Resource-Aware Migratory Services in Wide-Area Shared Computing Environments

Anand Tripathi, Vinit Padhye, Devdatta Kulkarni  
Department of Computer Science  
University of Minnesota  
Minneapolis 55455 Minnesota USA  
Email: (tripathi,padhye,dkulk)@cs.umn.edu

Abstract—In this paper we present the design and evaluation of a system for deploying highly available and migratable services in shared infrastructures, such as the PlanetLab, where the available resource capacities at a node can fluctuate significantly. A migratable service can monitor its operating conditions and autonomously relocate itself to another node when the available resource capacities at the current node fall below certain acceptable limits. We utilize the autonomous mobile agent paradigm for building such migratable services. Such agents can monitor their operating conditions and follow various migration policies. We investigate here the mechanisms for service relocation, and client-side protocols to access migratory services. The “blackout periods”, i.e., the time during which the clients are unable to access a migrating service, need to be minimized and kept within some tolerable limits for services required to be highly available. We first present the design of a migratable service implemented using a mobile agent, and evaluate its performance in terms of the blackout periods and the service agent’s abilities to autonomously migrate in the network. We replicate service agents to reduce the blackout periods, and develop the coordination protocols for autonomous agent migration in a group of service agents. We also present here our work for monitoring PlanetLab nodes for their available resource capacities in order to assist a migratory service in selecting a target node for relocation.

I. INTRODUCTION

The availability of shared computing infrastructures such as the PlanetLab [1] and Grid facilities [4] over the Internet provides an opportunity for building services which can exploit geographically distributed computing resources over the network for building highly available and resilient applications. Such services can be designed and deployed over the network with resource-aware and network-centric considerations. The focus of our work is on building highly available and resilient services in shared infrastructures such as the PlanetLab where the available resource capacities are allocated to different applications on fair-share basis, and there is no guarantee of resource availability due to fluctuating load conditions. The results presented in [10] show that the available resource capacities at a node may change within an hour. The nodes selected to deploy an application with resource availability considerations are most likely to become less than ideal choices due to changing load conditions over time. Using simulation based experiments, the work presented in [10] argues in favor of supporting dynamic relocation of applications based on resource availability. However, no mechanisms for dynamic relocation of services were developed or presented there.

We present here the design and evaluation of a system for building resource-aware migratory services which can dynamically relocate themselves to nodes which have the required resource capacities. Such a service can monitor its operating conditions and autonomously relocate itself to another node when the available resource capacities at the current node fall below the acceptable limits. In this paper we demonstrate the feasibility and utility of building such an infrastructure in the PlanetLab environment. The ability to dynamically relocate a service can also be utilized in other environments for different purposes, such as the placement of services closer to their clients. It can also be used for load balancing or maintenance operations that require shutting down a node hosting some services.

Several technical issues need to be addressed to build services which can autonomically relocate themselves. Our work was driven by the investigation of these issues. The first set of issues is related to mechanisms for supporting service migration. Migration of a service requires transfer of its current execution state in the volatile storage as well as its archival state in the secondary storage, such as a back-end database. We also need mechanisms for properly handling any currently ongoing client-sessions at the time of service migration. Mechanisms are needed for clients to determine the current location of a migratory service, and such mechanisms themselves must be highly available. The service migration mechanisms should be robust and should not introduce new causes of failures, otherwise making a migratory service less robust and less available in contrast to its non-migratory version. The blackout periods experienced by the clients during service migration should be either eliminated or made negligible so that they are tolerable within certain limits. The requirement of high availability may necessitate the use of a group of service replicas to implement a service. While a service replica is migrating, other replicas could be used to service client requests. A new replica for the service could be dynamically regenerated when the number of replicas in the group falls below some

This work was supported by National Science Foundation grants 0834357 and 0708604
threshold due to node crashes or other failures.

The second set of issues is related to supporting integration of migration policies in building autonomic migratory services. Policies are needed to be defined and integrated with a service for determining when to relocate the service to another node, and how to select a node as the relocation target. For building a resource-aware migratory service, a service needs mechanisms to monitor its workload, the resource capacities available at its current node, and a model to determine the migration trigger conditions. Selection of a new target node for relocation requires a global monitoring infrastructure that can provide to a service a list of nodes that best match the resource capacity requirements of the service.

In this paper we present the results of our design and evaluation of the mechanisms that we have developed for building autonomic migratory services in the PlanetLab environment. We use the Ajanta mobile agent programming system [19] for building migratory services. Ajanta provides the facilities for an agent to autonomously migrate to another host in the network, where an agent server is used for hosting such mobile agents. Ajanta provides the facilities for transporting the execution state of the agent to the remote server, and resuming the execution at a specified execution point. An agent can communicate with other applications and agents in the network using Java RMI or TCP connections.

A service is implemented by one or more mobile agents; these are called service agents. For a service implemented by a group of such agents, we use the primary-backup model [2] where one service agent functions as the primary and others act as backup servers. The main motivation is to reduce the blackout periods during migration, and increase the resiliency of a service in case of node crashes. Each service agent in a group is capable of migrating autonomously. We require a scheme that ensures that no new vulnerabilities are introduced if multiple service agents start migrating simultaneously.

The main contributions of this work are in the development and evaluation of mechanisms for building resource-aware autonomic migratory services. For increased resiliency and availability, we develop here the mechanisms for implementing a service using a group of replicated migratory service agents. To assist a migratory service in determining the potential target nodes for relocation, we have designed a system for monitoring PlanetLab nodes for their available resource capacities. This is used to provide a migratory service a list of nodes that meet the specified resource requirements. We evaluate the goodness of the recommendations provided by this monitoring service. We conducted several experiments with the deployment of migratory services over the PlanetLab. We present here the data about autonomic migrations of several service agents in our experiments. In another experiment we measured the blackout periods experienced by clients when only a single agent is deployed to implement a service. These experiments show wide variations in the blackout periods. We demonstrate the benefits of implementing a migratory service as a group of replicated service agents in significantly reducing and limiting the blackout periods.

In the next section we present an overview of the system architecture that we have developed for deploying autonomous migratory services over the PlanetLab environment. Section III describes the core elements of the architecture of a migratory service implemented using an Ajanta mobile agent. The focus of Section IV is on implementing a service using a group of replicated service agents. Section V describes the node monitoring system which we implemented over the PlanetLab. We evaluate here the goodness of the available node sets provided by this system for various requirements of available CPU capacity. We are interested in determining how long a node in this set continues to meet the specified resource requirements. Section VI describes the experiments that we performed over the PlanetLab in deploying several migratory service agents with different kinds of migration policies and CPU capacity requirements. In Section VII we discuss the results of our experiments and the future directions. Section VIII discusses the previous research efforts in building migratory services and service relocation. Finally, in the last section we present the conclusions of our work and discuss the directions of our future work.

II. FRAMEWORK FOR MIGRATORY SERVICES

In this section we present the system architecture for building migratory services in wide-area network environments. Such a system architecture needs the following mechanisms:

- Mechanisms to transfer a service agent from one node to another node.
- A service agent needs mechanisms to monitor its operating environment and to execute autonomic actions such as shedding client load, redirecting the clients to other service replicas, and migrating to a higher capacity node.
- In order to find and select a suitable node for hosting a service, the system needs to monitor the PlanetLab nodes for their available resource capacities.
- Mechanisms are needed for clients to find the current location of a migratory service.

In Figure 1 we present the system architecture that we have designed to build and deploy migratory services in shared wide-area network environments, such as the PlanetLab. The salient features of this architecture include: (a) an agent based programming framework implementing migratory services and supporting service replication; (b) an infrastructure for monitoring the PlanetLab nodes for available resource capacities; (c) a service location registry,
implemented as a distributed hash table (DHT), for maintaining up-to-date location information about migratory services. This system architecture is characterized by the interaction protocols between the following components: (i) a service agent and the DHT, (ii) clients and a service agent, (iii) coordination among service agents belonging to a service.

We use mobile agents for building migratory services. An agent is an active object encapsulating other application-specific components, such as those implementing a service’s functionality. A mobile agent is capable of autonomously migrating in the network. This agent model is supported through the Ajanta mobile agent programming framework [19]. Agents are named and accessed using the location-independent naming scheme based on Uniform Resource Names (URN) [17]. Ajanta provides the facilities for an agent to autonomously migrate to another host in the network, where an Ajanta agent server is used for hosting such mobile agents.

Each service in our system is named and accessed using a unique service-id (SID). A client accesses a service by first querying the DHT, which maintains the network location information of all service agents for a service-id. Additionally, the DHT record for a service-id contains information about the service access protocols to be used by the clients, and the roles of the various service agents as primary or backup. The client invokes a service by directly communicating with one or more service agents. The functionalities of this DHT-based service registry are implemented using FreePastry [14]. A service agent interacts with the DHT for registering the service information, which includes the following: the agent’s URN, its role (primary or backup), the agent’s status (active, or migrating), and the IP address and the port on which the agent can be contacted for accessing the service. A service agent performs such a registration whenever it migrates to a new host.

When a client queries the DHT for a service-id, it gets the replica records for all of the service agents. Each record contains the network location information, and the current status and role of the agent. The status indicates whether the agent is active, migrating, or suspected to have failed. In our current prototype the clients contact the primary service agent. In response to the client request, the agent sends the following responses to the client: success, or redirection. The service agent handles a client request only if it is active and not overloaded. Otherwise, the service agent sends a redirect response to a client to indicate another service replica agent that should be used as the service access point. The redirect message contains the URN, the IP address, and the port number for that service agent. Such redirection responses are sent to the clients for load redistribution or when an agent is about to begin migration from its current node. If a client fails to communicate with the primary service agent, it once again queries the DHT for the latest configuration information for that service-id, and tries to contact the primary agent in the configuration information received from the DHT.

We use a special agent, called the Deployment Agent (DA) for creating and launching the various replica agents of a service. The interactions between a service agent and the DA consists of two kinds of messages: periodic status reports and migration requests. A report contains two types of status information: service load status, and node status. The service load status information includes: the number of clients serviced by the agent, the average request processing time, the number of redirected clients, and the service agent’s current usage. The node status information includes: the average load on the node, the average free capacity on the node, and the number of slices running on the node.

A migration request is made by the agent based on its local policies. For example, a service agent may want to migrate to another node if the resources available at its current node are no longer adequate for servicing the current load presented by the client requests. For this purpose, each service agent monitors its local execution environment for the current workload and the available resource capacities at the current node. Migration can also be triggered by other conditions specified in the service deployment policies.

For assisting a service agent in selecting a potential target node for relocation, we have developed a service which monitors the PlanetLab nodes for their available resource capacities. This service periodically monitors a large set of nodes and maintains the following node-level statistics: average CPU utilization and free capacity based on 10-second observations of PlanetLab nodes’ slices’ data [11] over a sliding window of 5 minutes, the standard deviation of these average utilization values, and exponentially weighted utilization. A service agent seeking a target node queries this monitoring service by specifying its resource capacity requirement. Using a selector function, the monitoring
service determines a set of nodes satisfying the resource requirements and ranks them. We have used and tested different selector functions for this purpose. These are detailed further in Section V.

In our current experimental testbed, a service agent sends a migration request to the DA when its migration policy indicates a need to migrate to another node. As part of the request, it indicates the required resource capacities. The DA queries the monitoring service mentioned above to obtain a set of nodes that satisfy the service agent’s requirements. It then selects a subset of these nodes, based on the service deployment policies specified at the time of service deployment. The DA provides to the service agent this set of potential target nodes where the agent can migrate to.

III. Mobile Service Agent Architecture

The architecture of the mobile service agent is shown in Figure 2. It contains threaded components corresponding to the following tasks: (a) servicing client requests, (b) service usage monitoring, (c) node status monitoring, (d) interacting with the deployment agent (DA), and (e) executing the agent migration protocol.

![Figure 2. Mobile Service Agent Architecture](image)

The service request scheduler component handles client requests by creating a new thread for handling each request. Admission control policies determine whether the service agent should accept new client requests.

The service monitoring component performs monitoring of the agent’s own resource usage on the current host. This includes aggregating the following statistics as part of the service load status information: number of client requests successfully handled, number of requests redirected, the average number of requests over a time window of 5 minutes, average request processing time during the time window, the average resource usage and its standard deviation over a time window of 5 minutes, and exponentially weighted average usage.

The node monitoring component performs periodic monitoring of resource utilization at the current node. In our work the monitoring interval is set to 15 seconds. This includes aggregating the following statistics as part of the node status information: number of active slices running on the node, node’s average usage and standard deviation over a time window, its average free capacity over a time window of 5 minutes, and exponentially weighted average utilization. The service load status and the node status information is sent periodically to the deployment agent (DA) through the status reporting interface. In our current experiments the periodic reporting interval is set to 1 minute.

The migration handler component executes the migration protocol for an agent. When the migration policy indicates a need to relocate to another node, the service agent requests the deployment agent for the potential set of nodes that satisfy the required resource capacities. If it gets non-empty set of potential target nodes, it proceeds with the execution of the migration protocol, which consists of the following steps. First, it selects a node from the target set of nodes, and checks its network accessibility. It then updates its state in the DHT to indicate that it is in the migrating state and it stops accepting new client requests. The service state in the secondary storage is then transferred to the destination node. If this transfer is successful, the agent then stops its components for service status monitoring, node monitoring, status reporting, and client request scheduling. At this point it executes Ajanta’s mobile agent transfer primitive to relocate the agent to the agent server at the destination node. In case the selected node is not available, the transfer protocol attempts to connect to the next node in the target list.

When an agent arrives at the destination agent server, the arrive method of the agent is executed first, as part of the Ajanta agent migration protocol. This method starts the execution of the components shown in Figure 2, and registers with the DHT its new network location, and changes its status from migrating to active.

IV. Replication of Mobile Service Agents

Service replication is desired for two reasons. One is to reduce the blackout periods experienced by the clients when a service agent is migrating. If a service is implemented by a single agent, the blackout periods can be significant depending on the size of service footprint in the secondary storage. With replication of service agents, client requests can be redirected to the other agents in the service group during the migration period. The second reason is to increase the survivability of a service as the availability of a PlanetLab node is not guaranteed; a node could be shutdown or rebooted at any time by the local site administrators.

We used the primary-backup model for managing the group of agents for a service. The group may consist of any number of agents, with one acting as the primary and
others as backup. All agents in a group are capable of making the migration decision autonomously, as described in the previous section. Each agent could use its own policies to trigger migration. It is possible for multiple agents to concurrently decide to migrate from their current locations to new locations. Our design for managing a service agent group is driven by the requirement that a pair of primary and backup agents should be operating in the system while some of the agents in the group are migrating. One way to realize this is to have at least three agents in a service group, and use a coordination protocol that enforces mutual exclusion for permitting an agent to migrate. It is also possible to implement more general policies, such as allowing any \( k \) out of \( n \) agents to migrate. In addition to coordination of agent migration, we also need protocols for detecting the failure of a service agent, and reassignment of the primary role in case the current primary fails. Finally, the replication management protocol is also required to implement propagation of updates from the primary to the secondary.

There are three aspects in managing a group of replicated service agents. These are related to the managing the primary-backup modes of operation, performing migration coordination among the replica agents, and detecting service agent failures using heart-beat messages. To perform these management functions the service agents in a group are configured into a dynamically ordered set, with the primary service agent being the first element in the set. The DHT maintains the information about the ordering of the service agents in a group. The clients use this information to determine the current primary agent for serving their requests. If the primary fails or begins the migration protocol, the next agent in the group takes over the primary function. The new primary communicates the new group configuration information to all the other agents, and each agent in turn updates its registration record in the DHT.

When a service replica agent wants to migrate to another node, it executes the migration protocol presented in Figure 3. The steps executed as part of this protocol are as follows. First, the migrating agent executes the coordination protocol with the other replica agents that are currently active. This protocol is executed only after obtaining a non-empty set of potential target nodes. Only if the coordination protocol indicates that the service agent can migrate, it executes the next steps in the agent migration protocol (lines 3-18). These steps include (a) selecting the target node for migration; (b) checking that the target node is reachable; and (c) performing the role handoff, which transfers the primary role to one of the secondary agents along with any update logs. A handoff request is accepted by secondary agents only if it is coming from the current primary. The rest of the steps of the protocol are similar to those in case of a single service agent’s migration, which were presented in the previous section. The function \textit{transferSecondaryStorageState} transfers the service’s state in the secondary storage to the target node. If this function returns successfully then the remaining migration steps (lines 12-14) are performed. Otherwise, the agent attempts to migrate to another node in the set of target nodes.

To detect the loss of an agent during migration, the currently active nodes use a timeout mechanism that takes into account the estimated migration time for the agent. If no heart-beat messages are received from that agent after this timeout period, the agent is suspected to be failed and it is removed from the set.

V. PLANETLAB MONITORING SERVICE

The goal of designing the PlanetLab monitoring service is to provide a service agent with a set of nodes that satisfy its resource requirements in terms of available CPU capacity, memory, and bandwidth. We are interested in the average values for these measures, and also in their variation over time. We observe that \textit{CoMon} [11], the node monitoring service provided by the PlanetLab, cannot be used for our purposes directly. This is because of the following reasons. For CPU usage, CoMon provides average values over system-defined monitoring intervals of 1 and 5 minutes. In our experiments, we need node-level resource utilization data that is collected at a high frequency (such as every 10-20 seconds) and aggregated to determine statistics over configurable observation intervals. This is important in order to obtain accurate measurements of a node’s behavior over such intervals. We are also interested in the standard deviation of the CPU usage. CoMon does not provide this measurement.

In the PlanetLab monitoring service that we have built,
we measure a node’s idle CPU capacity as the difference between the node’s intrinsic capacity and the total usage of all the slices running on that node. We calculate a node’s intrinsic capacity as the product of the node’s CPU power (in MHz) and the number of cores available on the node. For a given CPU capacity requirement the monitoring service utilizes a selector function to identify the set of nodes that have available resource capacity meeting the given requirement. These nodes define the eligibility set for the given requirement. We have experimented with the following two selector functions.

- **Selector 1**: A node is selected for inclusion in the eligibility set if the average value (c) of its idle CPU capacity and its standard deviation (σ) over a system-defined observation interval satisfy the following relationship with the given CPU capacity requirement R: (c - 2 * σ) ≥ R.

  A node is removed from the eligibility set when its average idle CPU capacity c drops below the specified requirement R (i.e. c < R).

- **Selector 2**: We use the same selection criterion as for Selector 1. We remove a node from the eligibility set only if c < R holds for k consecutive monitoring cycles. Thus, Selector 2 degenerates into Selector 1 when k = 1.

A node’s eligibility period is defined as the time between the event when it is added to the eligibility set to the next event of its removal from the set. During an observation period a node may enter and leave this set several times. In our experimental measurements we determined for each node the median value of the observed eligibility periods. The reason for using the median value instead of the mean was to reduce the effect of small number of extreme values on the performance measures.

Between Selector 1 and Selector 2 (with k>1), we expect the latter to give increased values for the eligibility periods for a node. This is because Selector 2 continues to retain a node in the eligibility set by tolerating its temporary failures in meeting the eligibility criterion. In this case we are interested in finding out the fraction of the total eligibility period during which the node satisfies the selection criteria. We refer to this fraction as the goodness factor of the eligibility period. The effective gain of Selector 2 over Selector 1 for a node is defined as the ratio of their mean eligibility periods multiplied by the goodness factor.

We present here evaluation data from an experiment of monitoring randomly picked 189 PlanetLab nodes for a duration of 27 hours 49 minutes. We used Selector 1 and two instances of Selector 2 with k equal to 3 and 5. Every 10 seconds resource utilization data for each node was collected from the slicesstat. In Table I we present the mean and the median values of the eligibility periods, the effective gain when using Selector 2, and the average size of the eligible node sets.

We make the following observation from this data. The average eligibility period increases with increasing value of k. This is seen for all values of CPU capacity requirements. In all of the observed cases we found that the value of effective gain is greater than or equal to one. The value for this metric shows an increase from k = 3 to k = 5. It tends to decrease with increasing CPU capacity requirement. These results indicate that if a service agent tolerates a temporary lapse (in our experiments, 30 to 50 seconds) in its host node’s ability in meeting the resource requirements, then it is possible for that node to continue meeting the agent’s requirements for a longer time.

The work in [10] presents data showing that a selection of nodes for application placement remains suitable for 30 to 60 minutes. That work analyzed the CoMon data for an extended period. In contrast our experiments indicate that the expected period for which a node continues to meet a given CPU capacity requirement strongly depends on the specified requirement threshold. For example, for 3.2 GHz CPU capacity requirement the median values for this range from 55 minutes to 3 hours and 17 minutes for three different selectors that we used in our experiments. In comparison, for 1.6 GHz requirements the values ranged from about 2 hours 23 minutes to 11 hours and 33 minutes.

In Figure 4 we present the cumulative distributions of node eligibility periods for Selector 1, and Selector 2 with k=5. We observe that eligibility period decreases with increasing values of CPU capacity requirements.

VI. EXPERIMENTS IN DEPLOYING MIGRATORY SERVICE AGENTS IN PLANETLAB

Towards building a programming framework for agent based migratory services, we developed a new abstract class called ServiceReplicaAgent, inheriting from the basic Agent class of Ajanta. This class implements the core set of components of a service agent, as shown in Section III and the replication management and coordination protocols described in Section IV. For the purpose of our experiments with migratory services, we implemented a simple HTTP service inheriting from the ServiceReplicaAgent class. This HTTP service was intended to be used as a content distribution service. In order to validate that no requests were lost and that the agent state was correctly preserved across migrations, with each request we returned an unique integer number, which was sequentially incremented on each service request.

We now present the results of our experiments in deploying migratory service agents in the PlanetLab environment. In the course of our experiments we created services with agents using different kinds of migration trigger policies. These include timer based and CPU capacity requirement based migration trigger policies. We performed three sets of experiments.
### Table I

<table>
<thead>
<tr>
<th>CPU Capacity Requirement (MHz)</th>
<th>400 MHz</th>
<th>800 MHz</th>
<th>1600 MHz</th>
<th>2400 MHz</th>
<th>3200 MHz</th>
<th>4000 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>k = 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg. Eligibility Period</td>
<td>6 hr 29 min</td>
<td>6 hr 1 min</td>
<td>4 hr 54 min</td>
<td>3 hr 46 min</td>
<td>3 hr 42 min</td>
<td>2 hr 17 min</td>
</tr>
<tr>
<td>Median Eligibility Period</td>
<td>4 hr 7 min</td>
<td>3 hr 41 min</td>
<td>2 hr 23 min</td>
<td>1 hr 53 min</td>
<td>55 min</td>
<td>8 min</td>
</tr>
<tr>
<td>Avg. Size of Eligible Set</td>
<td>136</td>
<td>118</td>
<td>101</td>
<td>78</td>
<td>56</td>
<td>40</td>
</tr>
<tr>
<td><strong>k = 3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg. Eligibility Period</td>
<td>16 hr 14 min</td>
<td>14 hr 15 min</td>
<td>10 hr 26 min</td>
<td>7 hr 32 min</td>
<td>7 hr 8 min</td>
<td>4 hr 43 min</td>
</tr>
<tr>
<td>Median Eligibility Period</td>
<td>13 hr 54 min</td>
<td>13 hr 52 min</td>
<td>11 hr 5 min</td>
<td>4 hr 5 min</td>
<td>3 hr 7 min</td>
<td>16 min</td>
</tr>
<tr>
<td>Effective gain (Median)</td>
<td>2.08</td>
<td>2.0</td>
<td>1.5</td>
<td>1.37</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Avg. Size of Eligible Set</td>
<td>138</td>
<td>120</td>
<td>103</td>
<td>79</td>
<td>57</td>
<td>41</td>
</tr>
<tr>
<td><strong>k = 5</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg. Eligibility Period</td>
<td>19 hr 44 min</td>
<td>17 hr 9 min</td>
<td>12 hr 39 min</td>
<td>8 hr 54 min</td>
<td>8 hr 18 min</td>
<td>5 hr 34 min</td>
</tr>
<tr>
<td>Median Eligibility Period</td>
<td>27 hr 49 min</td>
<td>21 hr 4 min</td>
<td>11 hr 33 min</td>
<td>4 hr 24 min</td>
<td>3 hr 17 min</td>
<td>18 min</td>
</tr>
<tr>
<td>Effective gain (Median)</td>
<td>3.00</td>
<td>2.5</td>
<td>1.5</td>
<td>1.25</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Avg. Size of Eligible Set</td>
<td>138</td>
<td>122</td>
<td>104</td>
<td>80</td>
<td>58</td>
<td>41</td>
</tr>
</tbody>
</table>

Figure 4. Comparing cumulative distributions of node eligibility periods

- In the first set of experiments our goal was to validate the autonomous migration capabilities of agents programmed with migration trigger policies based on CPU capacity requirements. We present here the number of migrations performed by a service agent and agent’s residency time at a node. These experiments were performed with non-replicated service agents.
- The goal of the second set of experiments was to determine the client blackout times. For this we deployed non-replicated service agents which were configured to transfer secondary storage state of 10MB as part of the migration protocol.
- The goal of the third experiment was to measure the utility of replicating service agents for reducing the client blackout times.

In the first set of experiments, we used three different service agent configurations with the following values for free CPU capacity requirements: 3.0 GHz, 3.5 GHz, 4.0 GHz. These agents were programmed with a local migration control policy of not relocating to the 5 most recently visited nodes. These experiments were conducted independently at different times over several days. The durations of individual experiments were also different. We ran these experiments on 40 PlanetLab nodes. There is a large variability among the PlanetLab nodes with respect to the number of cores. We selected only those nodes that had two cores. In Table II, Table III, and Table IV we present the results of these experiments.

In Table II we present the migration data for three agents. The agent with 3.0 GHz requirement made 24 successful migrations in a period of 44 hours (which was the duration of this experiment). The experiments for the agents with requirements of 3.5 GHz and 4.0 GHz were run for 2.5 to 3 hours. For the agent with 3.5 GHz requirement, the deployment agent returned non-empty set of eligible nodes for each migration request by this agent. However, the agent’s local migration control policy mentioned above resulted in 38 migration failures. For 4.0 GHz requirement, there was only
one successful migration, and 279 failed migration attempts. This is because in those cases the monitoring service was not able to find any node satisfying such a high capacity requirement.

In Table III we present the data showing how long an agent stayed at a particular node (agent’s residency time). We see a large variation in the residency times for all the agents. This is to be expected because of the dynamically changing available capacities of the PlanetLab nodes.

In Table IV we present the agent migration times. This time is measured by the DHT; it is the interval from the time an agent’s state is set to migrating to the time when it is set again to active. An agent is unable to handle any client requests during this time. This is the minimum blackout time that any client would experience. In our experiments the maximum value of migration time (across all the agents) was about 1 minute and 30 seconds.

In the second experiment we deployed three non-replicated service agents and configured them to transfer 10MB file to the migration target nodes. The goal of this experiment was to measure the blackout times observed by the clients. We used timer based migration policy which caused agents to migrate every 3 minutes. We used two clients for each service agent and measured the blackout times observed by these clients. In Table V we present the client blackout times. We observe a large variance in blackout times which was mainly because of large variations in delays incurred in transferring the secondary storage state (10 MB file) during migrations.

In the third experiment we deployed a service using a group of three replicated agents. Each agent was programmed with a timer-based migration policy to migrate every 3 minutes, and with a fixed delay of 3 minutes to emulate secondary storage state transfer. We used three clients and measured the blackout times observed by these clients. In Table VI we present the client blackout times observed in this case. We observe that blackout times are significantly reduced with the use of replicated service agents, and the maximum value was 14.88 seconds. This is to be expected as a backup agent takes over the primary role as soon as the primary agent decides to relocate.

### Table II

**SERVICE AGENTS**

<table>
<thead>
<tr>
<th>CPU Requirement</th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0 GHz</td>
<td>24</td>
<td>42</td>
<td>280</td>
</tr>
<tr>
<td>3.5 GHz</td>
<td>24</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4.0 GHz</td>
<td>56 min</td>
<td>42</td>
<td>92.29 sec</td>
</tr>
</tbody>
</table>

### Table III

**SERVICE AGENT RESIDENCY TIMES AT A NODE**

<table>
<thead>
<tr>
<th></th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>10 sec</td>
<td>10 sec</td>
<td>2 hr 27 min</td>
</tr>
<tr>
<td>Maximum</td>
<td>13 hr</td>
<td>1 hr 42 min</td>
<td>2 hr 27 min</td>
</tr>
<tr>
<td>Average</td>
<td>1 hr 51 min</td>
<td>34 min</td>
<td>2 hr 27 min</td>
</tr>
<tr>
<td>Median</td>
<td>5 min</td>
<td>1 min</td>
<td>2 hr 27 min</td>
</tr>
</tbody>
</table>

### Table IV

**SERVICE AGENT MIGRATION TIMES**

<table>
<thead>
<tr>
<th></th>
<th>Agent 1</th>
<th>Agent 2</th>
<th>Agent 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>1.68 sec</td>
<td>6.33 sec</td>
<td>2.93 sec</td>
</tr>
<tr>
<td>Maximum</td>
<td>102.29 sec</td>
<td>89.87 sec</td>
<td>2.93 sec</td>
</tr>
<tr>
<td>Average</td>
<td>21.35 sec</td>
<td>39.7 sec</td>
<td>2.93 sec</td>
</tr>
<tr>
<td>Median</td>
<td>3.83 sec</td>
<td>89.87 sec</td>
<td>2.93 sec</td>
</tr>
</tbody>
</table>

### Table V

**CLIENT BLACKOUT TIMES IN CASE OF NON REPLICATED AGENTS**

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 sec</td>
<td>56 min</td>
<td>1 min</td>
<td>15 min</td>
</tr>
</tbody>
</table>

### Table VI

**CLIENT BLACKOUT TIMES IN CASE OF REPLICATED AGENTS**

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 sec</td>
<td>14.88 sec</td>
<td>4.2 sec</td>
<td>4.41 sec</td>
</tr>
</tbody>
</table>

VII. DISCUSSION

Here we have presented our work on monitoring and selection of PlanetLab nodes based on CPU capacity requirements of an agent. In subsequent work we have developed similar models for memory requirements. In our future work we plan to develop monitoring models for bandwidth requirements.

In selecting eligible nodes for a given CPU requirement, the deployment manager utilizes only the current values of the idle CPU capacity. One potential way to enhance the goodness of the node recommendation is to take into consideration a node’s eligibility periods in the recent past. However, implementing this approach requires maintaining node eligibility period profile. One way to do this is to maintain a node’s profile information for a small set of CPU capacity requirements and interpolate this information for other requirements.

In our experiments we observed that the time taken to transfer a service agent’s secondary storage state can vary substantially. One option to address this issue is to consider approaches that pro-actively select and transfer secondary storage state to a potential target node for any future relocations. In selecting a target node our current work can also be enhanced to take into consideration bandwidth data from slicestat.

In many of our experiments we have observed that the service agents crash due to rebooting of the PlanetLab nodes on which they are running. This observation suggests that agent regeneration capability is crucial in such an environment. In our future work we plan to utilize autonomic techniques [20] to realize agent regeneration capabilities in this system.
Finally, the scope of our current work does not encompass fault-tolerance issues of the DHT infrastructure. Therefore, in our current setup we run the DHT on reliable nodes in our lab environment rather than running it on the PlanetLab nodes.

VIII. RELATED WORK

The notion of dynamic service relocation for fault-tolerance has been studied in the past by other researchers [15], [13]. HydraNet-FT [15] addresses the problem of replication and dynamic regeneration of services in a wide area network, but it requires specially equipped routers as redirection points. The focus of the work in [13] is on service relocation in context-aware applications, where services execute on mobile personal devices in an ad hoc network. The work in [22] investigated cost models related to web service migration. Other researchers have developed server fault-tolerance techniques based on TCP connection redirection or migration [8], [18]. Such techniques require customized modifications to the operating systems kernel or the routers. The focus of our work is on autonomic service relocation in wide area networks.

Operating system level techniques such as process migration [9], and shadow processes [7] have been used in the past for relocating a process in the network. Issues related to system level process state migration have been studied in systems such as Sprite [5] and Condor [7] and discussed in [9]. The Condor system specifically aims at utilizing lightly loaded nodes in a network by migrating a process from one node to another. The issues and policies in selecting a node for load balancing are identified and discussed in [16].

In contrast to process migration mechanisms, mobile agent systems such as Ajanta and others [6] use application-level state transfer and avoid some of the issues related to system level state transfer. The mobile agent paradigm has been successfully utilized by us and others for several network based distributed applications [20], [12]. Through the work presented in this paper we have demonstrated its use in one more important area, i.e. for building migratory services in wide-area networks.

A number of research projects have developed and investigated techniques for monitoring resource utilization and node availability over the Internet [23] and also specifically on the PlanetLab [10], [21], [3]. The latter three systems have used the data from CoMon [11] for this purpose. In contrast to these systems the focus of our work is on building a system which uses recent utilization data to select a set of nodes meeting a given resource capacity requirement.

IX. CONCLUSION

In this work we have presented the design and evaluation of a system for supporting migratory services in wide-area shared computing environments. The mobile agent paradigm is used for building an autonomous migratory service. An agent can encapsulate the policies for making the migration decisions on its own. These policies can be based on the current service load and the available resources capacities at the current host. We developed and evaluated this system over the PlanetLab.

Relocating a service agent requires selection of a suitable target node. For this purpose we developed a system which continuously monitors the PlanetLab nodes. This system is used to provide to a mobile service agent a set of nodes that have the required resource capacities currently free. We evaluated the performance of this system with three different policies for selecting nodes. We focused on requirements for idle CPU capacity. The expected duration for which a selected node continues to meet the specified resource capacity requirement depends on the specified requirement.

In our experiments we deployed several migratory service agents over the PlanetLab and observed their behavior. The experiments demonstrated the capability of the migratory service agents to migrate autonomously. The time to relocate a service agent that does not have any footprint in the secondary storage is typically around five seconds. This time can significantly increase if a service has large footprint in the secondary storage that needs to be transported to a new location. This can result in unacceptable blackout periods for the clients. To eliminate such blackout periods, we developed the mechanisms for implementing a service using a group of agents which function in the primary-backup mode of operation. Because the agents in a service group can autonomously migrate, we required a coordination protocol which ensures that only one agent can be in the migrating state at any given time. We measured the service migration times and durations for which a service agent stayed at various nodes after each migration.

ACKNOWLEDGMENT

The authors want to thank Benjamin Simmons and Karthik Roop Kumar for their help in conducting some of the experiments related to the monitoring service.

REFERENCES


