Automatic Middleware Deployment Planning on Heterogeneous Platforms

Pushpinder Kaur Chouhan #1, Eddy Caron ∗2, Frédéric Desprez ∗3

#Department of Computer Science, University College Cork, Ireland.
1p.chouhan@cs.ucc.ie

∗Université de Lyon
LIP, UMR ENS Lyon - CNRS - INRIA - UCBL, Lyon
46, Allée d’Italie, F-69364 Lyon, France
2ENS Lyon
Eddy.Caron@ens-lyon.fr

3INRIA
Frederic.Desprez@inria.fr

Abstract—The use of many distributed, heterogeneous resources as a large collective platform offers great potential. A key issue for these grid platforms is middleware scalability and how middleware services can be mapped on the available resources. Optimizing deployment is a difficult problem with no existing general solutions. In this paper, we address the following problem: how to perform out an adapted deployment for a hierarchy of servers and resource brokers on a heterogeneous system?

Our objective is to generate a best platform from the available nodes so as to fulfill the clients demands. However, finding the best deployment among heterogeneous resources is a hard problem since it is close to find the best broadcast tree in a general graph, which is known to be NP-complete.

Thus, in this paper, we present a heuristic for middleware deployment on heterogeneous resources. We apply our heuristic to automatically deploy a distributed Problem Solving Environment on a large scale grid. We present experiments comparing the automatically generated deployment against a number of other reasonable deployments.

I. INTRODUCTION

Due to the scale of grid platforms, as well as the geography location of resources, middleware should be distributed to provide scalability and adaptability. Much work has focused on the design and implementation of distributed middleware [1], [2], [3], [4], [5]. To benefit most from such approaches, an appropriate mapping of middleware components to the distributed resource environment is needed. However, while middleware designers often note that this problem of deployment planning is an important problem, few solutions for efficient and automatic deployment exists. In other words, questions such as which (and how many) resources should be used and should the fastest and best-connected resource be used for middleware or as a computational resource remain difficult to answer for heterogeneous resources.

Software engineering processes for deployment and configuration (software deployment) of distributed system service-level functionality have been extensively researched [6], [7], [8]; however very little work has been done around system deployment [9], [10]. To the best of our knowledge, no deployment algorithm or model has been given for arranging the components of a problem solving environment (PSE) in such a way as to maximize the number of requests that can be treated in a time unit on heterogeneous resources. More related work is given in Section II.

The goal of this paper is to provide an automated approach to deployment planning that provides a good deployment for client-server frameworks on
heterogeneous platforms. We consider that a good deployment is the one that maximizes the throughput of the system in number of client requests. We define deployment planning as the process of assigning resources to middleware components with the goal of optimizing throughput. In this paper the terms “deployment planning” is often shortened to “deployment”.

We use a distributed Problem Solving Environment called DIET (Distributed Interactive Engineering Toolbox) [1] as a test case for our approach. It uses a hierarchical arrangement of agents\(^1\) and servers\(^2\) to provide scalable, high-throughput scheduling and application provision services clients\(^3\). Figure 1 gives the execution scheme of a computation request. A hierarchy is a simple and effective distribution approach and has been chosen by a variety of middleware environments as their primary distribution approach [11], [12], [13].

We describe a hierarchy as follows. A server \(s \in S\) has exactly one parent that is always an agent \(a \in A\). A root agent \(a \in A\) has one or more child agents and/or servers and no parents. Non-root agents \(a \in A\) have exactly one parent and two or more child agents and/or servers. The arrangement of these elements is shown in Figure 1. We don’t share resources between agents and servers. In a lot of case, it’s important for application to be run alone without disruption. Moreover this model is close to real platform, thus a lot of batch scheduler are deployed on a gateway to access to computational servers.

To apply our approach to DIET, we have developed performance models for DIET components. We present real-world experiments demonstrating that the deployment chosen by our approach performs well when compared to other reasonable deployments.

The rest of this paper is organized as follows.

\(^1\)Agents select potential servers from a list of servers maintained in the database by frequent monitoring and maps client requests to one of the servers.

\(^2\)Servers receive requests from clients and execute applications or services on their behalf.

\(^3\)Users that require computations.

Section II presents some related work around deployment and scheduling. Section III first presents the request performance modeling and then the steady-state throughput modeling of our problem. A deployment heuristic for heterogeneous platforms is presented in Section IV. Section V presents the experimental results to test the ability of designed deployment heuristic and models to predict the good real-world deployments. Finally, Section VI concludes the article and presents the future work.

II. RELATED WORK

A deployment is the mapping of a platform and middleware across many resources. Deployment can be broadly divided in two categories: software deployment and system deployment. Software deployment maps and distributes a collection of software components on a set of resources and can include activities such as configuring, installing, updating, and adapting a software system. Examples of tools that automate software deployment include SmartFrog [14] and Distributed Ant [15]. System deployment involves assembling physical hardware as well as organizing and naming whole nodes and assigning activities such as master and slave. Examples of tools that facilitate this process include the Deployment Toolkit [16] and Kadeploy [17].

ADAGE [18], [19] aims at automating applications deployments on a Grid environment. ADAGE relies on a generic description model for the applications: GADe. First, the user describes in its own formalism (the specific description) his/her application(s) he/she wants to deploy. Then, using a plugin ADAGE converts the specific description into the generic description GADe in XML (the plugin is related to the specific description, therefore it is the user’s job to write it). Then, using the XML files containing the available resources description, the deployment control parameters (submission method: \texttt{ssh} or \texttt{rsh}, the scheduler, the placement constraints…) ADAGE computes a deployment plan containing the mapping of each process on resources (the schedulers are also plugins, so one can bring its own, currently only round-robin is implemented). Finally, ADAGE transfers the necessary files (using \texttt{scp} or \texttt{rcp}), and launches the
applications. While it offers a generic description model for the applications, which allows to use it for a wide variety of deployments. It does not have efficient algorithms to create the deployment plan (only round-robin), but its modular architecture allows to extend its possibilities.

Jade [20] is a middleware which purpose is to deploy and dynamically manage applications. The idea is to have a system which deals with the maintenance and management operations, instead of a physical person. Each application to deploy has first to be encapsulated in a Fractal\(^4\) component which defines the incoming and outgoing interfaces, as well as controllers to control the behavior of the components (suspend, resume, or change its attributes). The user defines the application architecture in an XML file using an ADL (Architecture Description Language). This file contains the components description, their relations, the number of instances for the components, …The Software Installation Service component is in charge of installing the components on the cluster’s nodes, then the Cluster Manager maps the applications on the nodes. The autonomic manager executes and manages the components: it monitors the system’s state and reacts according to the values given by a set of sensors (for example it can add or remove instances of a particular component in order to adapt to the system load) in order to keep the system in a stable state. Jade is more a dynamic autonomic management software than a deployment software, it takes into account the environment thanks to sensors within the components. Nevertheless, it still has an application deployment system based on an encapsulation in Fractal components. The resources selection is done among a group of available nodes on a cluster.

Although these toolkits can automate many of the tasks associated with deployment, they do not automate the decision process of finding an appropriate mapping of specialized middleware components to resources so that the best performance can be achieved from the system.

To the best of our knowledge, no deployment algorithm or model has been given for arranging the components of a PSE in such a way as to maximize the number of requests that can be treated in a time unit. In [21], software components based on the CORBA component model are automatically deployed on the computational Grid. The CORBA component model contains a deployment model that specifies how a particular component can be installed, configured and launched on a machine. The authors note a strong need for deployment planning algorithms, but to date they have focused on other aspects of the system. Our work is thus complementary.

In [22] we presented a heuristic approach for improving deployments of hierarchical NES systems.

\(^4\)http://fractal.objectweb.org/
in heterogeneous Grid environments. The approach is iterative; in each iteration, mathematical models are used to analyze the existing deployment, identify the primary bottleneck, and remove the bottleneck by adding resources in the appropriate area of the system. The techniques given in [22] are heuristic and iterative in nature and can only be used to improve the throughput of a deployment that has been defined by other means; the current work provides an optimal solution to a more limited case and does not require a predefined deployment as input. In [23], we presented a decision builder tool to analyze existing hierarchical deployments, used mathematical models to identify bottleneck nodes, and removed the bottleneck by adding resources in the appropriate part of the system. The solution presented was iterative and heuristic in nature. In [24] we presented an approach for automatically determining optimal deployment for hierarchically distributed services on heterogeneous clusters. We proved that a complete spanning d-ary tree provides an optimal deployment and presented an algorithm to construct this optimal tree.

Optimizing deployment is an evolving field. In [25], the authors propose an algorithm called Sikitei to address the Component Placement Problem (CPP). This work leverages existing AI planning techniques and the specific characteristics of CPP. In [26] the Sikitei approach is extended to allow optimization of resource consumption and consideration of plan costs. The Sikitei approach focuses on satisfying component constraints for effective placement, but does not consider detailed but sometimes important performance issues such as the effect of the number of connections on a component’s performance.

The Pegasus System [27] frames workflow planning for the Grid as a planning problem. The approach is interesting for overall planning when one can consider that individual elements can communicate with no performance impact. Our work is more narrowly focused on a specific style of assembly and interaction between components and has a correspondingly more accurate view of performance to guide the deployment decision process.

Finally, this work is also somehow related to the design of efficient broadcast trees in grids [28], [29]. However, the big difference is that in our case, nodes also compute and this cost is taken into account.

III. Steady-state Throughput Modeling

In order to apply the heuristic presented in Section IV to DIET, we require models for the scheduling and service throughput in DIET. Variables used in the formation of performance models to estimate the time required for various phases of request treatment are the following.

- \( \rho \) is the throughput of the platform, \( \rho_{sched} \), the scheduling request throughput of resource \( i \), \( \rho_{service} \), the service request throughput, \( S_{req} \), the size of incoming requests, \( S_{rep} \), the size of the reply, \( w_i \), the computing power of resource \( i \), \( d_i \) the children supported by resource \( i \), \( W_{pre.i} \), the amount of computation of resource \( i \) needed to process the servers replies, \( W_{fix.i} \), a fixed cost to process the server reply, \( W_{app.i} \), the amount of computation needed by a server \( i \) to complete a service request for app service, \( B \), the bandwidth of the link between resources, and \( N \), the number of requests completed by all the servers in a time step. In our communication model, we assume that communication links are homogeneous, which is the case of our target platform.

Agent communication model: To treat a request, an agent receives the request from its parent, sends the request to each of its children, receives a reply from each of its children, and sends one reply to its parent. The time in seconds required by an agent \( i \) for receiving all messages associated with a request from its parent and \( d_i \) children is given by:

\[
agent_{receive\ time} = \frac{S_{req} + d_i \cdot S_{rep}}{B}
\]  

Similarly, the time in seconds required by an agent for sending all messages associated with a request to its \( d_i \) children and parent is given by:

\[
agent_{send\ time} = \frac{d_i \cdot S_{req} + S_{rep}}{B}
\]
Server communication model: Servers have only one parent and no children, so the time in seconds required by a server for receiving messages associated with a scheduling request is given by:

\[ server_{\text{receive\_time}} = \frac{S_{\text{req}}}{B} \] (3)

The time in seconds required by a server for sending messages associated with a request to its parent is given by:

\[ server_{\text{send\_time}} = \frac{S_{\text{rep}}}{B} \] (4)

Agent computation model: Agents perform two activities involving computation: the processing of incoming requests and the selection of the best server amongst the replies returned by the agent’s children. There are two activities in the treatment of replies: a fixed cost \( W_{\text{fix}} \) in MFlops and a cost \( W_{\text{sel}} \) that is the amount of computation in MFlops needed to process the server replies, and select the best server by agent \( i \). Thus, the computation associated with the treatment of replies is given by

\[ W_{\text{rep},i}(d_i) = W_{\text{fix}} + W_{\text{sel}} \cdot d_i \]

The time in seconds required by the agent for the two activities is given by the following equation.

\[ agent_{\text{comp\_time}} = \frac{W_{\text{req}} + W_{\text{rep},i}(d_i)}{w_i} \] (5)

Server computation model: Servers also perform two activities involving computation: performance prediction as part of the scheduling phase and provision of application services as part of the service phase. We suppose that a deployment with a set of servers \( S \) completes \( N \) requests in a given time step. Then each server \( i \) will complete \( N_i \) requests, a fraction of \( N \) such that:

\[ \sum_{i=1}^{N} N_i = N \] (6)

On average each server \( i \) do prediction of \( N \) requests and complete \( N_i \) service requests in a time step. For example, the servers as a group require \( T \) seconds to complete \( N \) requests, then

\[ T(N) = \frac{W_{\text{pre},i} \cdot N + W_{\text{app},i} \cdot N_i}{w_i} \] (7)

From Equation 7, we can calculate the requests completed by each server \( i \) as:

\[ N_i = \frac{T \cdot w_i - W_{\text{pre},i} \cdot N}{W_{\text{app},i}} \] (8)

From Equations 6 and 8, we get time taken by the servers to process \( N \) requests as

\[ T = N \cdot \frac{1 + \sum_{i=1}^{N} W_{\text{pre},i}}{\sum_{i=1}^{N} w_i \cdot W_{\text{app},i}} \] (9)

so, time taken by the servers to process one request is

\[ server_{\text{comp\_time}} = \frac{1 + \sum_{i=1}^{N} W_{\text{pre},i}}{\sum_{i=1}^{N} w_i \cdot W_{\text{app},i}} \] (10)

Now we use these constraints to calculate the throughput of each phase of the middleware deployment. We use a model with no internal parallelism, \( M(r, s, w) \), [30] for the capability of a computing resource to do computation and communication. In this model, a computing resource has no capability for parallelism. It can either send a message, receive a message, or compute. Only a single port is assumed. Messages must be sent and received serially.

Servicing throughput of server \( i \) is then:

\[ \rho_{\text{service},i} = \frac{N_i}{T + \frac{S_{\text{req}} + S_{\text{rep}}}{B} \cdot N} \] (11)

So according to Equations 6 and 11, servicing throughput of platform is given by:

\[ \rho_{\text{service}} = \sum_{i=1}^{N} \frac{N_i}{T + \frac{S_{\text{req}} + S_{\text{rep}}}{B} \cdot N} = \frac{1}{\frac{1 + \sum_{i=1}^{N} W_{\text{pre},i}}{\sum_{i=1}^{N} w_i \cdot W_{\text{app},i}} + \frac{S_{\text{req}} + S_{\text{rep}}}{B}} \] (12)

(13)

The scheduling throughput \( \rho_{\text{sched}} \) in requests per second is given by the minimum of the throughput provided by the servers for prediction and by the agents for scheduling as shown below.
\[ \rho_{\text{sched}} = \min_{i \in \mathcal{V}} \left( \frac{1}{W_{\text{req},i} + \frac{S_{\text{req}}}{B} + \frac{S_{\text{rep}}}{B}} \right) \]

\[ W_{\text{req},i} + W_{\text{rep}}(d_i) \]

\[ \frac{1}{S_{\text{req}} + d_i S_{\text{rep}} + d_i S_{\text{req}} + S_{\text{rep}} + \frac{S_{\text{req}}}{B} + \frac{S_{\text{rep}}}{B} + \frac{S_{\text{req}}}{B} + \frac{S_{\text{rep}}}{B} + \frac{1}{\sum_{i=1}^{N} \frac{W_{\text{pre},i}}{W_{\text{app},i}}}} \]

(14)

\[ \rho_{\text{service}} = \min_{i \in \mathcal{V}} \left( \frac{1}{S_{\text{req}} + \frac{S_{\text{req}}}{B} + \frac{1}{\sum_{i=1}^{N} \frac{W_{\text{pre},i}}{W_{\text{app},i}}}} \right) \]

(15)

In the servicing phase, only the server participate, as a result, the \( \rho_{\text{service}} \) is calculated from server’s computation constraint as shown below.

\[ \rho = \min_{i \in \mathcal{V}} \left( \frac{1}{W_{\text{req},i} + \frac{S_{\text{req}}}{B} + \frac{S_{\text{rep}}}{B}} \right) \]

\[ W_{\text{req},i} + W_{\text{rep}}(d_i) + \frac{S_{\text{req}}}{B} + \frac{d_i S_{\text{rep}}}{B} + \frac{d_i S_{\text{req}}}{B} + \frac{S_{\text{rep}}}{B} + \frac{1}{\sum_{i=1}^{N} \frac{W_{\text{pre},i}}{W_{\text{app},i}}}} \]

(16)

We use this formula to calculate the throughput of the hierarchy in our experiments.

IV. HEURISTIC FOR MIDDLEWARE DEPLOYMENT

Our objective is to find a deployment that provides the maximum throughput (\( \rho \)) of completed requests per second. A completed request is one that has completed both the scheduling and service request phases and for which a response has been returned to the client. When the maximum throughput can be achieved by multiple distinct deployments, the preferred deployment is the one using the least resources.

The platform architecture that we target is composed of heterogeneous resources that have homogeneous connectivity. The working phases of the architecture is shown in Figure 1.

The throughput of the deployment platform is dependent on the scheduling throughput (\( \rho_{\text{sched}} \)) of each node and servicing throughput (\( \rho_{\text{service}} \)) of the deployment. Scheduling throughput and servicing throughput, as well as placement of the nodes in the hierarchy depend on the computing power of the nodes. It is very difficult to consider heterogeneity of computing resource and communication at the same time. In this primary work we focus on heterogeneous computing resource and consider homogeneous communication. In case of cluster it is not so far from the reality but the results will be different when we consider communications between clusters. We plan to deal with heterogeneous communication in future works.

For sake of simplicity we have defined some procedures (Table I) for the middleware deployment heuristic 1. Variables used in the heuristic 1 are presented in Table II. The heuristic is based on the exact calculation of number of children supported by each node. As scheduling power of any agent is limited by the number of children that it can support, to select a node that can act as agent, we calculate the scheduling power of each node with a number of children equal to the available nodes, \( n_{\text{nodes}} \). Actually, it is not only the number of children that effects the scheduling power of the parent node but also the computing power of its children (that means the computing power dependency is taken into account). At this point we ignore which node will be agent, so as to remove from the total nodes, thus, initially we take children equal to \( n_{\text{nodes}} - 1 \).

First, we sort the nodes according to scheduling power with \( n_{\text{nodes}} \) children in descending order (Step 1 and 2). Top node in the sorted list is most suitable to be an agent. Now we calculate the scheduling power of each node only with one child, so as to calculate the maximum scheduling power possible by the node (Step 3). Step 6 checks
**calc_hier_ser_pow**: is a function to calculate the servicing power provided by the hierarchy when load is equally divided among the servers of the hierarchy.

**calc_sch_pow**: is a function to calculate the scheduling power of a node according to the computing power of the node and number of its children.

**plot_hierarchy**: is a function to fill the adjacency matrix. Adjacency matrix is filled according to the number of children that each agent (from agent array) can support.

**shift_nodes**: is a function to shift up the node id in the server array if any server is converted as an agent.

**sort_nodes**: is a function to sort the available nodes according to there scheduling power calculated by function **calc_sch_pow**.

**write_xml**: is a function to generate an XML file according to the adjacency matrix, that is given as an input to deployment tool to deploy the hierarchical platform.

<table>
<thead>
<tr>
<th>Table I</th>
<th>Procedures Used in Heuristic 1.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>diff</strong></td>
<td>store minimum throughput among, ( \rho_{\text{sched}}, \rho_{\text{service}} ) which is calculated in previous step, and client demand</td>
</tr>
<tr>
<td><strong>min_ser_cv</strong></td>
<td>store minimum values among servicing throughput of hierarchy and client demand</td>
</tr>
<tr>
<td><strong>n_nodes</strong></td>
<td>total nodes available for hierarchy construction</td>
</tr>
<tr>
<td><strong>throughput_diff</strong></td>
<td>store minimum throughput among, ( \rho_{\text{sched}}, \rho_{\text{service}} ), and client demand</td>
</tr>
<tr>
<td><strong>vir_max_sch_pow</strong></td>
<td>scheduling power of top node in sorted nodes array</td>
</tr>
<tr>
<td><strong>vir_max_ser_pow</strong></td>
<td>servicing power of the hierarchy</td>
</tr>
<tr>
<td><strong>sorted_nodes</strong></td>
<td>array to store the id of sorted nodes</td>
</tr>
<tr>
<td><strong>supported_children</strong></td>
<td>number of child nodes supported by an agent</td>
</tr>
</tbody>
</table>

**Table II**

<table>
<thead>
<tr>
<th>Variables Used in Heuristic 1.</th>
</tr>
</thead>
<tbody>
<tr>
<td>if the scheduling power of the node is less than the client demand then one level hierarchy is deployed with one agent and one server (top two nodes of the sorted list), because if more servers are added to the node, scheduling power will decrease. But if scheduling power is higher than client demand, then servicing power is increased by adding new nodes. For this new agents could be added.</td>
</tr>
<tr>
<td>In Step 14, we calculate the scheduling throughput of the top node in the sorted list by incrementing number of its children. If any other node can support more than one child, then that node is added in the hierarchy with the children that the node can support. Again servicing power is calculated considering the total number of servers in the hierarchy. Steps 15 to 39 are repeated until all the nodes are used or client demand is fulfilled or throughput of the hierarchy starts decreasing. Then the connection between the nodes is presented in the form of adjacency matrix in Step 40. In Step 41 hierarchy is represented written in an XML file which is used by the deployment tool.</td>
</tr>
</tbody>
</table>

**V. Experimental Results**

In this section we present our experiments to test the ability of our deployment model to correctly identify good real-world deployments. The deployment method described in section focuses on maximizing steady-state throughput, as a result, we focus our experiments on testing the maximum sustained throughput provided by different deployments.
Algorithm 1 Pseudo-code describing heuristic to find the best hierarchy

1: Compute calc_pow() for each node with n\_nodes-1 children
2: Compute sorted\_nodes[] by using sort\_nodes()
3: Compute vir\_max\_sch\_pow by using calc\_sch\_pow() with sorted\_nodes[0] as an agent
4: Compute vir\_max\_ser\_pow by using calc\_ser\_pow() with sorted\_nodes[1] as a server
5: Compute min\_ser\_pow
6: if (vir\_max\_sch\_pow < min\_ser\_pow) then
7: Deployment is just 1 agent and 1 server
8: else
9: Compute throughput\_diff and diff is a maximum arbitrary value
10: while (diff>throughput\_diff) do
11: diff=throughput\_diff;
12: Compute vir\_max\_sch\_pow with supported children equal to 2
13: Select next node from sorted\_nodes[] as server
14: Compute vir\_max\_ser\_pow and throughput\_diff
15: while (\(\exists\) unused nodes and vir\_max\_sch\_pow > vir\_max\_ser\_pow) do
16: if the number of children supported by the current server node >1 then
17: Change server to agent using shift\_nodes() and add next node from the sorted\_nodes[] as a child to this new converted agent
18: while the number of children < supported\_children by the node do
19: Compute vir\_max\_ser\_pow with one extra child
20: if ((vir\_max\_ser\_pow<client\_volume)\&\&(\(\exists\) unused nodes)\&\&(vir\_max\_ser\_pow<vir\_max\_sch\_pow)) then
21: Take next node from sorted\_nodes[] as a server
22: Compute vir\_max\_ser\_pow
23: end if
24: end while
25: end if
26: Compute new throughput\_diff
27: end while
28: if (vir\_max\_sch\_pow<vir\_max\_ser\_pow) then
29: if (diff<throughput\_diff) then
30: Remove 1 child from the last agent
31: else
32: diff = throughput\_diff
33: end if
34: end if
35: if (supported\_children == n\_nodes-1) then
36: diff=throughput\_diff;
37: end if
38: end while
39: end if
40: plot\_hierarchy();
41: write\_xml();

A. Experimental Environment

DIET 2.0 is used for all deployed agents and servers; GoDIET [31] version 2.0.0 is used to perform the actual software deployment. In general, at the time of deployment, one can know neither the exact job mix nor the order in which jobs will arrive. Instead, one has to assume a particular job mix, define a deployment, and eventually correct the deployment after launch if it was not well-chosen. For these tests, we use the DGEMM application, a simple matrix multiplication provided as part of the level 3 BLAS package [32].

Measuring the maximum throughput of a system is non-trivial: if too little load is introduced the maximum performance may not be achieved, if too much load is introduced the performance may suffer as well. A unit of load is introduced via a script that runs a single request at a time in a continual loop. We then introduce load gradually
by launching one client script every second. We introduce new clients until the throughput of the platform stops improving; we then let the platform run with no addition of clients for 10 minutes.

Table III presents the parameter values we used for DIET in the model. To measure message sizes $S_{req}$ and $S_{rep}$, we deployed an agent and a single DGEMM server on the Lyon cluster and then launched 100 clients serially from the same cluster. We collected all network traffic between the agent and the server machines using tcpdump and analyzed the traffic to measure message sizes using the Ethereal Network Protocol analyzer\(^5\). This approach provides a measurement of the entire message size including headers. Using the same agent-server deployment, 100 client repetitions, and the statistics collection functionality in DIET. Detailed measurements of the time required to process each message at the agent and server level was also recorded. The parameter $W_{rep}$ depends on the number of children attached to an agent. We measured the time required to process responses for a variety of star deployments including an agent and different numbers of servers. A linear data fit provided a very accurate model for the time required to process responses versus the degree of the agent with a correlation coefficient of 0.97. Thus, this linear model is used for the parameter $W_{rep}$. Finally, we measured the capacity of our test machines in MFlops using a mini-benchmark extracted from Linpack and this value is used to convert all measured times to estimates of the MFlops required.

**B. Deployment approach validation on homogeneous platform**

The usefulness of our deployment heuristic depends heavily on the performance model of the middleware. This section presents experiments designed to show the correctness of the performance model presented in previous section.

Experimental results shown in Figures 2 and 3 uses a workload of DGEMM 10x10 to compare the performance of two hierarchies: an agent with one

\(^5\)http://www.ethereal.com
server versus an agent with two servers. The model correctly predicts that both deployments are limited by agent performance and that the addition of the second server will in fact hurt performance. More important is the correct prediction by our model to judge the effect of adding servers than to correctly predict the throughput of the platform (which is tough using a small computation grain for which cache effects improve performance).

Experimental results shown in Figures 4 and 5 use a workload of DGEMM 200x200 to compare the performance of the same one and two servers hierarchies. In this scenario, the model predicts that both hierarchies are limited by server performance and therefore, performance will roughly double with the addition of the second server. The model correctly predicts that the two-server deployment will be the better choice.

In summary, our deployment performance model is able to accurately predict the impact of adding servers to a server-limited or agent-limited deployment.

To verify our heuristic we compared the predicted deployment given by the heuristic with the experimental results presented in [24]. Table IV presents the comparison by reporting the percentage of optimal throughput achieved by the deployments selected by different means.

### C. Heuristic validation on heterogeneous cluster

To validate our heuristic we did experiments using two sites, Lyon and Orsay of the french experimental Grid called Grid’5000, a set of distributed computational resources deployed in France. We
used 200 nodes of Orsay for the deployment of middleware elements and 30 nodes of Lyon for submitting the requests to the deployed platform.

To convert the homogeneous cluster into heterogeneous cluster, we changed the workload of the reserved nodes by launching different size of matrix multiplication as the background program on some of the nodes. After launching the matrix program in the background on the machines we used Linpack mini-benchmark to measure the capacity of the nodes in MFlops.

We compared two different deployments with the automatically generated deployment by our heuristic. The first deployment is a simple star type, where one node acts as an agent and all the rest are directly connected to the agent node (and these act as servers). In the second deployment, we deployed a balanced graph, one top agent connected to 14 agents and each agent connected to 14 servers with the exception of one agent with only 3 servers.

Clients submitted DGEMM problems of two different sizes. First we tested the deployments with DGEMM using 310x310 matrices. The heuristic generated deployment used only 156 nodes and deployment is organized as: top agent connected with 9 agents and each agent again connected to 9 agents. Two agents are connected with 9 servers, 6 agents are connected with 7 servers and one with 5 servers. Automatically generated deployment performed better than the two compared deployments. Results of the experimental results are shown in Figure 6.

Second experiment is done with DGEMM with matrices of size 1000x1000. Heuristic generated a balanced graph, one top agent connected to 14 agents and each agent connected to 14 servers with the exception of one agent with only 3 servers.

Fig. 6. Comparison of automatically-generated hierarchy for DGEMM 310x310 with intuitive alternative hierarchies.

Fig. 7. Comparison of automatically-generated hierarchy for DGEMM 1000*1000 with intuitive alternative hierarchy.
star deployment for this problem size. Results in
Figure 7 shows that star performed better than the
second compared deployment.

VI. CONCLUSION AND FUTURE WORK

We presented a deployment heuristic that accu-
trately predicts the maximum throughput that can
be achieved by the use of available nodes. The
heuristic predicts a good deployment hierarchies for
both homogeneous and heterogeneous resources. A
comparison is made to test the heuristic for homo-
geogeneous resources and the heuristic performed up
to 90% as compared to the homogeneous optimal
algorithm presented in [24]. To validate the heuris-
tic, experiments are performed on the Grid’5000
platform. Experiments have shown that automati-
cally generated deployment by the heuristic per-
forms better than some intuitive deployments for
heterogeneous platforms.

In the near future one of our principal objectives
is to implement the theoretical deployment plan-
techniques as Automatic Deployment Planning
Tool (ADePT). It will be interesting to validate
our theoretical concept of deployment planning
by further experimentation with other hierarchical
middlewares. We would also like to implement
deployment planning for arbitrary arrangements
of distributed resources.

In this model we consider that we have a function
to know the execution time but we should study an-
other approach with statistical mathematical func-
tion to forecast the execution time. Finally, we are
interested to find a modelization to deploy several
middlewares and/or applications on grid.

REFERENCES

[1] E. Caron and F. Desprez, “DIEF: A Scalable Tool-
box to Build Network Enabled Servers on the Grid”,
International Journal of High Performance Comput-
ing infrastructure toolkit”, The International Journal
of Supercomputer Applications and High Performance
“Webcom-G: grid enabled metacomputing”, Neural,
implementations of NINF: towards a global computing
and the Grid”, in Grid Computing: Making the Global
Infrastructure a Reality, Fran Berman, Geoffrey Fox, and
[6] A. Akkerman, A. Totok, and V. Karamcheti, “Infras-
structure for automatic dynamic deployment of j2ee ap-
lications in distributed environments”, in Component
Deployment, 2005, pp. 17–32.
methodology for developing and deploying distributed
applications”, in Component Deployment, 2005, pp. 37–
51.
description model: Toward automatic deployment of ap-
lications on computational grids”, in GRID ’05: Pro-
cedings of the 6th IEEE/ACM International Workshop
Component Deployment in Wide-Area Networks Using
components on a computational grid: General principles
and early experiments using the globus toolkit”, in
archical Processor Scheduling in Shared-Memory Multi-
processor Systems”, IEEE Trans. on Computers, vol. 48,
[12] A.W. Halderen, B.J. Overeinder, and P.M.A. Sloot, “Hi-
erarchical Resource Management in the Polder Metacom-
A. Sloot, “Simulation of Hierarchical Job Management for
configuration”, in Large Scale System Configuration
Workshop, 2001, National e-Science Centre UK, http:
//www.hpl.hp.com/research/smartfrog/.
system to support application deployment in the Grid”, in
Proceedings of the Fifth IEEE/ACM International
[16] Dell Power Solutions, “Simplifying system deployment
using the Dell OpenManage Deployment Toolkit”, Octo-
of PC”, in Parallel Computing, Proceedings of the
network topology description model for grid application
deployment”, in Proceedings of the 5th IEEE/ACM In-


Eddy Caron is an assistant professor at Ecole Normale Supérieure de Lyon and holds a position with the LIP laboratory (ENS Lyon, France). He is a member of GRAAL project and technical manager for the DIET software package. He received his PhD in C.S. from University de Picardie Jules Verne in 2000. His research interests include parallel libraries for scientific computing on parallel distributed memory machines, problem solving environments, and grid computing. See http://graal.ens-lyon.fr/~ecaron for further information.

Frederic Desprez is a director of research at INRIA and holds a position at LIP laboratory (ENS Lyon, France). He received is PhD in C.S. from the Institut National Polytechnique de Grenoble in 1994 and his MS in C.S. from the ENS Lyon in 1990. His research interests include scheduling for large scale distributed platforms, SOA architectures, data management, and grid computing. See http://graal.ens-lyon.fr/~desprez for further information.

Pushpinder Kaur Chouhan is a Post-doctoral researcher working in Centre for Unified Computing, University College Cork, Cork, Ireland. She got her Research Masters and PhD in Computer Science from ENS-Lyon, France. She did her Bachelor’s degree in Computer Science and Engineering from SLIET, Punjab, India. She has reviewed for a number of conference and journal publications. Her areas of interests include Automatic Deployment, Grid & Distributed Computing.