Self-Managing Federated Services*

Francisco Matias Cuenca-Acuna†
matias@hal.famaf.unc.edu.ar
FaMAF, Universidad Nacional de Córdoba
Córdoba 5000, Argentina

Thu D. Nguyen
tdnguyen@cs.rutgers.edu
Department of Computer Science
Rutgers University, Piscataway, NJ 08854, USA

Abstract

We consider the problem of deploying and managing federated services that run on federated systems spanning multiple collaborative organizations. In particular, we present a peer-to-peer framework targeted to the construction of self-managing services that automatically adjust the number of service components and their placements in response to changes in the system or client loads. Our framework is completely decentralized, depending only on a modest amount of loosely synchronized global state. More specifically, our framework is comprised of a set of per-node monitoring agents and per-service-component management agents that periodically exchange information about the state of the system and of the service with each other using a gossiping protocol. Each management agent then periodically searches for configurations that are better than the current one according to an application model and explicit 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. We show that while agents act autonomously, the service rapidly reaches a stable and appropriate configuration in response to system dynamics.

1 Introduction

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 because these services must execute in heterogeneous and volatile 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[1] that are typically federated and replicated across multiple sites. Such multi-site services must currently be configured and maintained by hand, placing a considerable burden on system operators [27]. A concrete example is the European Data Grid, which has around 160 nodes (some of them clusters) federated across Europe[2]. This system relies on a 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 are configured and maintained manually.

In this paper, we describe a distributed resource management framework that can be used to build self-managing federated services. Our goal is to reduce the burden of deploying and managing a service to three tasks: (1) defining an application model for the framework to choose appropriate runtime configurations and the desired performability 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 the appropriate service components and replicas to meet the desired performability goals. 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 service components to maintain the performability goals.

Our management framework is completely decentralized: each component of a service is wrapped with a management agent that makes autonomous decisions about whether new components should be spawned and whether the current component should be stopped or migrated to a better node. Agents rely on a set of loosely synchronized data to make their decisions, including the load on each component and load information about potential host machines. Critically, this information only has to be loosely synchronized with weak consistency; as shall be seen, our prototype implementation can tolerate data staleness of minutes. This autonomy and dependence only on weakly

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

†Part of this work was done while at Rutgers University.

[1] http://uddi.org: UDDI is the standard resource discovery protocol defined for web services.

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

consistent data make our framework scalable and highly robust to the volatility inherent within federated systems.

Note that the association of a management agent with each service component is a fundamental design decision. This decision is one of the major driving force behind our approach of autonomous agent actions, which allows our framework to scale to services with a large number of components. An alternative is to build a separate monitoring and management framework. Integrating the management framework with the application has two significant advantages, however. First, in the presence of failures, there will always be a management agent wherever a component 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 application model, and the resources available. Second, our framework is unstructured and completely decentralized, making it highly robust even in very volatile environments.

As a proof of concept, we have implemented a prototype self-managing replicated UDDI service. We evaluate our prototype by studying its behavior in response to a simulated workload on two distinct testbeds. The first is a local cluster that allows us to study the behavior of our framework in a controlled environment. The second is the wide-area PlanetLab testbed[3]. 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 system dynamics.

2 Self Management Approach

We assume that each self-managing service is described by a model that maps possible configurations, i.e., number of components and their placements, to "fitness" values. We call this the application fitness function (or model). We further assume that the set of nodes that can host services managed by our framework forms a *hosting community* that can efficiently share information through mechanisms such as a distributed hash table [23] or a scalable publish/subscribe network [8]. This mechanism allows our management agents to monitor the service and underlying system, adapting the service configuration as needed to meet explicitly specified performability goals.

With the above assumptions, our self-management approach can be broadly described as follows:

- Each node runs a monitoring agent that periodically publishes (i.e., shares with the hosting community) the node's processing capacity, availability, and current load.
- Each management agent publishes the fact that a component is running, and where it is running, when the component is first started. It also publishes a "stop" notification when it stops the execution of the wrapped component.
- Periodically, each management agent independently searches for a better configuration using the application model and current information about the service and the hosting community. If it finds a configuration whose fitness exceeds the current one by a threshold value, it will try to deploy this new configuration after waiting for a random delay period to avoid collisions. After the delay period, the agent proceeds with the deployment only if the service is still in the original configuration. Otherwise, it aborts the deployment.

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 developed 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 search algorithm that our framework uses to find good configurations. Finally, we describe how our decentralized adaptation algorithm avoids collisions by choosing appropriate random delay times.

2.1 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.

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

Given the above metrics for node availability and capacity, and assuming that nodes' availability are not correlated, we then estimate the expected processing capacity of a set

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

of nodes, called a configuration, using the following equation:

$$C(c) = \sum_{i=1}^{|c|} \left(\begin{array}{c} |c| \\ i \end{array} \right) A(c)^i \left(1 - A(c)\right)^{|c|-i} i I(c) \qquad (1)$$

where c is a configuration, $|c|$ is the number of nodes in c, $A(c)$ is the average availability of the nodes in c, and $I(c)$ is the weighted average of the current processing capacity of all nodes in c. This equation is just a summation over the probability of i nodes being available times the idle capacity given by these i nodes. $I(c)$ is computed by weighing the idle bogomips of a node by its availability; this weighting is introduced to prevent high performance but low availability node from unduly increasing the average processing capacity of a configuration. Observe that, in general, the use of average availability and capacity makes this equation an approximation. We have found that a similar approximation is generally conservative and works well in driving replication in highly heterogeneous P2P environments [6].

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 conservative in that our framework will spawn additional components as needed to satisfy the offered load. In general, we prefer to err conservatively because it is better to exceed the availability target when resources are available than to leave resources idle and missing the availability target.

Given Equation 1, we then design a model that is suitable to a class of applications where:

- The application is comprised of a set of replicated homogeneous components.
- Increasing the number of replicas increases throughput but can also increase the latency of a subset of requests. Specific to the UDDI service studied below, increasing the number of replicas will increase throughput because the workload is typically dominated by reads, which only requires contacting one replica. The response time for writes will degrade, however, because writes must be applied to all 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.

The above properties lead us to derive the following guiding constraints for our fitness model. If we define $C(c)$ to be the expected capacity of configuration c, l the current client load, a the target service availability level, $f(c, l, a)$ the fitness of c given l and a, and $|c|$ the number of nodes in c, then for two different configurations c_1 and c_2:

1. if $[(C(c_1) \geq la) \wedge (C(c_2) < la)]$, then $f(c_1, l, a) > f(c_2, l, a)$; i.e., a configuration exceeding the performability target, defined by la, is better than one that cannot meet the target;
2. if $[(C(c_1), C(c_2) < la) \wedge (C(c_1) > C(c_2))]$ or $[(C(c_1), C(c_2) < la) \wedge (C(c_1) = C(c_2)) \wedge (|c_1| < |c_2|)]$, then $f(c_1, l, a) > f(c_2, l, a)$; i.e., if neither configuration can meet the target, choose the one with greater fitness; if the two have equal fitness, choose the one with less nodes to minimize communication overheads;
3. if $[(C(c_1), C(c_2) \geq la) \wedge (|c_1| < |c_2|)]$ or $[(C(c_1), C(c_2) \geq la) \wedge (|c_1| = |c_2|) \wedge (C(c_1) > C(c_2))]$, then $f(c_1, l, a) > f(c_2, l, a)$; i.e., if both configurations exceed the target, choose the one with less nodes; if the two have the same number of nodes, choose the one with greater capacity.

One subtlety in the above constraints 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, the excess capacity of that node is no longer useful for tolerating failures within the configuration. Thus, if the capped capacity of a configuration is lower than la, then it cannot meet the expected load at the target availability.

Finally, to limit the amount of global resources acquired by a service, we put a cap l_{max} on the maximum load that it is expected to handle. Figure 1 shows one possible fitness function that fits our criteria. 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.

2.2 Finding the Right Configuration

Given a fitness function, we then use a genetic algorithm to find "good" configurations [11]. In particular, we use bit

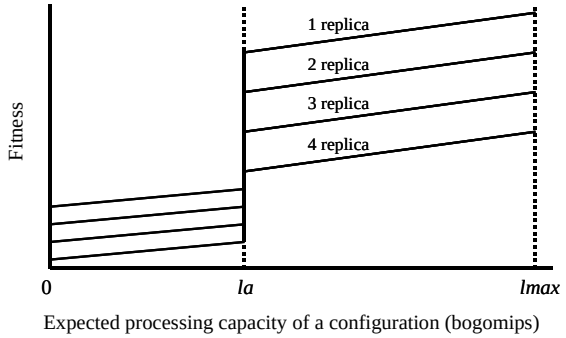


Figure 1. Fitness function used for our prototype UDDI service. l is the current offered load, a the target availability, and l_{max} the maximum expected offered load. Each curve gives the fitness for a different number of replicas.

vectors to represent service configurations, where each bit indicates the placement of a replica at a particular node. We use the standard crossover and mutation operations on these bit vectors to generate potentially good configurations. Periodically, each management agent searches for a configuration that is better than the current one according to its view of the community. Currently, each search starts from scratch and runs for a fixed number of generations. As the fitness model becomes more complex, there are many ways of optimizing this search, ranging from starting from previous good configurations to parallel searches. Our prototype implementation uses the JGAP open source genetic algorithm package[4].

2.3 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 probabilistic serialization approach reminiscent of Ethernet-style exponential back off. 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.

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 (probabilistic) bound on information propagation time, that is, the time from when an agent makes a configuration change and publishes it until when its peers will have seen the change (with high probability), and t_0 be the minimum chosen delay time. Then, assuming that the delay times are discrete, we can compute the probability of concurrent action as follows:

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

where *AllOutcomes* is the total number of possible outcomes (T^N) 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:

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

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 and vary T appropriately depending on the number of active management agents. 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 expected delay before a redeployment is enacted. In essence, this bounds the responsiveness of our system to system dynamics. Figure 2 plots the results vs. the number of management agents when v is equal to 8 seconds[5] and the probability of collision was set to 0.1. Observe that delays are just a little over 1 minute even for large numbers of replicas.

Finally, we also need to keep the redeployment time short to avoid collisions. This is because an agent does *not* publish its intention to deploy a new configuration: publishing intentions is similar to acquiring a lock and has the important disadvantage of requiring other agents to monitor the agent enacting the change so that if this latter agent fails, the system does not deadlock. Rather, changes are

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

[5] $v = 8$ is experimentally derived from our current implementation, where a hosting community uses a gossiping-based publish/subscribe system to share information. In this system, v is a logarithmic function of the size of the hosting community and the gossiping interval, evaluating to approximately 8 seconds for a community of 200 nodes.

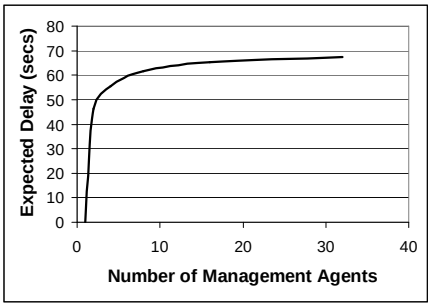


Figure 2. Expected delay vs. the number of management agents in a hosting community with 200 nodes. Expected information propagation time is 8secs and the target probability of collision is 0.1.

only published after they have been completed. Thus, the longer a node takes to instantiate a new configuration, the more likely that other nodes will start to concurrently deploy different configurations. We address this problem by introducing a deployment planning component. Specifically, our deployment planner will select a single step to enact per redeployment 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).

3 Architecture

We now describe a Java-based prototype implementation of our framework. In particular, we briefly describe PlanetP [7, 8], a P2P publish/subscribe infrastructure, and the self-management framework that we have implemented on top of PlanetP. Figure 3 shows an overview of this prototype in the context of the UDDI application: each node of a hosting community runs an instance of PlanetP and a monitoring agent; each service replica is a multi-tier application comprised of a jUDDI front-end and a HSQLDB/R database back-end[6]; each replica is managed by an instance of our management agent.

[6]jUDDI can be found at http://www.juddi.org/ and HSQLDB/R at http://www.javagroups.com/javagroupsnew/docs/hsqldbr.html.

3.1 PlanetP

Members of a PlanetP community share information by publishing "documents" to PlanetP. Each PlanetP node indexes the documents published at that node and maintains a detailed inverted index *local* to the node to support content searches. We call this the *local index*.

In addition, PlanetP implements two major components to enable community-wide sharing: (1) a gossiping layer that enables the replication of shared data structures across groups of nodes, and (2) a content search, ranking, and retrieval service. The latter requires two data structures to be replicated on every node: a membership directory and a very compact 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-node mappings, where the mapping $t \rightarrow n$ is in the index if and only if node n has published one or more documents containing term t. All members of a PlanetP community 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.

Previous work has shown that PlanetP can easily scale to community sizes of several thousands [8], which is more than sufficient for our purposes here. Critically, PlanetP dynamically throttles the gossiping rate so that the bandwidth used in the absence of changes quickly becomes negligible.

3.2 Runtime Environment

Similar to Tomcat[7], we package applications to be managed by our framework inside jar files that include instructions on how they should be installed and executed. Then, to run an instance of an application at a node, we just need to upload the jar file to that node. At the receiving node, a loader service verifies that all dynamic linkage dependencies can be fulfilled, downloading additional files as needed, and starts an instance of the application within an appropriate runtime environment. Optionally, a jar file can include an application fitness model.

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.

[7]http://jakarta.apache.org/tomcat/

[8]http://java.sun.com/webservices/jwsdp/index.jsp

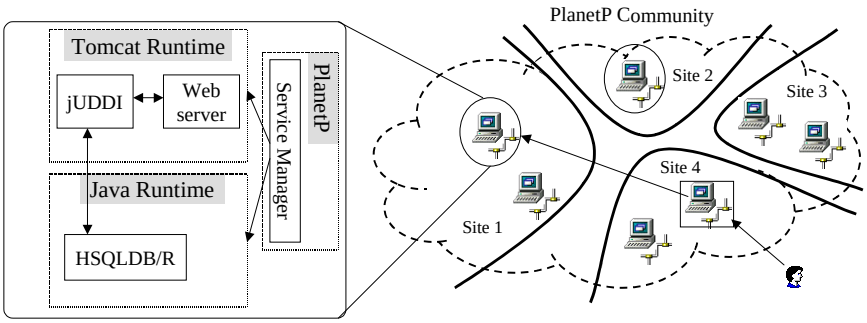


Figure 3. A hosting community spanning four organizations. Two replicas of the UDDI service are running on the circled machines. Clients access the service through proxies, one of which is shown running at Site 4, that use PlanetP to locate running instances of the service.

This runtime shares the same JVM with PlanetP so it allows services to use all the functionality provided by PlanetP.

One important extension to PlanetP developed in this work is support for services, which are Java objects that can be accessed over the network through RMI. Services advertise themselves by publishing *active documents* containing the appropriate client-side RMI stubs to PlanetP. Clients can then contact a service by retrieving the service's active document and dynamically loading the client-side stubs contained in the document. Effectively, PlanetP and active documents work as a completely decentralized RMI registry.

Node Monitoring. Each node in a hosting community runs a Node Monitoring service to provide load and idleness information. This information is published as properties of each node (peer) and so 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 CPU 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 it represents a change exceeding 20%.

Application Deployment and Management. A management agent is spawned whenever an instance of a service with a fitness model is started at a node. As already described, this agent is charged with the management of the service. In addition to pursuing performability goals, however, the management agent also monitors the health of the local replica and enforces a fail-stop failure model. For example, in the case of the UDDI service, if the local database stops responding then the agent will stop the entire service and unload all the dependencies so that the node can return to a clean state.

3.3 The UDDI Service and Clients

Our prototype UDDI service is built from two open source components: jUDDI and HSQLDB/R. 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. Although the UDDI v3 specification includes a replication protocol, this feature has not been implemented in jUDDI. We instead replicate our service by using HSQLDB/R[9], an experimental replicated version of the HSQLDB database. Our development of this service only consisted of wrapping the database into a PlanetP-based self-managed service, which required 270 lines of commented code given our framework. The fitness model for this service is as described in Section 2.1.

To evaluate our framework, we have also written a client application to load the UDDI service. This client application is itself a self-managing replicated application with a linearly increasing fitness function vs. the number of replicas up to a dynamically set maximum. Each replica generates a stream of requests according a Poisson process. These requests are load balanced across all the server replicas. In essence, this application really implements the proxies mentioned in Figure 3.

4 Evaluation

We now turn to evaluating the efficacy of our self-management framework. In particular, we study experimentally the responsiveness and stability of our framework in the presence of changing client load, interfering external load, and crashes of server nodes. We evaluate our framework using two distinct testbeds. The first is a LAN-connected cluster environment that allows us to easily un-

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

derstand the behavior of our framework in a controlled environment. The second is the PlanetLab testbed, which consists of nodes widely distributed across the WAN.

4.1 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 Pentium 4 machine is rated at 3971 bogomips. Our experiments do not run sufficiently long for meaningful measurements of node availability. Thus, we arbitrarily set the availability of all nodes to 90% when computing configuration fitness.

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%.

The fitness model for the UDDI service is constrained with a maximum desired performance of 60000 bogomips and a maximum number of 4 to 20 replicas depending on the experiment. (In practice, services always have a maximum number of replicas because they cannot scale indefinitely.) When running with greater than 4 replicas, we had to disable writes as the experimental database replication did not scale well, particularly in loaded environments where message latency may vary widely.

The base PlanetP gossiping interval is set to 1 second, which gives expected information propagation times of several seconds for all of our experiments here.

To emulate a realistic service, we populate our UDDI registry 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 issue random *findBusiness* queries against this set of registered providers. Even though this is a read-only workload, when writes are enabled in the service, replicas of the database sent many group-maintenance messages to each other.

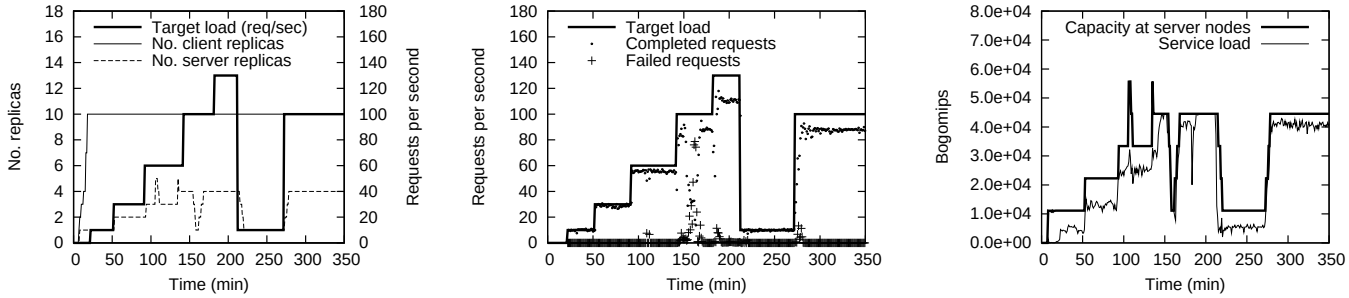
During each experiment, nodes individually log information about themselves and replicas running on them. Each node periodically contacts a time server and logs timing deviations together with the round trip latency. At the end of the experiment, we collect the individual logs and merge them into a single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

4.2 Behavior in a Quiet Environment

We begin by studying the adaptivity and stability of our framework when the service is running in isolation 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).

Figures 4 and 5 show the results from a representative run of the above experiment. Based on these results, 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 times the information propagation time between adjustments (Section 2.3).



(a) (b) (c)

Figure 4. Quiet cluster: (a) Number of service replicas (left y-axis), number of client replicas (left y-axis), and client offered load (right y-axis) plotted against experiment time. (b) Target load, completed requests, and failed requests plotted against experiment time. (At high loads, the clients were unable to reach the target load because of inaccuracies in the Java JVM timer: threads were not woken up quickly enough to generate the desired request rate.) (c) Total capacity allocated to the service and capacity actually used by the service plotted against experiment time.

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 4(c) 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 failed due to a buffer exhaustion error inside the group communication mechanism of HSQLDB/R. The management agent wrapped around each failed replica detected the failure and shutdown the replica. Then, the management agent on the remaining replica simply pushed 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.

Considering the impact of service adaptation on 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. 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 around time 200.

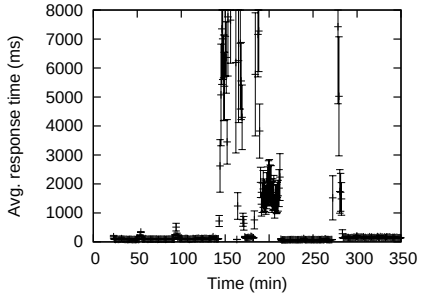


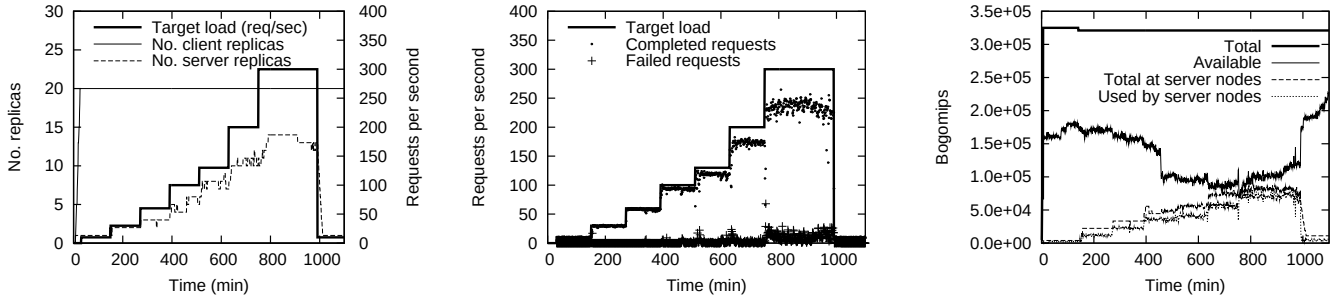
Figure 5. Average response time for completed requests with 95% confidence bars.

4.3 Scalability and Noise Resiliency

Figure 6 shows the results when we run a similar experiment on a loaded cluster testbed. In this experiment, we raise the maximum number of service and client replicas to 20 and the maximum load to 300rps. Also, note that in this experiment, the average global load introduced by other users on the clusters is 50%.

Observe that the management agents are still able to quickly track changes in the request rate. In contrast with the unloaded environment, the number of replicas is less smooth because the agents have to migrate instances in order to accommodate changes in nodes' instantaneous processing capabilities. This is particularly visible between times 600 and 800, during which the cluster is heavily loaded and there are visible spikes in the available processing capacity.

In this experiment, the efective throughput matches the



(a) (b) (c)

Figure 6. Loaded cluster: (a) Number of service replicas (left y-axis), number of client replicas (left y-axis), and client offered load (right y-axis) plotted against experiment time. (b) Target load, completed requests, and failed requests plotted against experiment time. (c) Total system capacity, idle capacity, capacity allocated to the service, and capacity actually used by the service plotted against experiment time.

request rate until the amount of bogomips starts to get closer to the maximum amount allowed for the service (between time 600 and 1000). For this same reason, the number of service replicas used throughout the experiment never exceeds 14 even though the maximum permitted is 20.

Finally, in Figure 7 we repeat the experiment using the PlanetLab testbed. 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 worse, 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.

Figure 7(c) shows that while the total amount of bogomips aggregated across 100 PlanetLab machines is similar to the amount present in the cluster testbed, the cpu availability is much lower (on average 20%). As shown in Figure 7(a) the reduced amount of CPU cycles and their volatility affects performance in two ways: (i) the number of replicas is less stable, and (ii) the agent reaction time is about 8 times worse than in the cluster environment. Note that in this testbed the agents use the maximum number of replicas allowed (i.e. 20) yet cannot harness the maximum CPU usage allowed for the service.

In summary, we find these results very encouraging. While there is more instability compared to results from the cluster testbed, PlanetLab currently presents a very challenging environment because of its extreme volatility and high load. Thus, the above results give us confidence that our framework would work well in realistic environments.

5 Related Work

Numerous efforts have explored the construction of infrastructures that would ease the development, deployment, and management of wide-area applications, e.g., [2, 9, 17, 18, 22, 24, 25]. To date, however, researchers have typically focused on lower-level building blocks such as group communication [17, 19, 22], distributed data structures [23, 28], remote execution [9, 24], autonomic deployment [2, 22], and service monitoring [9, 25]. Service management and deployment has either relied on strongly-consistent distributed algorithms such as group communication for coordination [22] or centralization [9, 24].

In contrast, the focus of our work is to raise the abstraction level for service management in federated systems. In particular, our framework uses an application model, which could be derived from a service level agreement (SLA), to automatically adapt the number and placement of service components to changes in the client load or in the underlying platform. Further, our framework is fully decentralized, relying on randomized algorithms such as gossiping and probabilistic serialization to tolerate even extreme volatility in the underlying system. We chose this probabilistic approach because strongly consistent protocols like two-phase commit, leader election, and virtual synchrony have been found to scale poorly in wide-area environments [12, 13, 26]. The trade-off, of course, is that our framework exports a weak consistency model, where different management agents may collide with each other in adapting the service configuration. This requires that applications behave correctly when such collisions occur.

Resource allocation based on SLAs and quality of service (QoS) metrics have been studied in the context of clus-

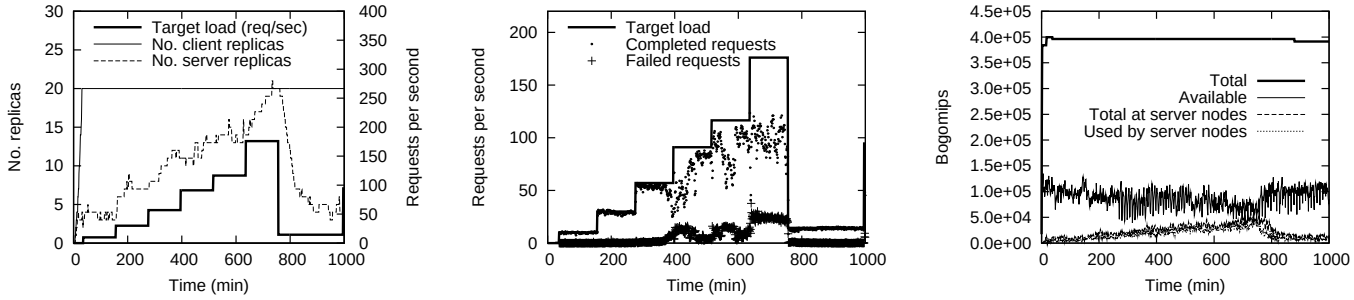


Figure 7. PlanetLab: (a) Number of service replicas (left y-axis), number of client replicas (left y-axis), and client offered load (right y-axis) plotted against experiment time. (b) Target load, completed requests, and failed requests plotted against experiment time. (c) Total system capacity, idle capacity, capacity allocated to the service, and capacity actually used by the service plotted against experiment time.

ters [3, 5, 21]. These studies were not concerned with robustness issues in wide-area systems, however, and so service management was centralized in contrast to our decentralized approach. More recently, several efforts have studied resource sharing in federated system [4, 10]; these efforts are complementary to ours in that they allow management services like ours to negotiate for resources in federated systems that span multiple organizations.

Previous work on distributed agent-based systems are also related to our decentralized design [1, 14, 15]. For example, Archon [14] has explored the use of agents that rely only on partial information and cooperate to reach global goals. A key difference from our work is that agents in these systems build a distributed database schema using information from an agent registry. Agents then use this schema to direct inquiries concerning the state of the community. The potential high volatility of the environments targeted by our work makes this approach less attractive.

Finally, some researchers have combined work on peer-to-peer and agent-based systems to build more robust and unstructured frameworks [16, 20]. However, these efforts did not study the coordination problem that we address here.

6 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 federated 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 components to meet desired performance and availability goals. 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 service components to maintain the target performance and availability levels.

The decentralized nature of our framework makes it highly robust to the volatility inherent to federated systems. Our experimental evaluation using two distinct testbeds, one being the 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 models. We will also need to explore models that include other performance and availability metrics than just expected CPU processing capacity.

References

[1] H. Afsarmanesh, F. Tuijnman, M. Wiedijk, and L. O. Hertzberger. The Implementation Architecture of PEER Federated Object Management System. Technical report,

Department of Computer Systems, University of Amsterdam, 1994.

[2] 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.

[3] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oc´eano-SLA Based Management of a Computing Utility. In *IEEE/IFIP Integrated Network Management Proceedings*, May 2001.

[4] M. Balazinska, H. Balakrishnan, and M. Stonebreaker. Contract-Based Load Management in Federated Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The International Journal of Supercomputer Applications*, 15(3), 2001.

[10] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

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

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1996.

[13] I. Gupta, K. P. Birman, and R. van Renesse. Fighting Fire with Fire: Using Randomized Gossip to Combat Stochastic Scalability Limits. *Special Issue of the Journal on Quality and Reliability Engineering International: Secure, Reliable Computer and Network Systems*, 29(8):165–184, May 2002.

[14] N. R. Jennings. The ARCHON System and its Applications. In *Second International Working Conference on Cooperating Knowledge Based Systems (CKBS-94)*, 1994.

[15] N. R. Jennings. An Agent-Based Approach for Building Complex Software Systems. *Communications of the ACM*, 44(4), 2001.

[16] A. Montresor, H. Meling, and O. Babaoglu. Messor: Load-Balancing through a Swarm of Autonomous Agents. In *Proceedings of the 1st Workshop on Agent and Peer-to-Peer Systems*, July 2002.

[17] 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.

[18] 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.

[19] 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.

[20] K. Schelfthout and T. Holvoet. A pheromone-based coordination mechanism applied in P2P. In *Proceedings of the 2nd International Workshop on Agents and Peer-to-Peer Computing*, July 2003.

[21] 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.

[22] Y. shun Wang and J. Touch. Application Deployment in Virtual Networks Using the X-Bone. In *Active Networks Conference and Exposition (DANCE)*, May 2002.

[23] 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.

[24] 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.

[25] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[26] W. Vogels, R. van Renesse, and K. Birman. The Power of Epidemics: Robust Communication for Large-Scale Distributed Systems. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, Oct. 2002.

[27] W. Wong. Web services directory still a dream. ZD Net, http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2873213,00.html, 2002.

[28] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), Jan. 2003.