on JWSDK [8] and Tomcat and is used to run Web Services like the jUDDI server. The other runtime, where most of our services run, is used for Java objects and is implemented similarly to an EJB container. This runtime shares the same JVM with PlanetP so it allows services to use all the functionality provided by it.

One important extension to PlanetP that was developed in this work was the support for services. Services are Java objects that advertise themselves using active documents and that can be accessed over the network using RMI. Active documents not only provide global naming, but also they are responsible for handing stub classes to clients so that they can communicate with servers at runtime. Effectively, PlanetP and active documents work as a completely decentralized RMI registry.

A further extension to the RMI model was the inclusion of global file handlers. In order to be able to pass file handlers on function calls across nodes, we extended Java's File class. This new abstraction provides the illusion of having read-only global files, which are very useful for example to share configuration files between management agents.

Although all the services that we have written communicate using RMI, this is not mandatory. Both runtime environments provide support to remotely start and stop applications.

[6] http://jakarta.apache.org/tomcat/
[7] http://www-unix.globus.org/toolkit/
[8] http://java.sun.com/webservices/jwsdp/index.jsp

4.2 Node Monitoring

We have implemented a Node Monitoring service to provide load and idleness information for evaluation of potential configurations. An instance of this service runs on each node of the hosting community and propagates static and dynamic information about the node (as node's properties that are gossiped throughout the community by PlanetP). Examples of static information include machine type, the amount of memory, and the CPU capacity (in bogomips). Examples of dynamic information include applications currently running on the node and current idleness.

To reduce the overhead of propagating dynamic information, we only publish changes that are considered stable and exceed a certain threshold. For example, in the case of CPU idleness, we use exponential smoothing ($alpha = .5$) and only publish a new value when the change is stable and exceeds 20%.

4.3 Application Deployment and Management

Whenever a service containing a fitness model is started at a node, a new a instance of the management agent is spawned. As already described, this agent is charged with the management of the service. In addition to pursuing performance and availability goals, however, the management agent is also responsible for monitoring the health of the local replica and presenting a fail-stop failure model for the replica. For example, in the case of the UDDI service, if the local database stops responding then the agent will stop the entire service and undeploy all the dependencies so that the node can return to a familiar state.

4.4 The UDDI Service

As already mentioned, we build our test UDDI service using three open source components: jUDDI, HSQLDB/R, and JGroups. jUDDI is a UDDI front-end written as a Java web service that can run on Tomcat. Internally, jUDDI stores all registry information in a database accessed through JDBC.

Although the UDDI v3 specification includes a replication protocol, this feature has not been implemented in jUDDI. We instead replicate our service using a replicated database. In particular, we use HSQLDB, which is a full SQL compatible database written in Java. The authors of JGroups [9], an open source group communication infrastructure, have developed an experimental replicated version of HSQLDB (HSQLDB/R) using JGroups.

Our development of this service only consisted of wrapping the database into a PlanetP service, which required 270 lines of (commented) code. The fitness model for this service is as described in Section 3.1.

[9] http://www.jgroups.org/

4.5 The UDDI Clients

To evaluate our framework, we have also written a client application that can impose load on the UDDI service. This client application itself is structured as a self-managing replicated service with a linearly increasing fitness function vs. the number of replicas up to a dynamically set maximum. Each replica generates a stream of requests with exponentially distributed inter-arrival times. These requests are load balanced across all the server replicas. In essence, this application really implements the proxies mentioned in Figure 2.

Client and service replicas are programmed to "repel" each other to avoid the need to carefully monitor the CPU usage of each application within the same JVM.

5 Evaluation

We now turn to evaluating the efficacy of our self-management framework. In particular, we study experimentally the responsiveness and stability of our management framework to changing client load as well as interfering external load and crashes of server nodes. We do not study the long term availability of the service because this would require running and loading the service experimentally for a long time[10]. We evaluate our framework using two distinct testbeds. The first is a LAN-connected cluster environment that allows us to easily understand the behavior of our framework in a controlled environment. The second is the PlanetLab testbed[11], which consists of nodes widely distributed across the WAN. PlanetLab presents an extremely challenging environment for our framework because it is heavily loaded and very volatile in addition to being widely distributed. In fact, we expect that PlanetLab currently presents a more challenging environment than what would reasonably be expected of real federated systems hosting highly available services. Thus, results showing that our framework works well on PlanetLab are very encouraging.

5.1 Experimental Environment

Our cluster environment is comprised of 2 clusters, one with 22 dual 2.8GHz Pentium Xeon 2GB RAM PCs and the second with 22 2GHz Pentium 4 512MB RAM PCs. The two clusters are interconnected with Gb/s and 100Mb/s Ethernet LANs. All machines run Linux. Each Xeon machine is rated at 5570 bogomips and each

[10]Alternatively, we could have studied this issue via simulation but decided against it because of the limited value of such a study. The only hypothesis that we would be able to validate is whether our approximation of a configuration's availability is sufficiently accurate. We already have good reason to believe that this approximation works well [5], however, and its accuracy can always be increased, say by using several averages computed over subsets with similar availability instead of just one global average.

[11]http://www.planet-lab.org/

Pentium 4 machine is rated at 3971 bogomips. Our experiments do not run sufficiently long so that meaningful availability measurements can be made for each machine. Thus, we arbitrarily set the availability of all nodes as reported by PlanetP to 90%.

PlanetLab is an open, globally distributed platform for developing, evaluating, and deploying planetary-scale network services. At writing time, PlanetLab has 336 nodes located throughout the globe. We run our experiments on 100 machines randomly selected from the set of .edu machines. We choose .edu machines because they are generally least restricted in terms of network ports being blocked by firewalls. PlanetLab nodes are highly heterogeneous, ranging in bogomips rating from 794 to 5570 bogomips. Again, we arbitrarily set the availability of all nodes to 90%.

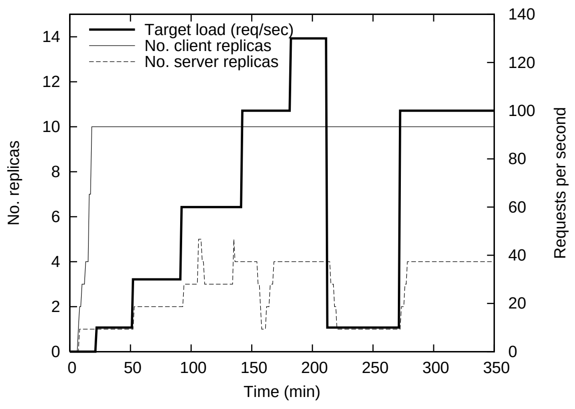
As already noted, PlanetLab presents an extremely challenging environment. In fact, we had to turn off the replication of the database for the UDDI service because it would not work properly on PlanetLab. Thus, when running on PlanetLab, the UDDI service consists of independent replicas that can only service read requests.

The fitness model for the UDDI service is constrained with a maximum desired performance of 60000 bogomips and a maximum number of 4 replicas. In order to have room for the CPU demand to grow, the model also requires new configurations to have 20% more CPU capacity than needed.

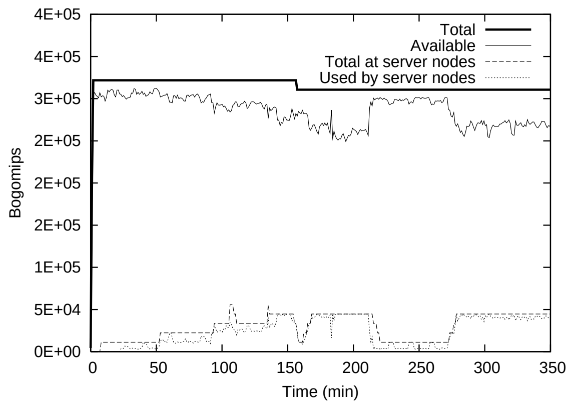
We limit the maximum number of service replicas to 4 mainly because the experimental replicated database does not scale well beyond this size. The client application is also constrained to a maximum of 10 replicas. Turning off the replication, we have experimented with maximums of 8 service and 20 client replicas and the results are essentially the same as those presented here.

To emulate a realistic service, we populate our UDDI service with actual data obtained from http://www.xmethods.org, a site that lists publicly available web services. This site has over 3000 service providers registered to provide over 400 services. This gave us a database of approximately 3MB in size. Clients were written to issue random *findBusiness* queries against this set of registered providers.

During each experiment, nodes individually log information about themselves and replicas running on them. Each node periodically contacts a time server and log timing deviations together with the round trip latency. At the end of the experiment, we collect the individual logs and merge them into a single ordered list of events. The accuracy of this merge is on order of two seconds. All reported data are averaged across 1 minute intervals to amortize inaccuracies introduced by skewed time lines.



(a)



(b)

Figure 3: *(a) Number of service replicas, number of client replicas, and client offered load plotted against experiment time. (b) Total system capacity, idle capacity, capacity allocated to the service, and capacity actually used by the service plotted against experiment time.*

5.2 Behavior on Cluster Testbed

Service Behavior. We begin by studying the adaptivity and stability of our framework when the service is running on the cluster testbed. The hosting community is comprised of all 44 machines in the two clusters. We start the experiment by deploying a single instance of the client and one of the service on two random machines. Given their respective fitness models, the client application quickly grows to 10 replicas while the service stays at 1 replica in the absence of load. Once the client stabilizes at 10 replicas, we instruct it to progress through several different levels of load and observe the behavior of the service.

The experiment has three phases. First, we slowly increase the load to observe the server's reaction. We start at 10rps and go up to 130rps in five steps taking a total of 200 minutes (time 0–200). At each step, a corresponding increase of 1 service replica on a Xeon node would be sufficient to handle the offered load. Second, we abruptly decrease the load from 130rps back to 10rps (time 200–250). Finally, we abruptly increase the load from 10rps back to 130rps in a single step (time 250–350).

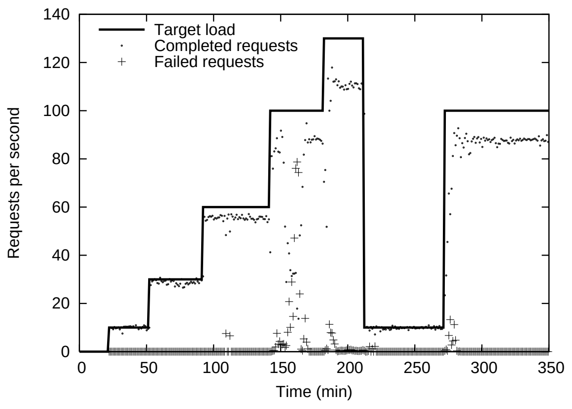
Figure 3(a) plots the behavior of the client and service while Figure 3(b) plots system capacity, capacity allocated to the service, and capacity actually used by the service. (While Figure 3 shows only one run of the experiment for clarity, we have repeated this experiment many times with essentially the same results.)

Based on Figure 3, we conclude that our framework is quite stable despite its decentralized nature, yet adapts quickly and effectively in response to changes in load. In particular, observe that throughout the experiment, the framework chooses configurations that match the current load quite well, almost always using the minimum number of required Xeon nodes. At each change in load during the first phase, our framework starts a new replica within 1 minute. (Note that the actual length of the adjustment interval is dependent on a number of environmental settings such as the gossiping rate. Thus, we are less concern with this magnitude as compared to whether the system makes the right decisions and its stability.) Even during large changes in load, our system responds smoothly although the adjustment time is of course longer. When load drops from 130rps to 10rps, our framework smoothly reduces the number of service replicas from 4 to 1 over 7 minutes. When load spikes from 10rps to 130rps, our framework again smoothly increases the number of service replicas back to 4 over 6 minutes. Note the longer adjust time of 6-7 minutes compared to the total adjust time of around 3 minutes in response to slow changes. This is caused by the fact that we limit the framework to one change per adjustment and wait at least 3 periods of vulnerability between adjustments (Section 3.3).

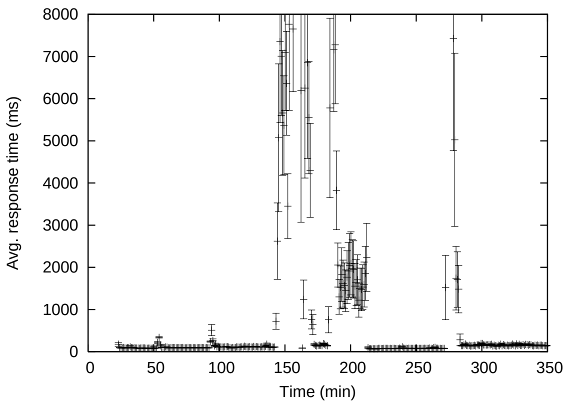
Of course, our framework is not perfect at all times due to its probabilistic nature and fluctuations in the observed load and system idleness. Between times 100 and 150, it twice deploys 5 replicas unnecessarily for short periods of time. Figure 3(b) shows that at these instances, the processing load shows peaks large enough to saturate the 3 replicas. Further, at each instance, two agents collided in their decisions to increase the number of replicas, resulting in the creation of 2 additional replicas instead of just 1. The agents quickly detects this over-replication, however, and stops 1 replica almost immediately.

Interestingly, at time 155, 3 replicas of the service fail due to a buffer exhaustion error inside JGroups triggered by the high load. This failure is reflected in the reduced total capacity of the system because PlanetP fails the entire run-time system to enforce a fail-stop failure model, thus effectively removing the failed nodes from the hosting set. After the failure, the management agent on the remaining replica simply pushes the service back to 4 replicas in order to meet the offered load. Note that even if all 4 replicas fail, our framework would have restarted the service because we always run at least one additional agent separate from the application being managed.

Impact of Self-Management on Clients. We now consider how the above service adaptation to changing load impacts the clients. In particular, Figure 4 plots throughput, the rate of failed requests, and average response time as seen by the clients. Observe that response time can increase noticeably during adjustment periods because of warm up effects and temporary overload, particularly when more than 1 replica needs to be started. Very few requests actually fail during adjustment, however. A few tens of requests were lost around time 275 when the load rose sharply from 10rps to 130rps. In addition, a few requests failed when an excess replica (started because of a collision) was stopped at time 106. Beyond this, all the lost requests were due to the service failure around time 155 and the service being loaded beyond its maximum capacity (as given by the cap of 4 replicas)



(a)



(b)

Figure 4: *(a) Target load, completed requests, and failed requests plotted against experiment time. Note that at high loads, the clients were unable to reach the target load even when there are no failures because of inaccuracies in the Java timer; threads were often not woken up quickly enough to generate the desired rate of requests. (b) Average response time for completed requests with 95% confidence bars.*

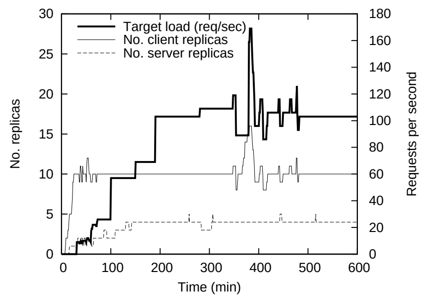
around time 200.

Response to Failures. We have also studied the behavior of our framework under node failure (as opposed to the application failure in the above experiment) as well as interfering external load on nodes hosting the service replicas. We do not show the results here because they are entirely as expected. When a node fails, one of the remaining agent will eventually detect the failure and starts a new replica when its PlanetP instance attempts to gossip to the failed node. We can speed up this detection by having the agent gossip heartbeats among themselves. When external load is placed on a node hosting a replica of the service so that it is overloaded, the agent quickly detects this overload and migrates the replica elsewhere.

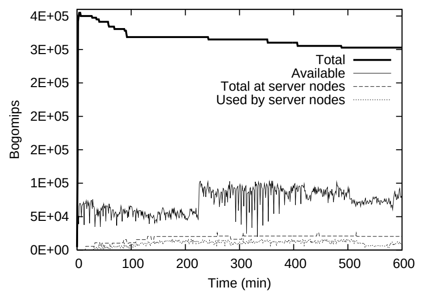
5.3 Behavior on PlanetLab

Figure 5 shows the results when we run a similar experiment on the PlanetLab testbed. In this experiment, we started loading the servers with 10rps, increasing to 130rps in five steps. As already noted, PlanetLab presents an extremely challenging environment for our self-management framework. In this environment, machines are so heavily loaded that it can take up to several minutes for PlanetP to complete a gossiping round (i.e. synchronize the state of two machines). Even worst, TCP SYN packets are frequently denied by nodes with backed-up accept queues. At the application level, this condition translates to having frequent temporal inconsistencies in the membership directory.

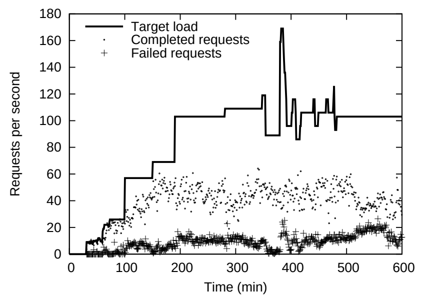
The above problem is particularly evident for the clients since their fitness model considers all configurations



(a)



(b)



(c)

Figure 5: *(a) Number of service replicas, number of client replicas, and client offered load plotted against experiment time. (b) Total system capacity, idle capacity, capacity allocated to the service, and capacity actually used by the service plotted against experiment time. (c) Target load, completed requests, and failed requests plotted against experiment time.*

with 10 nodes equal. Thus, replicas often land on heavily loaded nodes. For example between times 350 and 500, examining detailed logs revealed that the agents managing the client application were in frequent disagreement on the number of currently executing replicas. This is the cause of the instability for the client seen during this time period.

On the other hand, the UDDI servers are much more stable although they did make a number of adjustments between times 200 and 600, migrating replicas when hosting machines become overloaded and briefly reducing the number of replicas in the rare period of lower external load. The servers are more stable because their model rewards configurations containing less overloaded nodes. This allows PlanetP to more closely synchronize the shared global data. Note that because of the small amount of bogomips available per machine, the servers quickly reach the maximum number of replicas even when the client load is relatively low.

In summary, we find these results very encouraging. While there is much more instability compared to results from the cluster testbed, PlanetLab currently presents a very challenging environment. In fact, one would expect any practical environment hosting highly available services to be much less volatile than PlanetLab. Thus, the above results give us confident that our framework would work well in such realistic environments.

6 Related Work

Previous research [10, 17, 11, 8, 1, 14, 7, 16] have looked at building wide area infrastructures to simplify application development, deployment and management. Issues like communication [10, 12, 14], monitoring [17, 8], distributed data structures [19, 15], deployment [14, 1] and remote execution [8, 16] have been extensively studied. The focus of our work has been to provide a decentralized deployment and maintenance service that does not limit the scalability of the application being deployed. Previous work has either used centralized solutions [16, 8] or relied on group communication to coordinate deployment agents [14].

In our approach we use an application model, which could be derived from an SLA, to constantly improve the service configuration. Moreover, our deployment agents optimize the application model and deploy new configurations in a completely autonomous manner. Traditionally, the use of SLAs and quality of service metrics (QoS) has been applied on cluster environments [13, 2, 4] and therefore instantiating a configuration was not an issue.

Closer to our target environment, WebOS [16] provides infrastructure for building Internet scale services although its experimental studies have focused on applications that are embarrassingly parallel like HTTP proxies.

7 Conclusions and Future Work

In this work, we have described the design, prototype implementation, and evaluation of a decentralized framework that can be used to build self-managing replicated services that dynamically adapt to changing system configuration and client load. We specifically target federated systems, where applications must run in heterogeneous and potentially highly dynamic environments that are not under the control or management of any single entity. Statically configured and deployed services are unlikely to perform well in such dynamic environments.

Given a set of hosting machines, our framework will start an appropriate number of service replicas to meet the desired quality of service. Further, it will monitor the application as well as the hosting infrastructure; as the membership of the hosting set, processing capabilities and availability profiles of hosting machines, and client load change, our framework will adjust the number and placement of replicas to maintain the target quality of service.

The decentralized nature of our framework makes it highly robust to the volatility inherent within federated systems. Our experimental evaluation on two distinct testbeds, one being PlanetLab open planetary-scale testbed, validates this robustness, showing that the framework adapts efficiently in response to changes but is quite stable even when operating under very challenging conditions.

Finally, we conclude by observing that much future work remains. Our self-management infrastructure is a concrete first step toward autonomous federated services. However, our fitness model is quite simple at this point in time. We intend to explore compiler-based techniques for assisting developers to design fitness model. We will also need to explore models that include other performance and availability metrics than just expected

CPU processing capacity. Also, we intend to perform much larger experimental studies to validate the scalability of our approach. To date, we have been limited by the scalability of the existing implementation of the UDDI service we chose as the first test case.

References

[1] E. Amir, S. McCanne, and R. H. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Sept. 1998.

[2] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano-SLA based management of a computing utility. In *IEEE/IFIP Integrated Network Management Proceedings*, May 2001.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, June 2003.

[5] F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. Autonomous Replication for High Availability in Unstructured P2P Systems. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, Oct. 2003.

[6] F. M. Cuenca-Acuna and T. D. Nguyen. Text-Based Content Search and Retrieval in ad hoc P2P Communities. In *Proceedings of the International Workshop on Peer-to-Peer Computing (co-located with Networking 2002)*, May 2002.

[7] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, June 2003.

[8] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2001.

[9] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[10] S. Pallickara and G. Fox. NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Proceedings of the International Middleware Conference*, June 2003.

[11] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, Oct. 2002.

[12] L. Rodrigues, S. Handurukande, J. Pereira, R. Guerraoui, and A.-M. Kermarrec. Adaptive Gossip-Based Broadcast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2003.

[13] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[14] Y. shun Wang and J. Touch. Application deployment in virtual networks using the x-bone. In *Active Networks Conference and Exposition (DANCE)*, May 2002.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Aug. 2001.

[16] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 1998.

[17] R. van Renesse and K. Birman. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining.

[18] W. Wong. Web services directory still a dream. ZD Net, http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2873213,00.html.

[19] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.