Deployment of a Hierarchical Middleware

Eddy Caron, Benjamin Depardon, and Frédéric Desprez

University of Lyon, LIP Laboratory, UMR CNRS - ENS Lyon
INRIA - UCBL 5668, France
{Eddy.Caron,Benjamin.Depardon,Frederic.Desprez}@ens-lyon.fr

Abstract. Accessing the power of distributed resources can nowadays easily be done using a *middleware* based on a client/server approach. Several architectures exist for those middleware. The most scalable ones rely on a hierarchical design. Determining the best shape for the hierarchy, the one giving the best throughput of services, is not an easy task.

We first propose a computation and communication model for such hierarchical middleware. Our model takes into account the deployment of several services in the hierarchy. Then, based on this model, we propose an algorithm for automatically constructing a hierarchy. This algorithm aims at offering the users the best obtained to requested throughput ratio, while providing fairness on this ratio for the different kind of services, and using as few resources as possible. Finally, we compare our model with experimental results on a real middleware called DIET.

Keywords: Hierarchical middleware, Deployment, Modelization.

1 Introduction

Using distributed resources to solve large problems ranging from numerical simulations to life science is nowadays a common practice. Several approaches exist for porting these applications to a distributed environment; examples include classic message-passing, batch processing, web portals and GridRPC systems. In this last approach, clients submit computation requests to a meta-scheduler (also called agent) that is in charge of finding suitable servers for executing the requests within the distributed resources. Scheduling is applied to balance the work among the servers. A list of available servers is sent back to the client; which is then able to send the data and the request to one of the suggested servers to solve its problem.

There exists several grid middleware to tackle the problem of finding services available on distributed resources, choosing a suitable server, then executing the requests, and managing the data. Several environments, called Network Enabled Servers (NES) environments, have been proposed. Most of them share a common characteristic which is that they are built with broadly three main components: *clients* which are applications that use the NES infrastructure, *agents* which are in charge of handling the clients' requests (scheduling them) and of finding suitable servers, and finally *computational servers* which provide computational

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 343–354, 2010.

power to solve the requests. Some of the middleware only rely on basic hierarchies of elements, a star graph, such as Ninf-G [1] and NetSolve [2]. Others, in order to divide the load at the agents level, can have a more complicated hierarchy shape: WebCom-G [3] and DIET [4,5]. In this latter case, a problem arises: what is the *best* shape for the hierarchy?

In this paper, we will mainly focus on one particular hierarchical NES: DIET (**D**istributed **I**nteractive **E**ngineering **T**oolbox). The DIET component architecture is structured hierarchically as a tree to obtain an improved scalability. Such an architecture is flexible and can be adapted to diverse environments, including arbitrary heterogeneous computing platforms. DIET comprises several components. *Clients* that use DIET infrastructure to solve problems using a remote procedure call (RPC) approach. SED*s,* or server daemons, act as service providers, exporting functionalities via a standardized computational service interface; a single SED can offer any number of computational services. Finally, *agents* facilitate the service location and invocation interactions of clients and SEDs. Collectively, a hierarchy of agents provides higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single *Master Agent (MA)* (the root of the hierarchy) and several *Local Agents (LA)* (internal nodes).

Deploying applications on a distributed environment is a problem that has already been addressed. We can find in the literature a few deployment software: DeployWare [6], ADAGE [7], TUNe [8], and GODIET [9]. Their field of action ranges from single deployment to autonomic management of applications. However, none include intelligent deployment mapping algorithms. Either the mapping has to be done by the user, or the proposed algorithm is random or round-robin. Some algorithms have been proposed in [10,11] to deploy a hierarchy of schedulers on clusters and grid environments. However, a severe limitation in these works is that only one kind of service could be deployed in the hierarchy. Such a constraint is of course not desirable, as nowadays many applications rely on workflows of services. Hence, the need to extend the previous models and algorithms to cope with hierarchies supporting several services.

The contribution of this paper is twofold. We first present a model for predicting the performance of a hierarchical NES on a homogeneous platform. As we will see this model can easily be applied to a computation heterogeneous platform. Secondly, we present an algorithm for automatically determining the *best* shape for the hierarchy, *i.e.,* the number of servers for each services, and the shape of the hierarchy supporting these servers.

We first present in Section 2 the hypotheses for our model, then the model itself in Section 3 for both agents and servers. Then, we explain our approach to automatically build a suitable hierarchy in Section 4, and finally compare the theoretical results with experimental results in Section 5, before concluding.

2 Model Assumptions

Request definition. Clients use a 2-phases process to interact with a deployed hierarchy: they submit a scheduling request to the agents to find a suitable

server in the hierarchy (the scheduling phase), and then submit a service request (job) directly to the server (the service phase). 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. We consider that a set $\mathcal{R}$ of services have to be available in the hierarchy. And that for each service $i \in \mathcal{R}$, the clients aim at attaining a throughput ρ_i^* of completed requests per seconds.

Resource architecture. In this paper we will focus on the simple case of deploying the middleware on a fully homogeneous, fully connected platform $G = (V, E, w, B)$, *i.e.,* all nodes' processing power are the same: w in $Mflops/s$, and all links have the same bandwidth: B in Mb/s. We do not take into account contentions in the network.

Deployment assumptions. We consider that at the time of deployment we do not know the clients locations or the characteristics of the clients' resources. Thus, clients are not considered in the deployment process and, in particular, we assume that the set of computational resources used by clients is disjoint from V. A valid deployment will always include at least the root-level agent and one server per service $i \in \mathcal{R}$. Each node $v \in V$ can be assigned either as a server for any kind of service $i \in \mathcal{R}$, or as an agent, or left idle. Thus with $|\mathcal{A}|$ agents, $|\mathcal{S}|$ servers, and $|V|$ total resources, $|\mathcal{A}| + |\mathcal{S}| \leq |V|$.

Objective. As we have multiple services in the hierarchy, our goal cannot be to maximize the global throughput of completed requests regardless of the kind of services, this would lead to favor services requiring only small amount of computing power for scheduling and solving phases, and with few communications. Hence, our goal is to obtain for each service $i \in \mathcal{R}$ a throughput ρ_i such that all services receive almost the same obtained throughput to requested throughput ratio: $\frac{\rho_i}{\rho_i^*}$, while having as few agents in the hierarchy as possible, so as not to use more resources than necessary.

3 Hierarchy Model

3.1 Overall Throughput

For each service $i \in \mathcal{R}$, we define ρ_{sched_i} to be the scheduling throughput for requests of type i offered by the platform, *i.e.,* the rate at which requests of type i are processed by the scheduling phase. We define as well ρ_{serv_i} to be the service throughput.

Lemma 1. *The completed request throughput ρ_i of type i of a deployment is given by the minimum of the scheduling and the service request throughput ρ_{sched_i} and ρ_{serv_i}.*

$$\rho_i = \min\{\rho_{sched_i}, \rho_{serv_i}\}$$

Lemma 2. *The service request throughput ρ_{serv_i} for service i increases as the number of servers included in a deployment and allocated to service i increases.*

3.2 Hierarchy Elements Model

We now precise the model of each element of the hierarchy. We consider that a request of type i is sent down a branch of the hierarchy, if and only if service i is present in this branch, *i.e.,* if at least a server of type i is present in this branch of the hierarchy. Thus a server of type i will never receive a request of type $j \neq i$. Agents will not receive a request i if no server of type i is present in its underlying hierarchy, nor will it receive any reply for such a type of request. This is the model used by DIET.

Server model. We define the following variables for the servers. w_{pre_i} is the amount of computation in $MFlops$ needed by a server of type i to predict its own performance when it receives a request of type i from its parent. Note that a server of type i will never have to predict its performance for a request of type $j \neq i$ as it will never receive such a request. w_{app_i} is the amount of computation in $MFlops$ needed by a server to execute a service. m_{req_i} is the size in Mb of the messages forwarded down the agent hierarchy for a scheduling request, and m_{resp_i} the size of the messages replied by the servers and sent back up the hierarchy. Since we assume that only the best server is selected at each level of the hierarchy, the size of the reply messages does not change as they move up the tree.

Server computation model. Let's consider that we have n_i servers of type i, and that n_i requests of type i are sent. On the whole, the n_i servers of type i require $\frac{n_i.w_{pre_i}+w_{app_i}}{w}$ time unit to serve the n_i requests: each server has to compute the performance prediction n_i times, and serve one request. Hence, on average, the time to compute one request of type i is given by Equation (1).

$$T_{comp_i}^{server} = \frac{w_{pre_i} + \frac{w_{app_i}}{n_i}}{w} \tag{1}$$

Server communication model. A server of type i needs, for each request, to receive the request, and then to reply. Hence Equations (2) and (3) represent respectively the time to receive one request of type i, and the time to send the reply to its parent.

$$T_{recv_i}^{server} = \frac{m_{req_i}}{B} \tag{2}$$

$$T_{send_i}^{server} = \frac{m_{resp_i}}{B} \tag{3}$$

Service throughput. Concerning the machines model, and their ability to compute and communicate, we consider the following models:

– Send or receive or compute, single port: a node cannot do anything simultaneously.

$$\rho_{serv_i} = \frac{1}{T_{recv_i}^{server} + T_{send_i}^{server} + T_{comp_i}^{server}} \tag{4}$$

– Send or receive, and compute, single port: a node can simultaneously send or receive a message, and compute.

$$\rho_{serv_i} = \min\left\{\frac{1}{T^{server}_{recv_i} + T^{server}_{send_i}}, \frac{1}{T^{server}_{comp_i}}\right\} \tag{5}$$

– Send, receive, and compute, single port: a node can simultaneously send and receive a message, and compute.

$$\rho_{serv_i} = \min\left\{\frac{1}{T^{server}_{recv_i}}, \frac{1}{T^{server}_{send_i}}, \frac{1}{T^{server}_{comp_i}}\right\} \tag{6}$$

Agent model. We define the following variables for the agents. w_{req_i} is the amount of computation in $MFlops$ needed by an agent to process an incoming request of type i. For a given agent $A_j \in \mathcal{A}$, let $Chld^j_i$ be the set of children of A_j having service i in their underlying hierarchy. Also, let δ^j_i be a Boolean variable equal to 1 if and only if A_j has at least one child which knows service i in its underlying hierarchy. $w_{resp_i}\left(\left|Chld^j_i\right|\right)$ is the amount of computation in $MFlops$ needed to merge the replies of type i from its $\left|Chld^j_i\right|$ children. This amount grows linearly with the number of children.

Our agent model relies on the underlying servers throughput. Hence, in order to compute the computation and communication times taken by an agent A_j, we need to know both the servers throughput ρ_{serv_i} for each $i \in \mathcal{R}$, and the children of A_j.

Agent computation model. The time for an agent A_j to schedule a request it receives and forwards is given by Equation (7).

$$T^{agent_j}_{comp} = \frac{\sum_{i\in\mathcal{R}} \rho_{serv_i}.\delta^j_i.w_{req_i} + \sum_{i\in\mathcal{R}} \rho_{serv_i}.w_{resp_i}\left(\left|Chld^j_i\right|\right)}{w} \tag{7}$$

Agent communication model. Agent A_j needs, for each request of type i, to receive the request and forwards it to the relevant children, then to receive the replies and forward the aggregated result back up to its parent. Hence Equations (8) and (9) present the time to receive and send all messages when the servers provide a throughput ρ_{serv_i} for each $i \in \mathcal{R}$.

$$T^{agent_j}_{recv} = \frac{\sum_{i\in\mathcal{R}} \rho_{serv_i}.\delta^j_i.m_{req_i} + \sum_{i\in\mathcal{R}} \rho_{serv_i}.\left|Chld^j_i\right|.m_{resp_i}}{B} \tag{8}$$

$$T^{agent_j}_{send} = \frac{\sum_{i\in\mathcal{R}} \rho_{serv_i}.\delta^j_i.m_{resp_i} + \sum_{i\in\mathcal{R}} \rho_{serv_i}.\left|Chld^j_i\right|.m_{req_i}}{B} \tag{9}$$

We combine (7), (8), and (9) according to the chosen communication / computation model (similar to Equations (4), (5), and (6)).

Lemma 3. *The highest throughput a hierarchy of agents is able to serve is limited by the throughput an agent having only one child of each kind of service can support.*

4 Automatic Planning

Given the models presented in the previous section, we propose a heuristic for automatic deployment planning. The heuristic comprises two phases. The first step consists in dividing N nodes between the services, so as to support the servers. The second step consists in trying to build a hierarchy, with the $|V| - N$ remaining nodes, which is able to support the throughput generated by the servers. In this section, we present our automatic planning algorithm in three parts. In Section 4.1 we present how the servers are allocated nodes, then in Section 4.2 we present a bottom-up approach to build a hierarchy of agents, and finally in Section 4.3 we present the whole algorithm.

4.1 Servers' Repartition

Our goal is to obtain for all services $i \in \mathcal{R}$ the same ratio $\frac{\rho_{serv_i}}{\rho_i^*}$. As we try to build a hierarchy of agents being able to support the throughput offered by the servers, in the end we will have $\rho_i = \rho_{serv_i}$. Algorithm 1 presents a simple way of dividing the available nodes to the different services. We iteratively increase the number of assigned nodes per services, starting by giving nodes to the service with the lowest $\frac{\rho_{serv_i}}{\rho_i^*}$ ratio.

Algorithm 1. Servers' repartition

Require: N: number of available nodes
Ensure: n: number of nodes allocated to the servers
1: $S \leftarrow$ list of services in $\mathcal{R}$
2: $n \leftarrow 0$
3: **repeat**
4: $\quad i \leftarrow$ first service in S
5: $\quad$ Assign one more node to i, and compute the new ρ_{serv_i}
6: $\quad n \leftarrow n + 1$
7: $\quad$ **if** $\rho_{serv_i} \geq \rho_i^*$ **then**
8: $\quad\quad \rho_{serv_i} \leftarrow \rho_i^*$
9: $\quad\quad S \leftarrow S - \{i\}$
10: $\quad S \leftarrow$ Sort services by increasing $\frac{\rho_{serv_i}}{\rho_i^*}$
11: **until** $n = N$ **or** $S = \emptyset$
12: **return** n

4.2 Agents' Hierarchy

Given the servers' repartition, and thus, the services' throughput ρ_{serv_i}, for all $i \in \mathcal{R}$, we need to build a hierarchy of agents that is able to support the throughput offered by the servers. Our approach is based on a bottom-up construction:

we first distribute some nodes to the servers, then with the remaining nodes we iteratively build levels of agents. Each level of agents has to be able to support the load incurred by the underlying level. The construction stops when only one agent is enough to support all the children of the previous level. In order to build each level, we make use of an integer linear program (ILP): ($\mathcal{L}_1$).

We first need to define a few more variables. Let k be the current level: $k = 0$ corresponds to the server level. For $i \in \mathcal{R}$ let $n_i(k)$ be the number of elements (servers or agents) obtained at step k, which know service i. For $k \geq 1$, we recursively define new sets of agents. We define by M_k the number of available resources at step k: $M_k = M_1 - \sum_{l=1}^{k-1} n_i(l)$. For $1 \leq j \leq M_k$ we define $a_j(k) \in \{0, 1\}$ to be a Boolean variable stating whether or not node j is an agent in step k. $a_j(k) = 1$ if and only if node j is an agent in step k. For $1 \leq j \leq M_k, \forall i \in \mathcal{R}, \delta_i^j(k) \in \{0, 1\}$ defines whether of not node j has service i in its underlying hierarchy in step k. For the servers, $k = 0$, $1 \leq j \leq M_0, \forall i \in \mathcal{R}, \delta_i^j(0) = 1$ if and only if server j is of type i, otherwise $\delta_i^j(0) = 0$. Hence, we have the following relation: $\forall i \in \mathcal{R}, n_i(k) = \sum_{j=1}^{M_k} \delta_i^j(k)$. For $1 \leq j \leq M_k, \forall i \in \mathcal{R}, \left|Chld_i^j(k)\right| \in \mathbb{N}$ is as previously the number of children of node j that know service i. Finally, for $1 \leq j \leq M_k, 1 \leq l \leq M_{k-1}$ let $c_l^j(k) \in \{0, 1\}$ be a Boolean variable stating that node l in step $k-1$ is a child of node j in step k. $c_l^j(k) = 1$ if and only if node l in step $k-1$ is a child of node j in step k.

Using integer program ($\mathcal{L}_1$), we can recursively define the hierarchy of agents, starting from the bottom of the hierarchy.

Let's have a closer look at ($\mathcal{L}_1$). Lines (1), (2) and (3) only define the variables. Line (4) states that any element in level $k-1$ has to have exactly 1 parent in level k. Line (5) counts, for each element at level k, its number of children that know service i. Line (6) states that the number of children of j of type i cannot be greater than the number of elements in level $k-1$ that know service i, and has to be 0 if $\delta_i^j(k) = 0$. The following two lines, (7) and (8), enforce the state of node j: if a node has at least a child, then it has to be an agent (line (7) enforces $a_j(k) = 1$ in this case), and conversely, if it has no child, then it has to be unused (line (8) enforces $a_j(k) = 0$ in this case). Line (9) states that at least one agent has to be present in the hierarchy. Line (10) is the transposition of the agent model in the *send or receive or compute, single port* model. Note that the other models can easily replace this model in ILP ($\mathcal{L}_1$). This line states that the time required to deal with all requests going through an agent has to be lower than or equal to one second.

Finally, our objective function is the minimization of the number of agents: the equal share of obtained throughput to requested throughput ratio has already been cared of when allocating the nodes to the servers, hence our second objective that is the minimization of the number of agents in the hierarchy has to be taken into account.

Remark 1. In order to improve the converge time to an optimal solution for integer program ($\mathcal{L}_1$), we can add the following constraint:

$$a_1(k) \geq a_2(k) \cdots \geq a_{M_k}(k) \tag{10}$$

$$
\begin{array}{l}
\textbf{Minimize } \sum_{j=1}^{M_k} a_j(k) \\
\textbf{Subject to} \\
\left\{ \begin{array}{lll}
(1) & 1 \leq j \leq M_k & a_j(k) \in \{0,1\} \\
(2) & 1 \leq j \leq M_k, \forall i \in \mathcal{R} & \delta_i^j(k) \in \{0,1\} \text{ and } \left|Chld_i^j(k)\right| \in \mathbb{N} \\
(3) & 1 \leq j \leq M_k, & \\
 & 1 \leq l \leq M_{k-1} & c_l^j(k) \in \{0,1\} \\
(4) & 1 \leq l \leq M_{k-1} & \sum_{j=1}^{M_k} c_l^j(k) = 1 \\
(5) & 1 \leq j \leq M_k, \forall i \in \mathcal{R} & \left|Chld_i^j(k)\right| = \sum_{l=1}^{M_{k-1}} c_l^j(k).\delta_i^l(k-1) \\
(6) & 1 \leq j \leq M_k, \forall i \in \mathcal{R} & \left|Chld_i^j(k)\right| \leq \delta_i^j(k).n_i(k-1) \\
(7) & 1 \leq j \leq M_k, i \in \mathcal{R} & \delta_i^j(k) \leq a_j(k) \\
(8) & 1 \leq j \leq M_k & a_j(k) \leq \sum_{i \in \mathcal{R}} \delta_i^j(k) \\
(9) & & \sum_{j=1}^{M_k} a_j(k) \geq 1 \\
(10) & 1 \leq j \leq M_k & \sum_{i \in \mathcal{R}} \rho_{serv_i} \times \\
 & & \left(\frac{\delta_i^j(k).w_{req_i} + w_{resp_i}\left(\left|Chld_i^j(k)\right|\right)}{w} + \right. \\
 & & \frac{\delta_i^j(k).m_{req_i} + \left|Chld_i^j(k)\right|.m_{resp_i}}{B} + \\
 & & \left. \frac{\delta_i^j(k).m_{resp_i} + \left|Chld_i^j(k)\right|.m_{req_i}}{B} \right) \leq 1
\end{array} \right. \quad (\mathcal{L}_1)
\end{array}
$$

This states that only the first nodes can be agents. This prevents the solver from trying all swapping possibilities when searching a solution. We can safely add this constraint, as we suppose that we have a homogeneous platform.

4.3 Building the Whole Hierarchy

So far, we did not talk about the repartition of the available nodes between agents and servers. We will now present the whole algorithm for building the hierarchy.

Maximum attainable throughput per service. Whatever the expected throughput for each service is, there is a limit on the maximum attainable throughput. Given Equations (7), (8) and (9), and the fact that a hierarchy must end at the very top by only one agent, the maximum throughput attainable by an agent serving all kinds of services (which is the case of the root of the hierarchy), is attained when the agent has only one child of each service (see Lemma 3). Hence, the maximum attainable throughput for each service, when all services receive the same served to required throughput ratio, from the agents' point of view is given by integer program $(\mathcal{L}_2)$ which computes $\rho_{serv_i}^{max}$ for $i \in \mathcal{R}$, the maximum attainable throughput for each service i that an agent can offer under the assumption that all services receive an equal share.

$$\textbf{Maximize } \mu$$
$$\textbf{Subject to}$$
$$\begin{cases} (1)\ \forall i \in \mathcal{R} & \mu \leq \dfrac{\rho^{max}_{serv_i}}{\rho^*_i} \text{ and } \mu \in [0,1], \rho^{max}_{serv_i} \in [0, \rho^*_i] \\ (2)\ \forall i, i' \in \mathcal{R} & \dfrac{\rho^{max}_{serv_i}}{\rho^*_i} = \dfrac{\rho^{max}_{serv_{i'}}}{\rho^*_{i'}} \\ (3)\ 1 \leq j \leq M_k & \sum_{i \in \mathcal{R}} \rho^{max}_{serv_i} \times \left(\dfrac{w_{req_i} + w_{resp_i}}{w} + \dfrac{2.m_{req_i} + 2.m_{resp_i}}{B} \right) \leq 1 \end{cases} \quad (\mathcal{L}_2)$$

When building the hierarchy, there is no point in allocating nodes to a service i if ρ_{serv_i} gets higher than $\rho^{max}_{serv_i}$. Hence, whenever a service has a throughput higher than $\rho^{max}_{serv_i}$, then we consider that its value is $\rho^{max}_{serv_i}$ when building the hierarchy. Thus, lines 7 and 8 in Algorithm 1 become:

7: **if** $\rho^{comp}_{serv_i} \geq \min\left\{\rho^*_i, \rho^{max}_{serv_i}\right\}$ **then**
8: $\quad \rho_{serv_i} \leftarrow \min\left\{\rho^*_i, \rho^{max}_{serv_i}\right\}$

Building the hierarchy. Algorithm 2 presents how to build a hierarchy, it proceeds as follows. We first try to give as many nodes as possible to the servers (line 4 to 7), and we try to build a hierarchy on top of those servers with the remaining nodes (line 8 to 24). Whenever building a hierarchy fails, we reduce the number of nodes available for the servers (line 24, note that we can use a binary search to reduce the complexity, instead of decreasing by one the number of available nodes). Hierarchy construction may fail for several reasons: no more nodes are available for the agents (line 10), ($\mathcal{L}_1$) has no solution (line 12), or only *chains* of agents have been built, *i.e.,* each new agent has only one child (line 20). If a level contains agents with only one child, those nodes are set as available for the next level, as having chains of agents in a hierarchy is useless (line 23). Finally, either we return a hierarchy if we found one, or we return a hierarchy with only one child of each type $i \in \mathcal{R}$, as this means that the limiting factor is the hierarchy of agents. Thus, only one server of each type of service is enough, and we cannot do better than having only one agent.

Correcting the throughput. Once the hierarchy has been computed, we need to correct the throughput for services that were limited by the agents. Indeed, the throughput computed using ($\mathcal{L}_2$) may be too restrictive for some services. The values obtained implied that we had an equal ratio between obtained throughput over requested throughput for all services, which may not be the case if a service requiring lots of computation is deployed alongside a service requiring very few computation. Hence, once the hierarchy is created, we need to compute what is really the throughput that can be obtained for each service on the hierarchy. To do so, we simply use our agent model, with fixed values for ρ_{serv_i} for all $i \in \mathcal{R}$ such that the throughput of i is not limited by the agents, and we try to maximize the values of ρ_{serv_i} for all services that are limited by the agents.

Algorithm 2. Build hierarchy

$N \leftarrow |V| - 1$ // One node for an agent, $|V| - 1$ for the servers
$Done \leftarrow$ **false**
while $N \geq |\mathcal{R}|$ and not $Done$ **do**
  Use Algorithm 1 to find the server repartition with N nodes
  $nbUsed \leftarrow$ number of nodes used by Algorithm 1
  $M_0 \leftarrow nbUsed$
  Set all variables: $n_i(0)$, $a_j(0)$, $\delta_i^j(0)$, $Chld_i^j(0)$ and $c_l^j(0)$
  $k \leftarrow 1$
  $M_k \leftarrow |V| - nbUsed$
  while $M_k > 0$ and not $Done$ **do**
    Compute level k using integer program $(\mathcal{L}_1)$
    if level k could not be built (*i.e.,* $(\mathcal{L}_1)$ failed) **then**
      break
    $nbChains \leftarrow$ count number of agents having only 1 child
    $availNodes \leftarrow M_k$
    $M_k \leftarrow \sum_{j=1}^{M_k} a_j(k)$ // Get the real number of agents
    if $M_k == 1$ **then**
      $Done \leftarrow$ **true** // We attained the root of the hierarchy
      break
    if $nbChains == M_{k-1}$ **then**
      break// This means we added 1 agent over each element at level $k-1$
    $k \leftarrow k + 1$
    $M_k \leftarrow availNodes - M_{k-1} + nbChains$
  $N \leftarrow nbUsed - 1$
if $Done$ **then**
  return the hierarchy built with $(\mathcal{L}_1)$ without chains of agents
else
  return a star graph with one agent and one server of each type $i \in \mathcal{R}$

Extending the model to heterogeneous machines. The model and the algorithms can easily be extended to support the case where each machine has a different computing power w_j, but are still connected with the same bandwidth B. Indeed, we only need to replace w by w_j in all the previous agents' equations, replace Equation (1) by $\frac{w_{app_i}+|\mathcal{S}_i|.w_{pre_i}}{\sum_{j\in\mathcal{S}_i} w_j}$ (with $\mathcal{S}_i$ the set of servers of type i), and modify Algorithm 1 so as to take into account the power of the nodes (for example by sorting the nodes by increasing power) to be able to deal with heterogeneous machines interconnected with a homogeneous network. Note that in this model Remark 1 is no longer relevant.

5 Experimental Results

In order to validate our model, we conducted experiments with DIET on the French experimental testbed Grid'5000 [12]. After a phase of benchmarking for the DIET elements, the services (`dgemm` [13] and computation of the Fibonacci

Table 1. Comparison between theoretical and experimental throughput (number of requests per second)

No. nodes	Services	Theoretical	Experimental			Relative Error
			Mean	Median	Std. Dev.	
3	dgemm	273.0	278.0	278	4.4	1.87%
	Fibonacci	238.3	242.5	242	11.2	1.76%
5	dgemm	534.8	543.2	544	6.1	1.58%
	Fibonacci	470.1	476.1	477	10.7	1.26%
10	dgemm	1027.8	984.9	995	49.5	4.17%
	Fibonacci	915.4	912.9	922	52.1	0.26%
20	dgemm	1699.1	1624.4	1666	114.7	4.39%
	Fibonacci	1738.6	1699.0	1735	114.0	2.28%

number using a naive algorithm) and the platform, we generated hierarchies for a number of nodes ranging from 3 to 50 (even though the algorithm is based on an ILP, it took only a few seconds to generate all the hierarchies). Our goal here is to stress DIET, so we use relatively small services: `dgemm` on 100×100 matrices and computing the Fibonacci number for $n = 30$. With more than 20 nodes the generated hierarchy was always the same as with 20 nodes. We ran as many clients as necessary to obtain the best throughput at the same time for all services. From 3 to 10 nodes, the hierarchy has only one agent, for 20 nodes it has 2 levels of agents, with 3 agents on the second level. Results are given in Table 1. As can be seen, the experimental results closely follow the theoretical predictions. Due to lack of space we do not present all the conducted experiments[1], more results can be found in our research report [14].

6 Conclusion

In this paper we presented a computation and communication model for hierarchical middleware, when several services are available in the middleware. We proposed an algorithm to find a hierarchy that gives the best obtained throughput to requested throughput ratio for all services. The algorithm uses a bottom-up approach, and is based on an ILP to successively determine levels of the hierarchy. Our experiments on a real middleware, DIET, show that the obtained throughput closely follows what our model predicts.

As future works, we intend to run experiments on larger platforms, with "bigger" services. Deployment on homogeneous machines is only the first step that allowed us to validate our model, we intend to extend our model and algorithms to fully heterogeneous platforms.

[1] Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`)

References

1. Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T., Matsuoka, S.: Ninf-g: A reference implementation of RPC-based programming middleware for grid computing. Journal of Grid Computing 1(1), 41–51 (2003)
2. Casanova, H., Dongarra, J.: Netsolve: a network server for solving computational science problems. In: Supercomputing 1996: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM), Washington, DC, USA, p. 40. IEEE Computer Society, Los Alamitos (1996)
3. Morrison, J.P., Clayton, B., Power, D.A., Patil, A.: Webcom-G: grid enabled metacomputing. Neural, Parallel Sci. Comput. 12(3), 419–438 (2004)
4. Caron, E., Desprez, F.: DIET: A scalable toolbox to build network enabled servers on the grid. International Journal of High Performance Computing Applications 20(3), 335–352 (2006)
5. Amar, A., Bolze, R., Caniou, Y., Caron, E., Depardon, B., Gay, J.S., Le Mahec, G., Loureiro, D.: Tunable scheduling in a GridRPC framework. Concurrency and Computation: Practice and Experience 20(9), 1051–1069 (2008)
6. Flissi, A., Merle, P.: A generic deployment framework for grid computing and distributed applications. In: Proceedings of the 2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA 2006). LNCS, vol. 4279, pp. 1402–1411. Springer, Heidelberg (November 2006)
7. Lacour, S., Pérez, C., Priol, T.: Generic application description model: Toward automatic deployment of applications on computational grids. In: 6th IEEE/ACM International Workshop on Grid Computing (Grid 2005), Seattle, WA, USA. Springer, Heidelberg (November 2005)
8. Broto, L., Hagimont, D., Stolf, P., Depalma, N., Temate, S.: Autonomic management policy specification in tune. In: SAC 2008: Proceedings of the 2008 ACM Symposium on Applied Computing, pp. 1658–1663. ACM, New York (2008)
9. Caron, E., Chouhan, P.K., Dail, H.: GoDiet: A deployment tool for distributed middleware on grid'5000. In: IEEE (ed.) EXPGRID Workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In Conjunction with HPDC-15, Paris, France, June 19, pp. 1–8 (2006)
10. Caron, E., Chouhan, P., Legrand, A.: Automatic deployment for hierarchical network enabled servers. In: Proceedings of 18th International Parallel and Distributed Processing Symposium, p. 109 (April 2004)
11. Chouhan, P.K., Dail, H., Caron, E., Vivien, F.: Automatic middleware deployment planning on clusters. Int. J. High Perform. Comput. Appl. 20(4), 517–530 (2006)
12. Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Irena, T.: Grid'5000: a large scale and highly reconfigurable experimental grid testbed. International Journal of High Performance Computing Applications 20(4), 481–494 (2006)
13. Dongarra, J., et al.: Basic linear algebra subprograms technical forum standard. International Journal of High Performance Applications and Supercomputing 16(1), 1–111 (2002)
14. Caron, E., Depardon, B., Desprez, F.: Modelization for the Deployment of a Hierarchical Middleware on a Homogeneous Platform. Technical report, Institut National de Recherche en Informatique et en Automatique, INRIA (2010)