Modelization for the Deployment of a Hierarchical Middleware on a Homogeneous Platform

Eddy Caron, Benjamin Depardon, Frédéric Desprez

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Modelization for the Deployment of a Hierarchical Middleware on a Homogeneous Platform

Eddy Caron ,
Benjamin Depardon ,
Frédéric Desprez

University of Lyon. LIP Laboratory. UMR CNRS - ENS Lyon - INRIA - UCBL 5668. France.

February 2010

Research Report N° RRLIP2010-10

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`




Modelization for the Deployment of a Hierarchical Middleware on a Homogeneous Platform

Eddy Caron , Benjamin Depardon , Frédéric Desprez

University of Lyon. LIP Laboratory. UMR CNRS - ENS Lyon - INRIA - UCBL 5668. France.

February 2010

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 middlewares. 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, Grid.

Résumé

De nos jours, l'accès à des ressources distribuées peut être réalisé aisément en utilisant un *intergiciel* se basant sur une approche client/serveur. Différentes architectures existent pour de tels intergiciels. Ceux passant le mieux à l'échelle utilisent une hiérarchie d'agents. Déterminer quelle est la meilleure hiérarchie, c'est à dire celle qui fournira le meilleur débit au niveau des services, n'est pas une tâche aisée.

Nous proposons tout d'abord un modèle de calcul et de communication pour de tels intergiciels hiérarchiques. Notre modèle prend en compte le déploiement de plusieurs services au sein de la hiérarchie. Puis, en nous basant sur le modèle, nous proposons un algorithme pour construire automatiquement la hiérarchie. L'algorithme vise à offrir aux utilisateurs le meilleur ratio entre le débit demandé, et le débit fourni, tout en utilisant le moins de ressources possible. Enfin, nous comparons notre modèle à des résultats expérimentaux obtenus avec l'intergiciel de grille DIET.

Mots-clés: Intergiciel hiérarchique, Déploiement, Modélisation, Grille.

1 Introduction

Using distributed resources to solve large problems ranging from numerical simulations to life science is nowadays a common practice [3, 15]. Several approaches exist for porting these applications to a distributed environment; examples include classic message-passing, batch processing, web portals and GridRPC systems [18]. 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 middlewares [6] 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 power to solve the requests. Some of the middlewares only rely on basic hierarchies of elements, a star graph, such as Ninf-G [19] and NetSolve [2, 10, 21]. Others, in order to divide the load at the agents level, can have a more complicated hierarchy shape: WebCom-G [17] and DIET [1, 9]. In this latter case, a problem arises: what is the *best* shape for the hierarchy?

Modelization of middlewares behavior, and more specifically their needs in terms of computations and communications at the agents and servers levels can be of a great help when deploying the middleware on a computing platform. Indeed, the administrator needs to choose how many nodes must be allocated to the servers, and how many agents have to be present to support the load required by the clients. Using as many nodes as possible, may not be the best solution: firstly it may lead to using more resources than necessary; and secondly this can degrade the overall performances. The literature do not provide much papers on the modelization and evaluation of distributed middleware. In [20], Tanaka *et al.* present a performance evaluation of Ninf-G, however, no theoretical model is given. In [7, 12, 11] the authors present a model for hierarchical middlewares, and algorithms to deploy a hierarchy of schedulers on clusters and grid environments. They also compare the model with the DIET middleware. However, a severe limitation in these latter 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.

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 [14], ADAGE [16], TUNe [5], and GODIET [8]. 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 [7, 12] 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. We then compare the behavior of the DIET middleware with the model in Section 5. Then, we present, in Sections 6 and 7 the platform and DIET elements benchmarks necessary for the experiments. Finally, we compare the theoretical results with experimental results in Section 8, 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 (see Figure 1. We do not take into account contentions in the network.

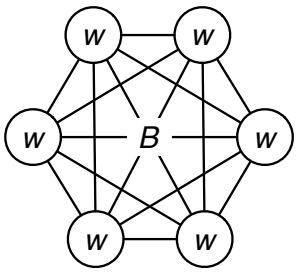

Figure 1: Homogeneous platform

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 power for scheduling and solving, 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 3.1 *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}.*

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

Proof: A completed request has, by definition, completed both the scheduling and the service request phases, whatever the kind of request $i \in \mathcal{R}$.

Case 1: $\rho_{sched_i} \geq \rho_{serv_i}$. In this case, requests are sent to the servers at least as fast as they can be processed by the servers, so the overall rate is limited by ρ_{serv_i}.

Case 2: $\rho_{sched_i} < \rho_{serv_i}$. In this case, the servers process the requests faster than they arrive. The overall throughput is thus limited by ρ_{sched_i}. □

Lemma 3.2 *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.

3.2.1 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 requests. 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.

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

Thus, the service throughput for requests of type i is given by the following formula, note that $\rho_{serv_i}^{comp}$ is the service throughput without taking into account communications:

$$\rho_{serv_i}^{comp} = \frac{w}{w_{pre_i} + \frac{w_{app_i}}{n_i}} \tag{2}$$

Lemma 3.3 *$\rho_{serv_i}^{comp}$ tends to $\frac{w}{w_{pre_i}}$ as n_i grows larger.*

Server communication model. A server of type i needs, for each request, to receive the request, and then to reply. Hence Equations 3 and 4 represent respectively the time to receive one request of type i, and the time to send the reply to its parent.

$$T_{recv_i}^{server} = \frac{m_{req_i}}{B} \tag{3}$$

$$T_{send_i}^{server} = \frac{m_{resp_i}}{B} \tag{4}$$

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.

$$\rho_{serv_i} = \frac{1}{T_{recv_i}^{server} + T_{send_i}^{server} + T_{comp_i}^{server}} \tag{5}$$

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

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

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

$$\rho_{serv_i} = \min\left\{\frac{1}{T_{recv_i}^{server}}, \frac{1}{T_{send_i}^{server}}, \frac{1}{T_{comp_i}^{server}}\right\} \tag{7}$$

3.2.2 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 $Chld_i^j$ be the set of children of A_j having service i in their underlying hierarchy. Also, let δ_i^j be a Boolean variable equal to 1 if and only if A_j has at least one children having service i in its underlying hierarchy. $w_{resp_i}\left(\left|Chld_i^j\right|\right)$ is the amount of computation in $MFlops$ needed to merge the replies of type i from its $\left|Chld_i^j\right|$ 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 8.

$$T_{comp}^{agent_j} = \frac{\sum_{i \in \mathcal{R}} \rho_{serv_i}.\delta_i^j.w_{req_i} + \sum_{i \in \mathcal{R}} \rho_{serv_i}.w_{resp_i} \left(\left|Chld_i^j\right|\right)}{w} \tag{8}$$

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 9 and 10 present the time to receive and send all messages when the servers provide a throughput ρ_{serv_i} for each $i \in \mathcal{R}$.

$$T_{recv}^{agent_j} = \frac{\sum_{i \in \mathcal{R}} \rho_{serv_i}.\delta_i^j.m_{req_i} + \sum_{i \in \mathcal{R}} \rho_{serv_i}.\left|Chld_i^j\right|.m_{resp_i}}{B} \tag{9}$$

$$T_{send}^{agent_j} = \frac{\sum_{i \in \mathcal{R}} \rho_{serv_i}.\delta_i^j.m_{resp_i} + \sum_{i \in \mathcal{R}} \rho_{serv_i}.\left|Chld_i^j\right|.m_{req_i}}{B} \tag{10}$$

We combine (8), (9), and (10) according to the chosen communication / computation model (Equations (5), (6), and (7)).

Lemma 3.4 *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.*

Proof: The bottleneck of such a hierarchy is clearly its root. Whatever the shape of the hierarchy, at its top, the root will have to support *at least* one child of each type of service (all messages have to go through the root). As the time required for an agent grows linearly with the number of children (see (8), (9) and (10)), having only one child of each type of service is the configuration that induces the lowest load on an agent. □

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 Section4.3 we present the whole algorithm.

4.1 Servers repartition

Our goal is to obtain for all services $i \in \mathcal{R}$ the same ratio $\frac{\rho_{serv_i}}{\rho_i^*}$. 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.

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

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: $i \leftarrow$ first service in S
5: Assign one more node to i, and compute the new ρ_{serv_i}
6: $n \leftarrow n + 1$
7: **if** $\rho_{serv_i} \geq \rho_i^*$ **then**
8: $\rho_{serv_i} \leftarrow \rho_i^*$
9: $S \leftarrow S - \{i\}$
10: $S \leftarrow$ Sort services by increasing $\frac{\rho_{serv_i}}{\rho_i^*}$
11: **until** $n = N$ **or** $S = \emptyset$
12: **return** n

is enough to support all the children of the previous level. In order to build each level, we make use of a mixed integer linear program (MILP): ($\mathcal{L}_1$).

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 linear 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 children, 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 MILP ($\mathcal{L}_1$). 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 4.1 *In order to improve the converge time to an optimal solution for linear program ($\mathcal{L}_1$), we can add the following constraint:*

$$a_1(k) \geq a_2(k) \cdots \geq a_{M_k}(k) \tag{11}$$

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

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 (8), (9) and (10), 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 3.4). Hence, the maximum attainable throughput for each service, when all service receive the same served to required throughput ratio, from the agents' point of view is given by linear 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.

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

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 effectively 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. We use linear program $\mathcal{L}_3$ and Algorithm 3 for this purpose.

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 linear 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 $i \in \mathcal{R}$

Algorithm 3 Correct Throughput

$\mathcal{R}_{agLim} \leftarrow i \in \mathcal{R}$ such that service i is "agent limited"
$Ag \leftarrow$ set of agents per level
while $\mathcal{R}_{agLim} \neq \emptyset$ **do**
  $\mu, \{\rho_i^{max}\} \leftarrow$ Solve linear program ($\mathcal{L}_3$)
  Find $i \in \mathcal{R}_{agLim}$ such that $\mu = \frac{\rho_i^{max}}{\rho_i^*}$
  $\rho_{serv_i} \leftarrow \rho_i^{max}$
  $\mathcal{R}_{agLim} \leftarrow \mathcal{R}_{agLim} - \{i\}$

Maximize μ
Subject to

$$
\begin{cases}
(1) \quad \forall i \in \mathcal{R}_{agLim} & \mu \in [0,1], \mu \leq \dfrac{\rho_i^{max}}{\rho_i^*} \text{ and } 0 \leq \rho_i^{max} \leq \rho_{serv_i} \\
(2) \quad \forall k, \forall j \in Ag[k] & \displaystyle\sum_{i \in \mathcal{R}-\mathcal{R}agLim} \rho_{serv_i} . \frac{\delta_i^j(k).w_{req_i} + w_{resp_i}\left(\left|Chld_i^j(k)\right|\right)}{w} + \\
& \displaystyle\sum_{i \in \mathcal{R}agLim} \rho_i^{max} . \frac{\delta_i^j(k).w_{req_i} + w_{resp_i}\left(\left|Chld_i^j(k)\right|\right)}{w} + \\
& \displaystyle\sum_{i \in \mathcal{R}-\mathcal{R}agLim} \rho_{serv_i} . \frac{\delta_i^j(k).m_{req_i} + \left|Chld_i^j(k)\right| .m_{resp_i}}{B} + \\
& \displaystyle\sum_{i \in \mathcal{R}agLim} \rho_i^{max} . \frac{\delta_i^j(k).m_{req_i} + \left|Chld_i^j(k)\right| .m_{resp_i}}{B} + \\
& \displaystyle\sum_{i \in \mathcal{R}-\mathcal{R}agLim} \rho_{serv_i} . \frac{\delta_i^j(k).m_{resp_i} + \left|Chld_i^j(k)\right| .m_{req_i}}{B} + \\
& \displaystyle\sum_{i \in \mathcal{R}agLim} \rho_i^{max} . \frac{\delta_i^j(k).m_{resp_i} + \left|Chld_i^j(k)\right| .m_{req_i}}{B} \leq 1
\end{cases}
\qquad (\mathcal{L}_3)
$$

In ($\mathcal{L}_3$), Equation (1) states that μ is the minimum of all ratios, (2) states that value of ρ_i^{max} cannot be greater than the throughput that is offered at the server level. The following equations ensure that bandwidth and computing power aren't violated.

4.4 A few discussion about this model

Reducing complexity The problem of using an MILP representation for our problem, is that the time required to compute the solution may grow exponentially with the number of nodes. Hence, if dealing with large number of nodes, we can reduce the time spent in searching the hierarchy of agents by first constructing a few homogeneous sub-hierarchies, *i.e.,* hierarchies within which only one service is present, using d-ary trees (maybe not complete d-ary trees, but only some of the lower levels, and keeping only nodes that are fully loaded, *i.e.,* where the degree d is attained). Doing so will reduce the number of levels we will need to build using MILP, and should give a good solution.

Messages without a fixed size In this model, we supposed that the size of the reply messages was fixed, *i.e.,* that whatever the number of servers, the hierarchy always returns only one choice to the client. We could also modify our model in the case where the size of return messages in the hierarchy grows linearly with the number of servers found so far. The middleware could also return to the client the full list of servers. Hence, at each level of the hierarchy the reply message would grow. This can easily be taken into account in our model. Let $Cpt_i^j(k)$ be the number of servers of type i in the hierarchy under element j at level k. For level $k = 0$ we set $Cpt_i^j(0) = 1$ if node j is a server of type i, otherwise we set $Cpt_i^j(0) = 0$. For $k > 0, i \in \mathcal{R}, 1 \leq j \leq M_k$ we have $Cpt_i^j(k) = \sum_{l=1}^{M_{k-1}} Cpt_i^l(k-1) \times c_l^j(k)$.

Then, we would need to change the equations for the sending and receiving time at the agent level with the following ones:

$$
\frac{\sum_{i \in \mathcal{R}} \rho_{serv_i} . \delta_i^j(k) . m_{req_i} + \sum_{i \in \mathcal{R}} \rho_{serv_i} . m_{resp_i} . \sum_{l=1}^{M_{k-1}} c_l^j(k) . Cpt_i^l(k-1)}{B} \qquad (12)
$$

$$\frac{\sum_{i\in\mathcal{R}} \rho_{serv_i}.m_{resp_i}.\sum_{l=1}^{M_{k-1}} c_l^j(k).Cpt_i^l(k-1) + \sum_{i\in\mathcal{R}} \rho_{serv_i}.\left|Chld_i^j(k)\right|.m_{req_i}}{B} \tag{13}$$

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 4.1 is no longer relevant.

5 Comparing the model with the behavior of Diet

Diet follows the model presented in Section 3: whenever a requests arrives at an agent, it is forwarded only to its underlying children that declared having this service in their underlying hierarchy. We ran some experiments to assess the fact that an agent forwards only to the rightful children a request.

We deployed two kinds of hierarchies, and used TAU [**?**] to retreive the number of instructions executed per request per LA. The first one has 1 MA, 2 LA and under each LA n SeD, but only 1 service is present under each LA, see Figure 2. The second hierachy has 1 MA, 1 LA and n SeD per service (we used 2 services), note that in this case, the LA's degree is twice as large as the degree of each LA in the first case, see Figure 3. We then made 10 requests per service.

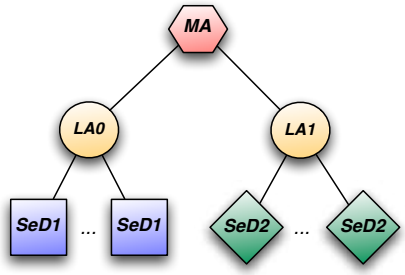

Figure 2: Platform 1: 1 MA, 2 LA, n SeD, 2 services

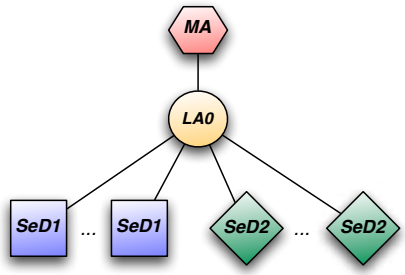

Figure 3: Platform 2: 1 MA, 1 LA, $2n$ SeD, 2 services

Figures 4 and 5 respectively show the total number of instructions per request for each LA, and the total number of cycles per request for each LA. As can be seen, the work required to deal

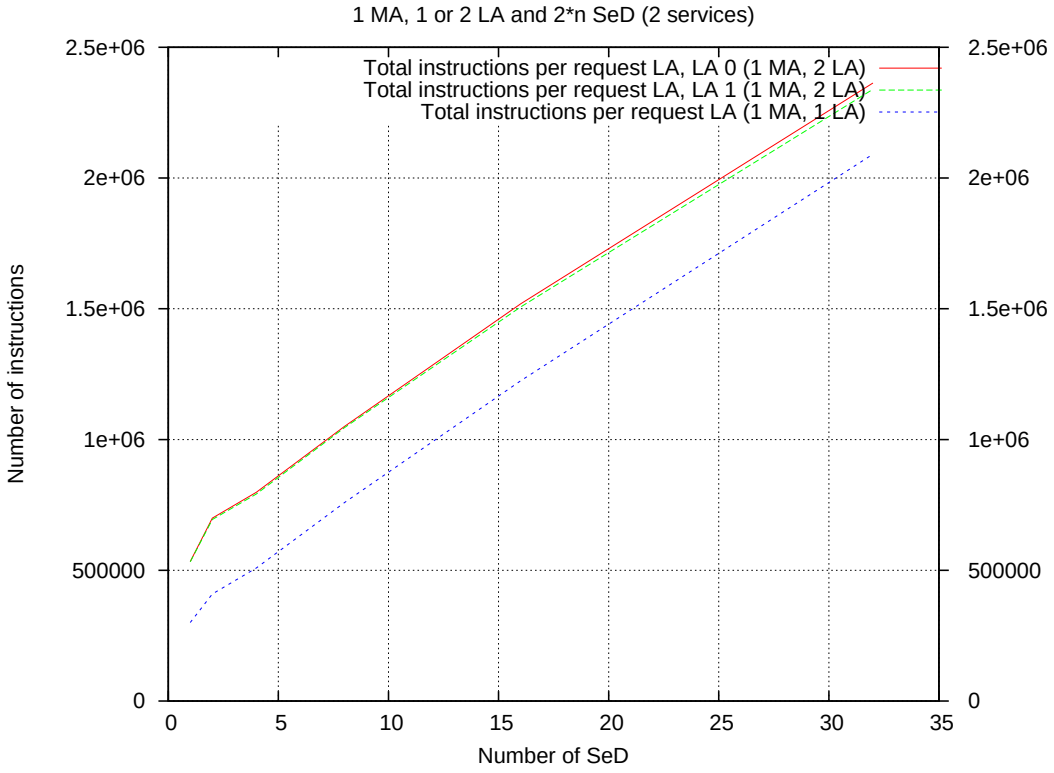

Figure 4: Total number of instructions per request for each LA

with one request is almost the same for each LA, whatever the platform used. If DIET was not following the model depicted in Section 3.2, but instead would forward the requests to all children, the work required for the LA of the second hierarchy should have been twice as much as for the LA of the first hierarchy (as the degree is twice as large).

In platform 1, each LA received only 10 requests, whereas in platform 2, the LA received 20 requests.

6 Benchmarking the platform

6.1 Platform

We used a 79-nodes cluster present in the Grid'5000 experimental platform [4]: the cluster Sagittaire from the Lyon site. Each node has an AMD Opteron 250 CPU at 2.4GHz, with 1MB of cache and 2GB of memory. All those nodes are connected on a Gigabit Ethernet network supported by an Extreme Networks Blackdiamond 8810. Hence, for this platform our fully homogeneous, fully connected platform assumption holds (we ran bandwidth tests using iPerf[1] to confirm that there really was a Gigabit network between any two machines).

We measured the computing capacity of the nodes with HPL[2] (along with the Altas[3] version of BLAS) we found a mean value of 3.249 Gflops.

[1] http://sourceforge.net/projects/iperf/
[2] http://www.netlib.org/benchmark/hpl/
[3] http://math-atlas.sourceforge.net/

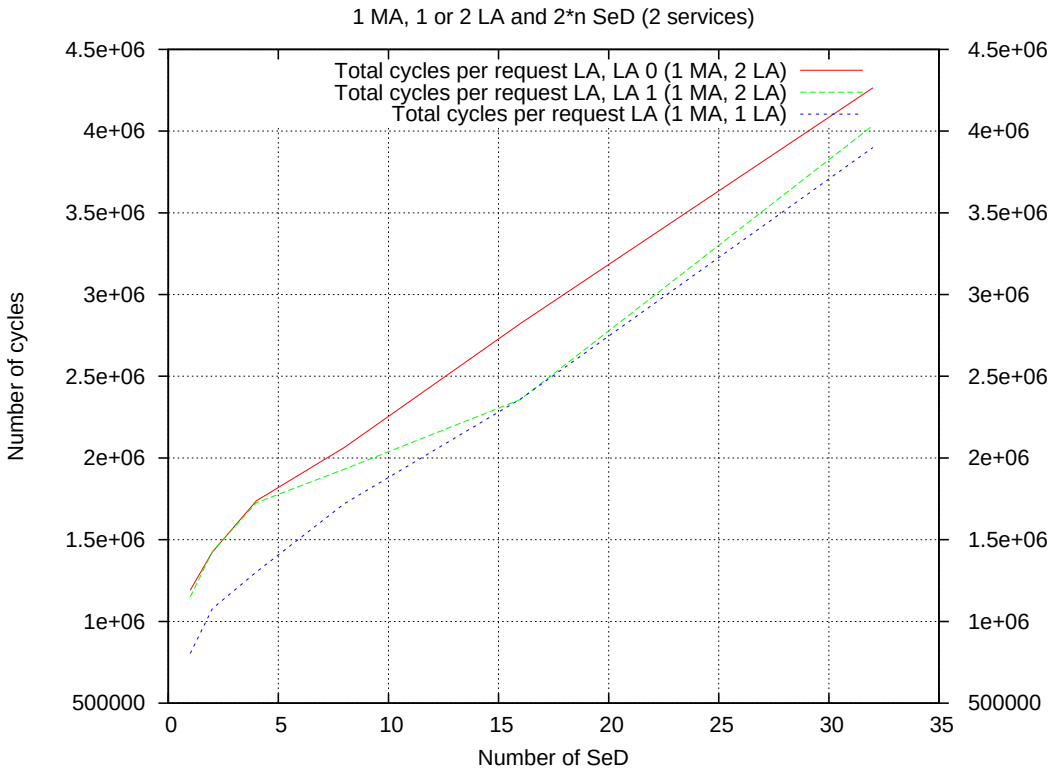

Figure 5: Total number of cycles per request for each LA

6.2 Impact of the bandwidth on the model

The bandwidth the message receive is not necessarily the maximum bandwidth attainable on the cluster. Indeed, in order to have the full links capacity, one has to transmit large messages. In our case messages in the hierarchy are quite small (a few kilobits), hence we need to determine the bandwidth received by the messages. We used NWS[4] to determine the bandwidth when sending messages of different size. Figures 6, 7 and 8 show the impact of the bandwidth on the model: the red and green line present respectively the theoretical `dgemm` and Fibonacci throughput, and the red and green points represent respectively the experimental throughput for `dgemm` and Fibonacci during the experiment.

Figure 6 presents the results obtained when using a bandwidth of $1033 Mb.s^{-1}$ in the model: maximum bandwidth measured on the cluster. Figure 7 presents the results for a bandwidth of $53 Mb.s^{-1}$ bandwidth measured with NWS for $1kb$ messages. And finally, Figure 8 presents the results for a bandwidth of $186.7 Mb.s^{-1}$: bandwidth measured with NWS for $8kb$ messages, *i.e.,* the buffer size just above the messages size (around $5kb$), the minimum buffer size of the system is $4kb$, hence when using messages of about $5kb$, the system buffer increases up to $8kb$. As can be seen, the results with $186.7 Mb.s^{-1}$ are the most accurate.

[4] http://nws.cs.ucsb.edu/ewiki/

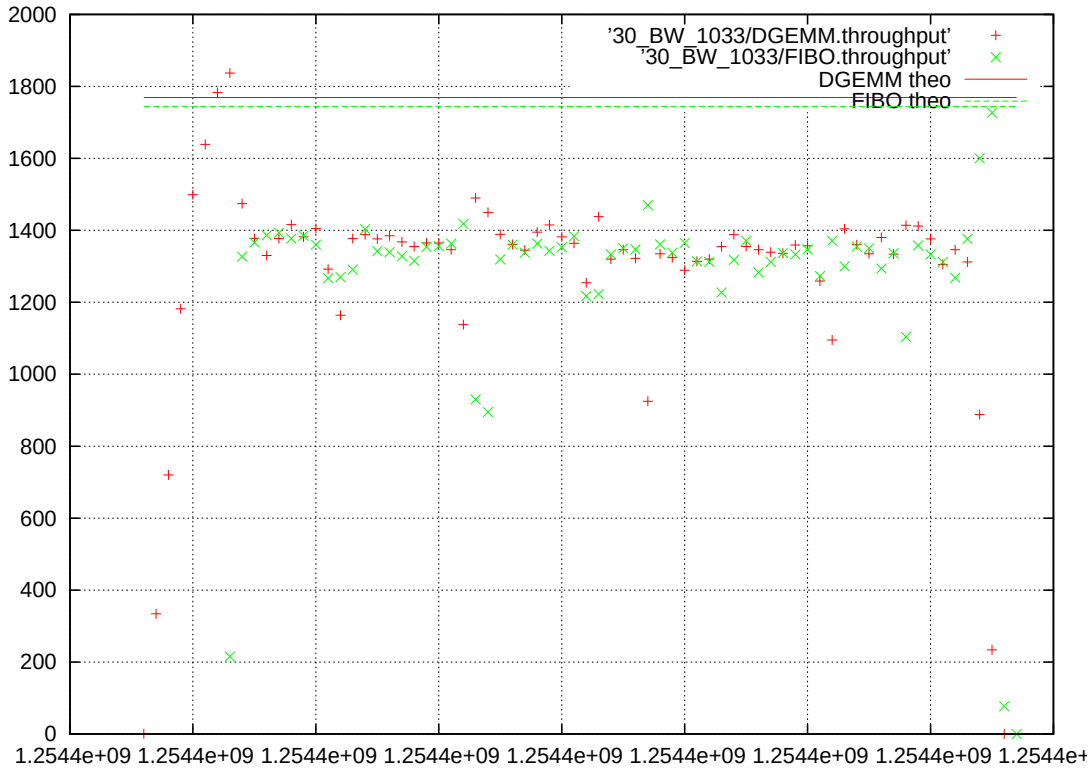

Figure 6: Throughputs obtained when modeling with $1033 Mb.s^{-1}$

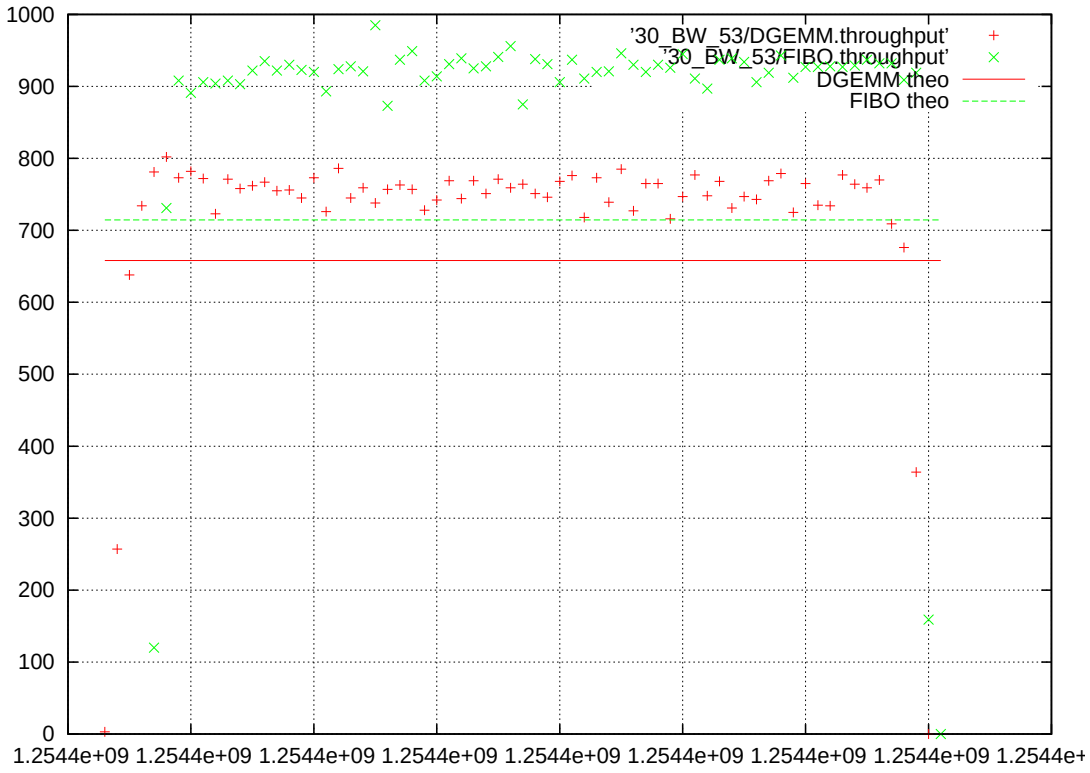

Figure 7: Throughputs obtained when modeling with $53Mb.s^{-1}$

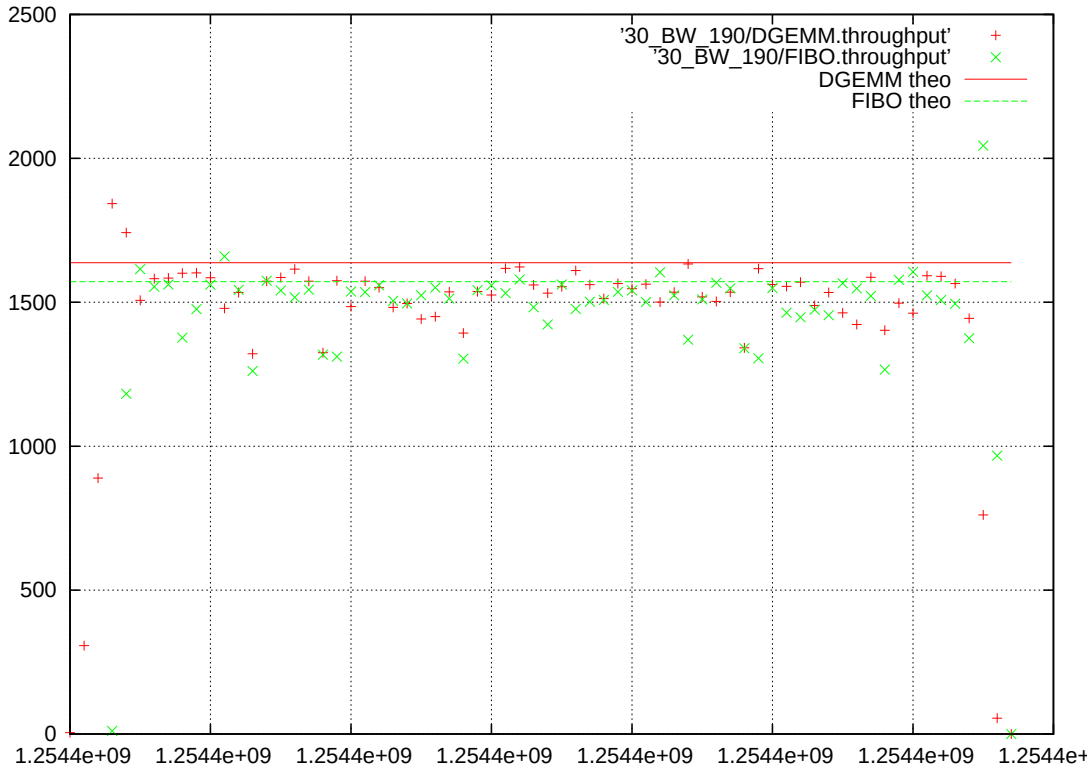

Figure 8: Throughputs obtained when modeling with $186.7 Mb.s^{-1}$

7 Benchmarking the Diet elements

Throughout the rest of the paper, we will denote by `dgemm` x the fact of calling the `dgemm` service on a $x \times x$ matrix, and Fibonacci x the fact of calling the Fibonacci service for computing the Fibonacci number for $n = x$.

7.1 Computation/communication models versus experiments

Figures 9 and 10 present the comparison between the theoretical model of communication/computation and the experimental results. As can be seen, the serial model, *i.e.,* send or receive or compute, single port is the one that best matches the experimental results.

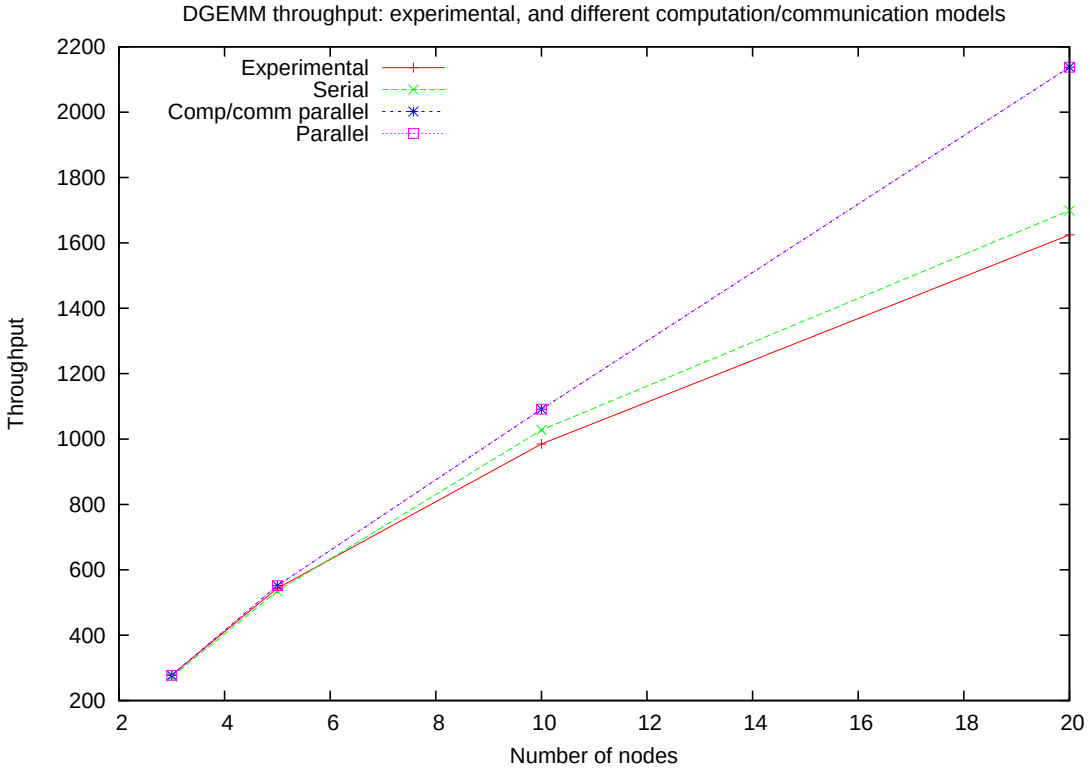

Figure 9: `dgemm` experimental, and theoretical throughput with the different models.

7.2 Messages

We used `tcpdump` and `wireshark` to analyze the messages sent between the agent and the SeD. Figures 11 and 12 present the messages exchanged when requesting a service (respectively for DGEMM and Fibonacci). Table 1 presents the messages size for both `dgemm` and Fibonacci services.

Service	m_{req_i}	m_{resp_i}
`dgemm`	5.136×10^{-3}	5.456×10^{-3}
Fibonacci	4.176×10^{-3}	5.456×10^{-3}

Table 1: Messages size in Mb for `dgemm` and Fibonacci services.

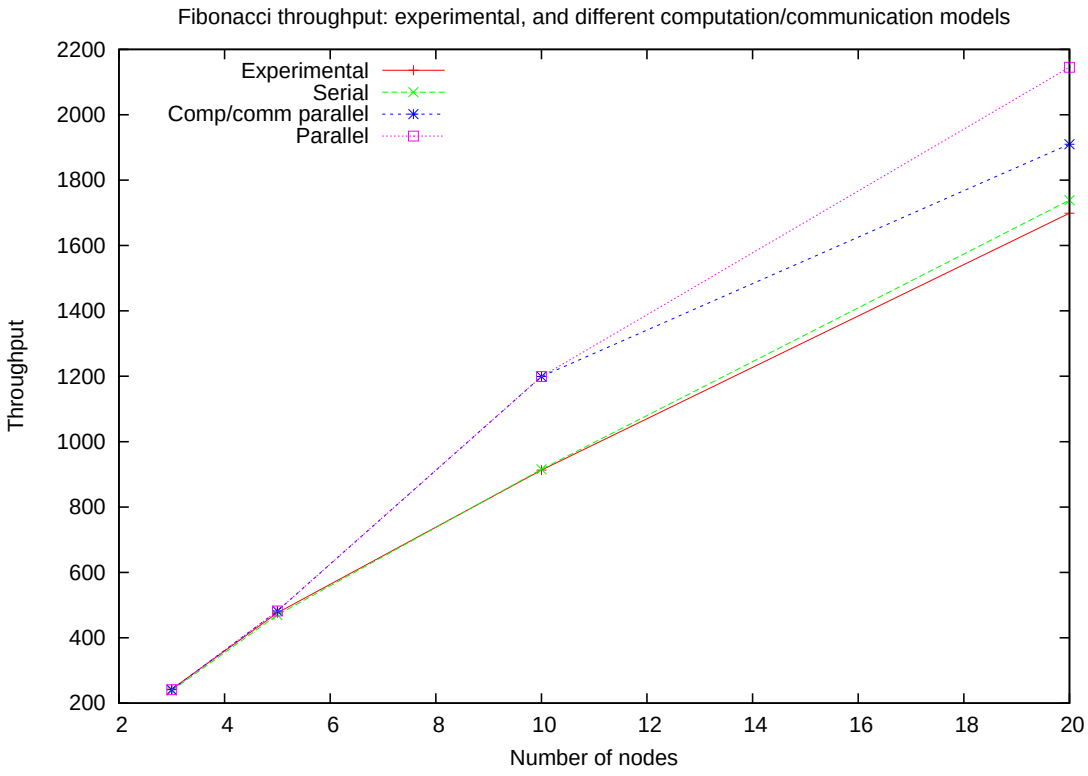

Figure 10: Fibonacci experimental, and theoretical throughput with the different models.

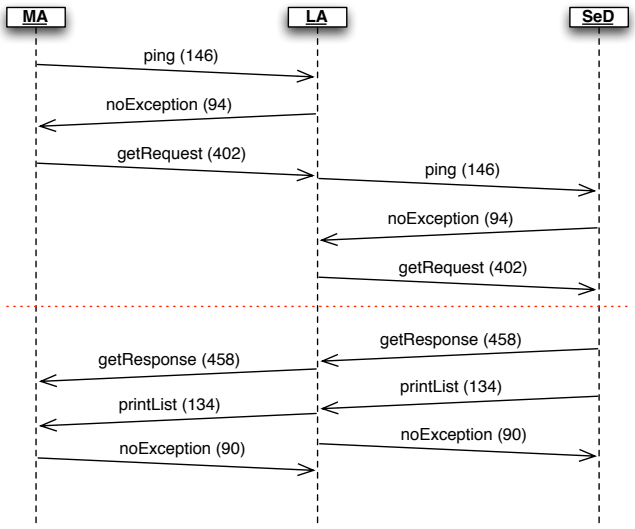

Figure 11: Messages exchanged during a request for service `dgemm` on a hierarchy composed of 1 MA, 1 LA and 1 SeD. Numbers in brackets represent the size of the message in bytes.

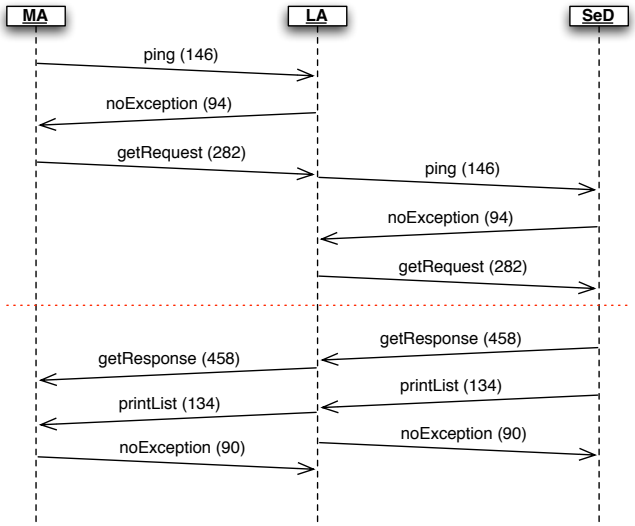

Figure 12: Messages exchanged during a request for service Fibonacci on a hierarchy composed of 1 MA, 1 LA and 1 SeD. Numbers in brackets represent the size of the message in bytes.

7.3 SeD

In order to benchmark the SeDs, we used a two steps approach. The first step consisted in finding what was the required number of clients to load a SeD (*i.e.,* obtaining the maximum throughput, without having too much variations on the throughput value). Then, we deploy a new platform, and launch the number of clients found in the previous step, and determine using Diet statistics, the model parameters model.

We deploy a platform composed of 1 MA, 1 LA and 1 SeD. Then, we run threaded clients to load the platform. A threaded client launches a new thread every second until attaining its total number of allowed thread. We launch a client, then wait for it to run all its thread, and wait 20 seconds more for the throughput to stabilize, then we run a new client on another node. We run enough clients to fully load the platform. Once all the clients are run, we let the system work for 60 seconds before cleaning the platform. The number of threads per client depends on the type of service (for a `dgemm` on a large matrix, a client cannot have too many threads, otherwise it will start to swap).

7.3.1 Determining the necessary number of clients for `dgemm`

`dgemm` 10 we ran 5 clients, each having 40 threads. Figure 13 presents the throughput. The maximum throughput is attained at about 90s. Hence, the maximum number of threads we need is 70 spread on two nodes.

`dgemm` 100 we ran 10 clients, each having 20 threads. Figure 14 presents the throughput. The maximum throughput is attained at about 80s. Hence, the maximum number of threads we need is 40 spread on two nodes.

`dgemm` 500 we ran 40 clients, each having 5 threads. Figure 15 presents the throughput. The maximum throughput is attained at about 70s. Hence, the maximum number of threads we need is 15 spread on 3 nodes.

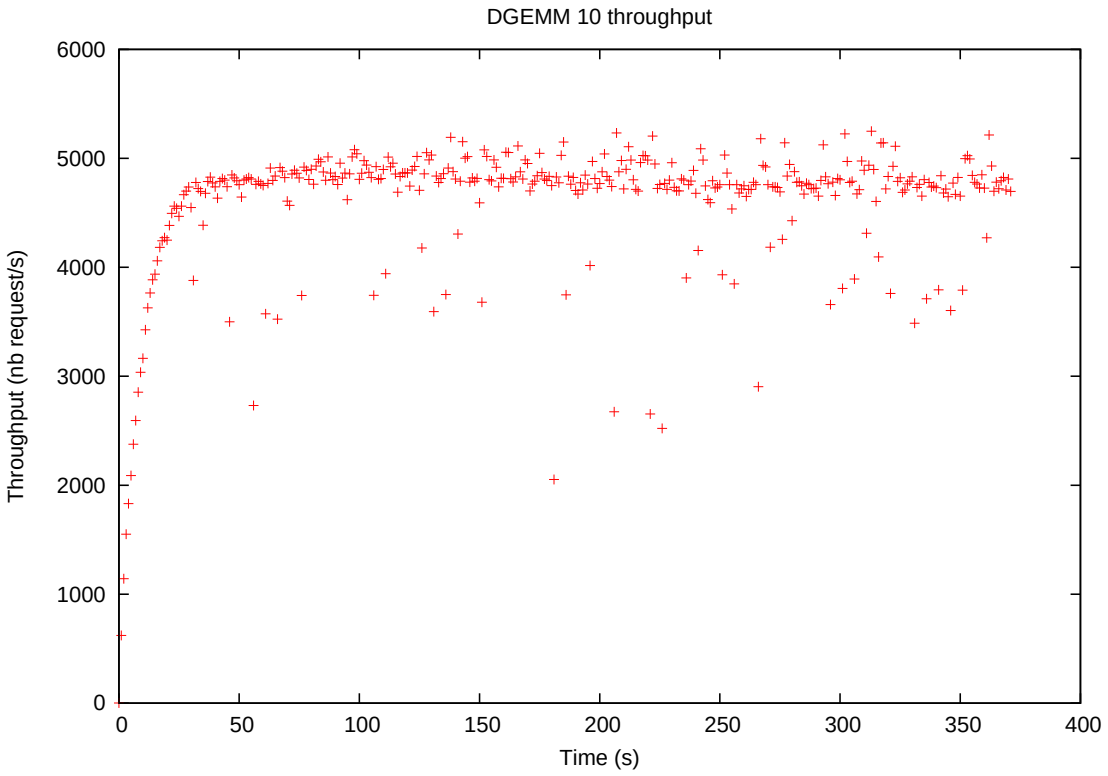

Figure 13: `dgemm` 10 throughput

7.3.2 Determining the necessary number of clients for Fibonacci

FIBO 20 we ran 4 clients, each having 50 threads. Figure 16 presents the throughput. The maximum throughput is attained at about 220s. Hence, the maximum number of threads we need is 170 spread on 5 nodes.

FIBO 30 we ran 4 clients, each having 50 threads. Figure 17 presents the throughput. The maximum throughput is attained at about 50s. Hence, the maximum number of threads we need is 50 on 2 nodes.

FIBO 40 we ran 4 clients, each having 50 threads. Figure 18 presents the throughput. The maximum throughput is attained at about 10s. Hence, the maximum number of threads we need is 10 on 1 nodes.

7.3.3 Benchmarking

We then ran the previously found number of clients for each kind of service, and let them send requests for 60 seconds. During this time, we collected statistics on the time required by the SeD to process a request, and to solve a request. Table 2 and 3 present the mean time for `getRequest` (the time to process a request) and `solve` (the time to solve a request), respectively for `dgemm` and Fibonacci services.

Hence, the values in MFlops presented in tables 4 and 5.

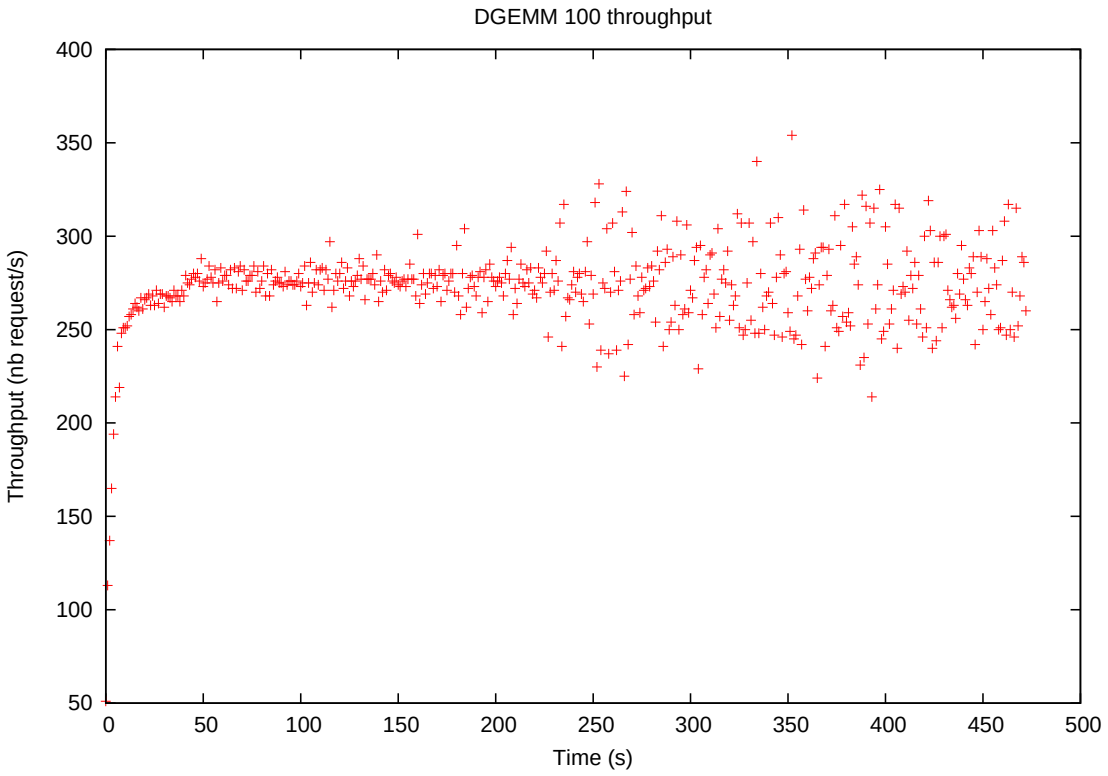

Figure 14: `dgemm` 100 throughput

	Nb requests	`getRequest`	`solve`
`dgemm` 10	286050	0.000415694734846	0.00319375696017
`dgemm` 100	16633	0.000314177963958	0.0585402570871
`dgemm` 500	536	0.000297451642022	0.656850664473

Table 2: `dgemm` mean `getRequest` and `solve` times (in s).

	Nb requests	`getRequest`	`solve`
FIBO 20	305711	0.000401932459236	0.00507951654684
FIBO 30	14477	0.000253283154893	0.165682609844
FIBO 40	115	$2.28923300038e^{-5}$	4.77842594437

Table 3: FIBO mean `getRequest` and `solve` times (in s).

	`getRequest`	`solve`
`dgemm` 10	0.0784808320277	0.602963382785
`dgemm` 100	0.0625602101851	11.6567398347
`dgemm` 500	0.164611652527	363.505383958

Table 4: `dgemm` mean `getRequest` and `solve` required computing power (in MFlops).

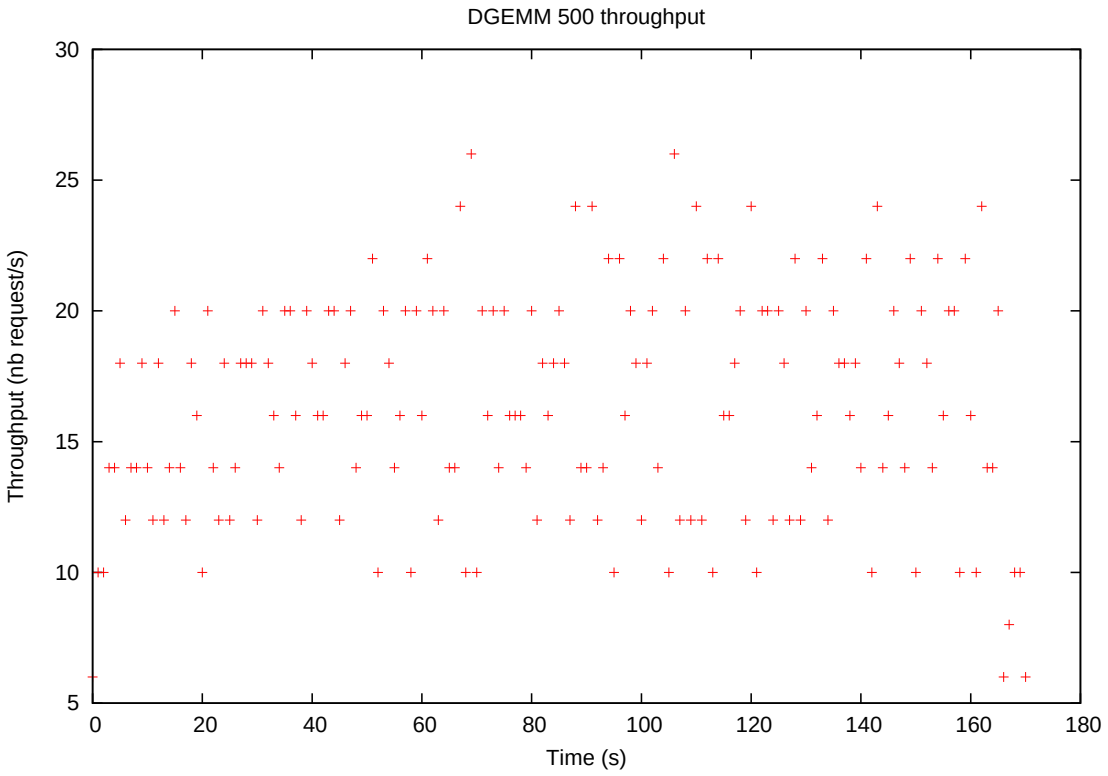

Figure 15: `dgemm` 500 throughput

	`getRequest`	`solve`
FIBO 20	0.0467540134516	0.590864906532
FIBO 30	0.0205522633808	13.4440548822
FIBO 40	0.00812040104002	1695.01029392

Table 5: FIBO mean `getRequest` and `solve` required computing power (in MFlops).

7.4 Agent

We deploy a platform composed of 1 MA, 1 LA and n SeD. Then we run threaded clients to load the platform. We launch all clients, then wait 10s for the clients to finish to run their threads, then we conduct the measurements for 60s. As we aim at loading the agent, we need lots of requests. Hence, we run only small services at the SeD level: the client call a `dgemm` on a 10×10 matrix, or request to solve the Fibonacci number for $x = 20$. The number of threads per client depends on the type of service, and is set to 170 threads on 5 nodes for FIBO 20, and 70 threads on 2 nodes for `dgemm` 10.

Service	w_{req_i}	w_{resp_i}
`dgemm`	0.398057810949015	0.2370227507159
FIBO	0.376124815390629	0.235280539487372

Table 6: Agents parameters

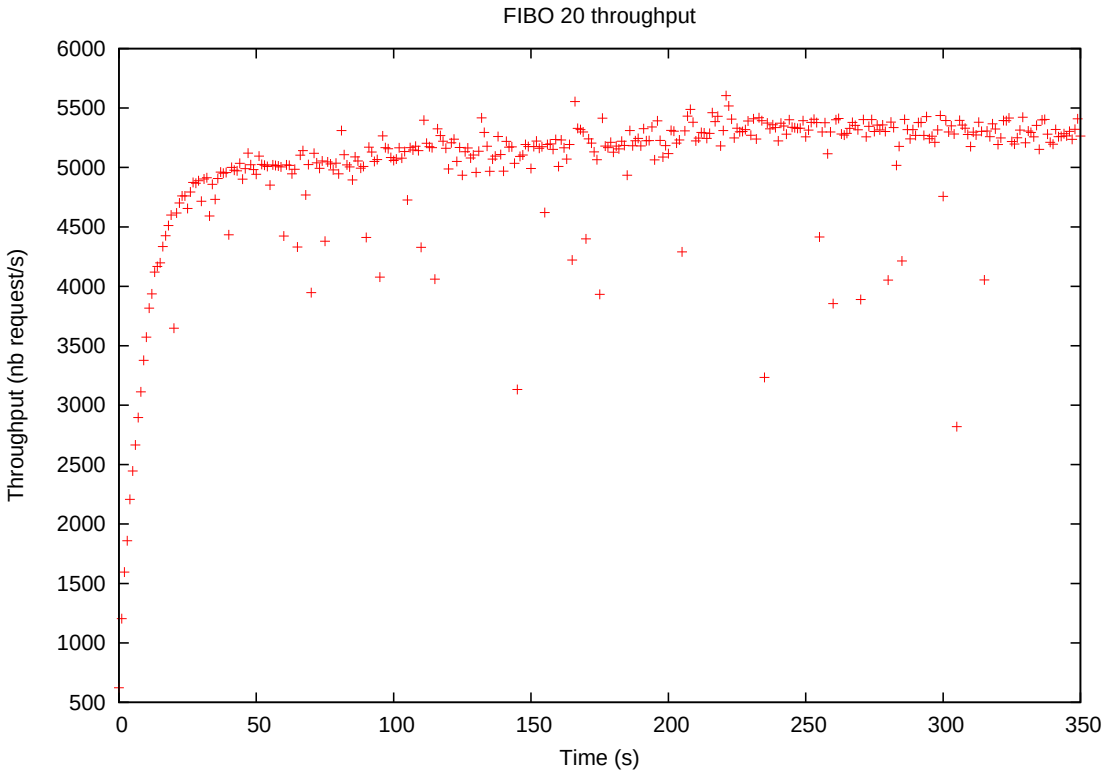

Figure 16: FIBO 20 throughput

Service	w_{req_i}	w_{resp_i}
`dgemm`	0.230450419663851	0.0694153594307296
FIBO	0.235280539487372	0.0676731482022025

Table 7: Agents parameters when removing the communication time

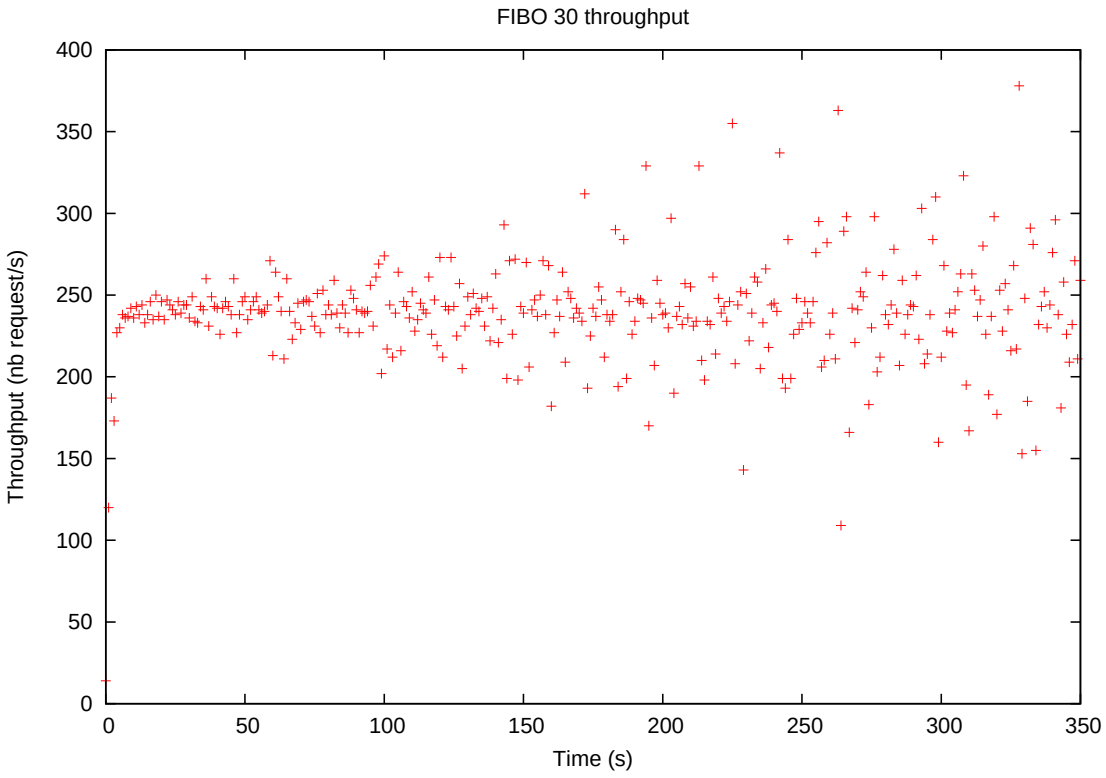

Figure 17: FIBO 30 throughput

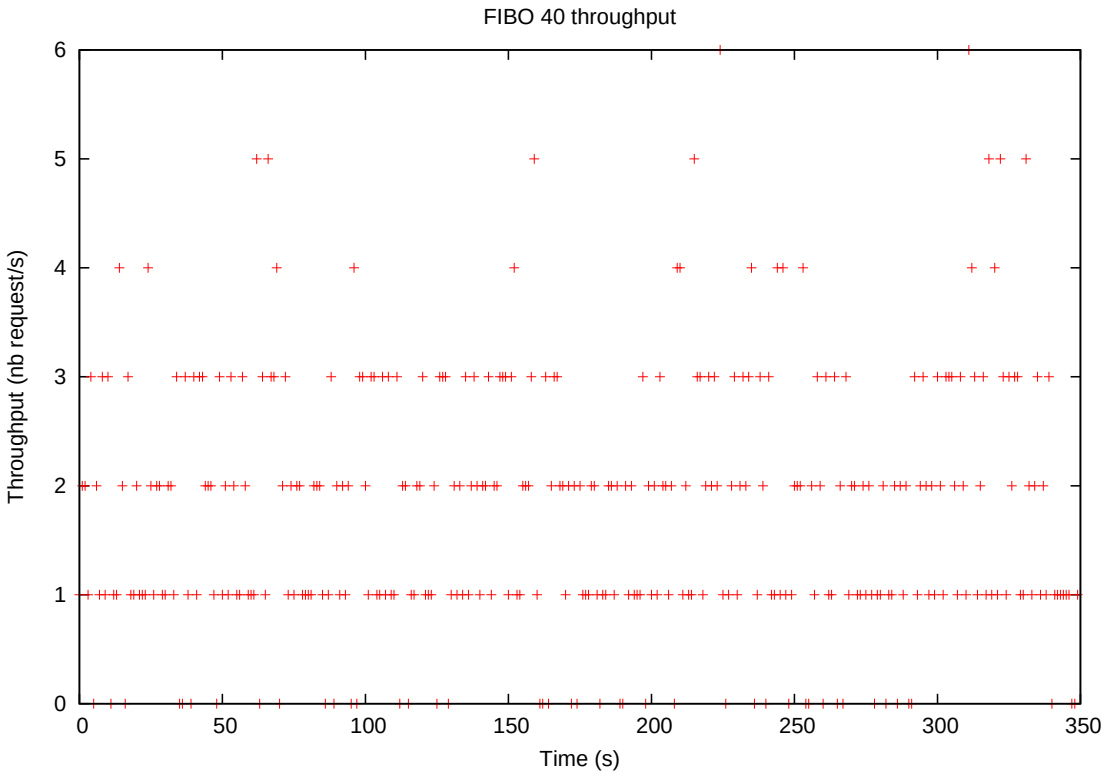

Figure 18: FIBO 40 throughput

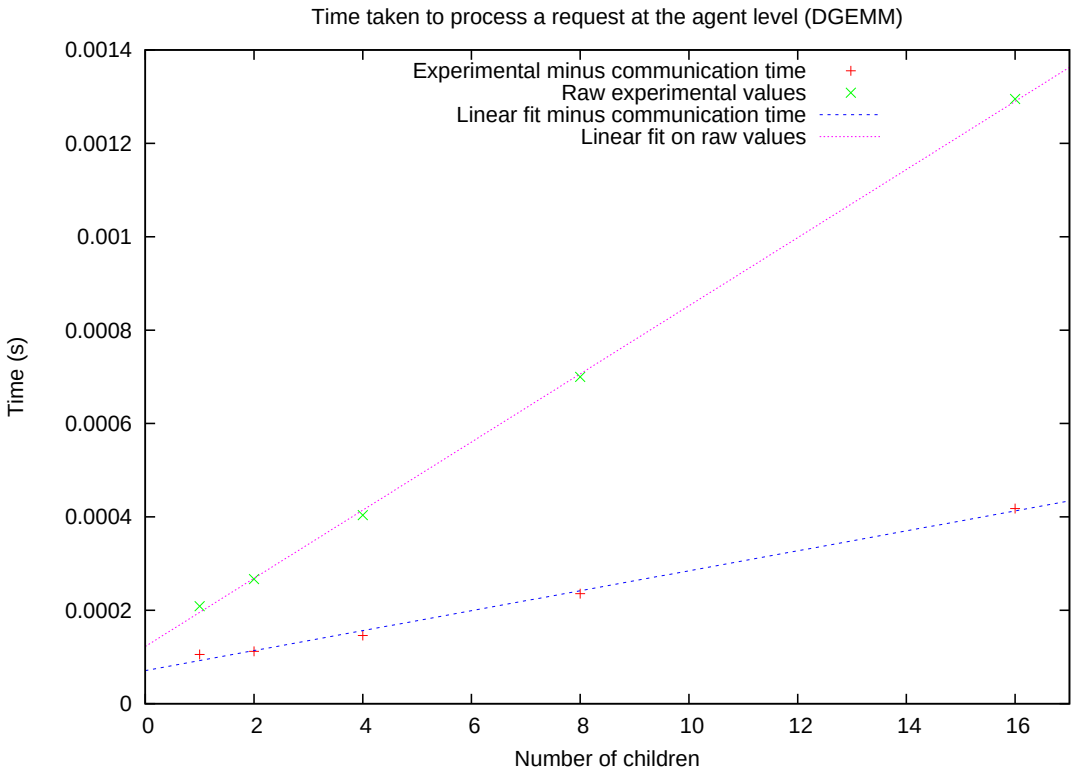

Figure 19: Agent's time to process a `dgemm` request.

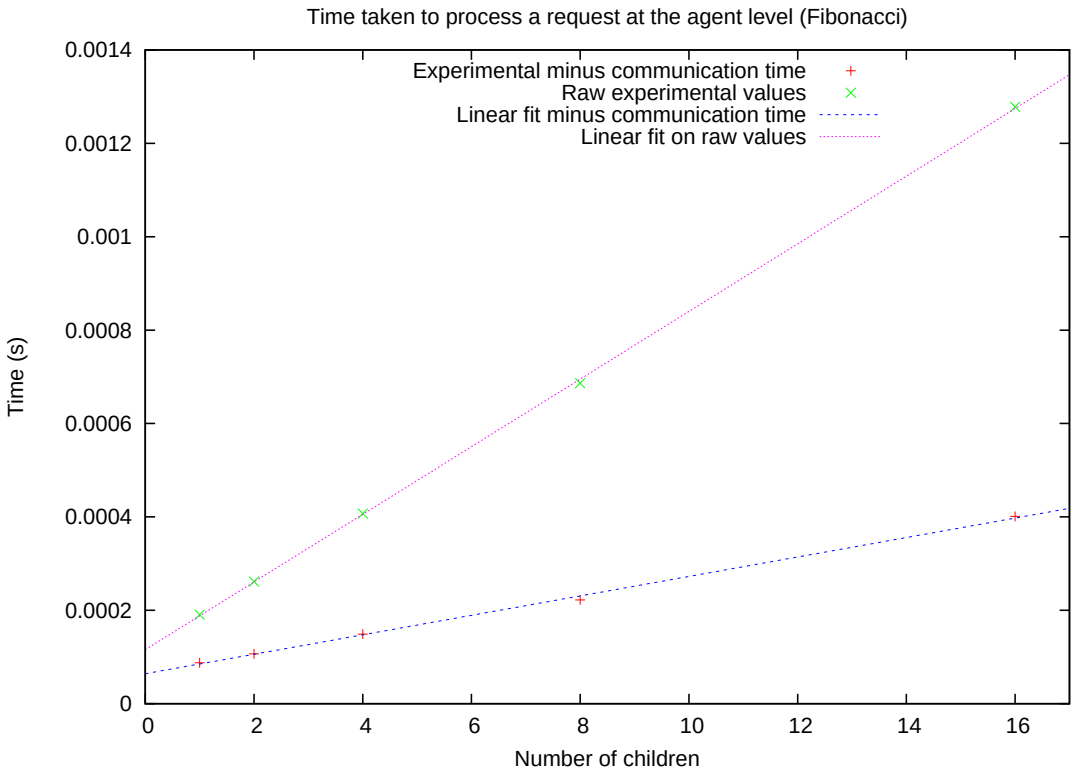

Figure 20: Agent's time to process a FIBO request.

8 Experimental Results

In order to validate our model, we conducted experiments with DIET on the French experimental testbed Grid'5000 [4]. After a phase of benchmarking for the DIET elements, the services (`dgemm` [13] and computation of the Fibonacci 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 MILP, it took only a few seconds to generate all the hierarchies).

Our goal here is to stress DIET, so we use relatively small services. We compared the throughput measured and predicted for various services combinations:

- `dgemm` 100×100 and Fibonacci 30 (medium size services)
- `dgemm` 10×10 and Fibonacci 20 (small size services)
- `dgemm` 10×10 and Fibonacci 40 (small size `dgemm`, large size Fibonacci)
- `dgemm` 500×500 and Fibonacci 20 (large size `dgemm`, small size Fibonacci)
- `dgemm` 500×500 and Fibonacci 40 (large size `dgemm`, large size Fibonacci)

8.1 Results `dgemm` 100 Fibonacci 30

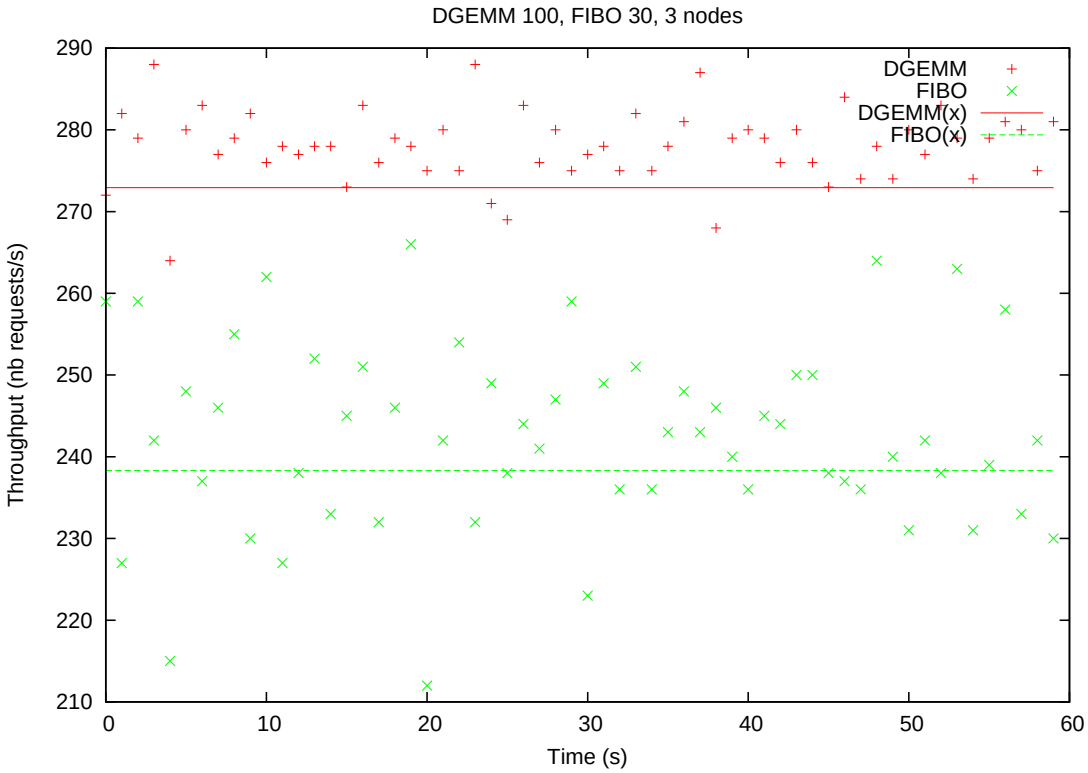

Figure 21: Throughputs for services `dgemm` 100 and Fibonacci 30, on 3 nodes.

Figures 21 to 24 present the results when requesting concurrently services `dgemm` 100 and Fibonacci 30. Table 8 sums up the results. The results are quite close to the expected throughput, even when the hierarchy as two levels of agents: for 20 nodes, the agent hierarchy contained 1 MA and 3 LA. Over 20 nodes, the algorithm returned the same hierarchy as with 20 nodes. Figure 25 presents graphically the results of Table 8.

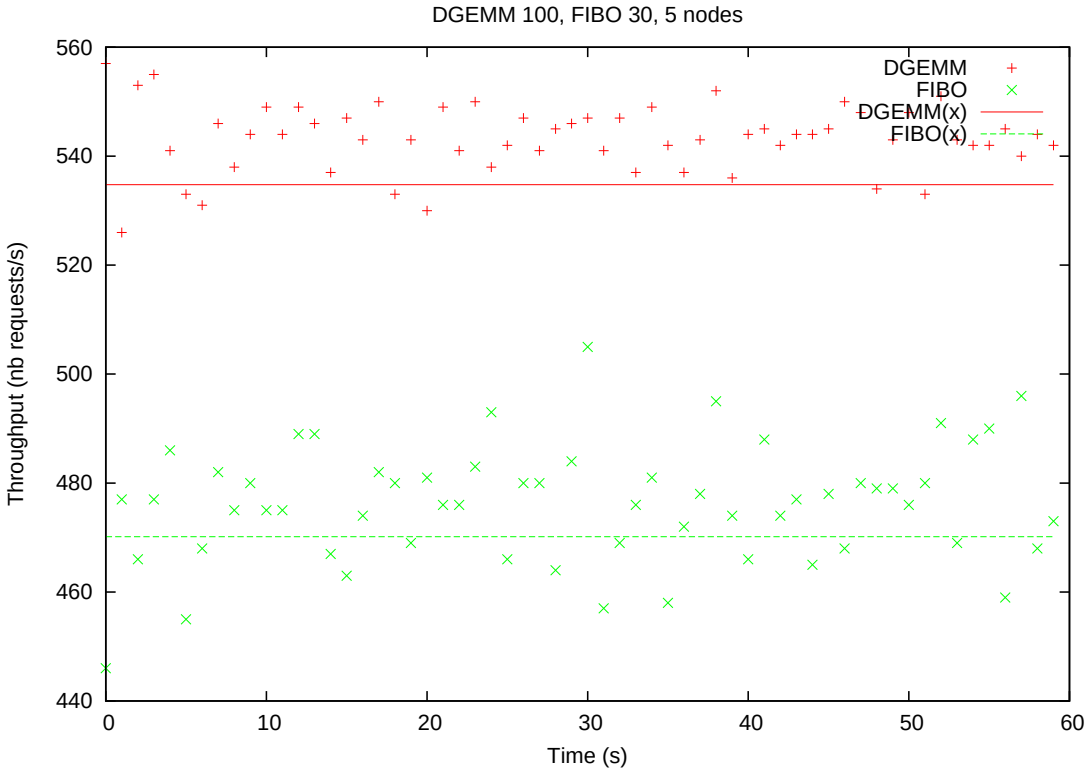

Figure 22: Throughputs for services `dgemm` 100 and Fibonacci 30, on 5 nodes.

No. nodes	Client	Theoretical	Mean	Median	Std Dev	Relative Error
3	`dgemm`	272.924	278.0	278	4.4	1.87%
	Fibonacci	238.317	242.5	242	11.2	1.76%
5	`dgemm`	534.758	543.2	544	6.1	1.58%
	Fibonacci	470.144	476.1	477	10.7	1.26%
10	`dgemm`	1027.75	984.9	995	49.5	4.17%
	Fibonacci	915.364	912.9	922	52.1	0.26%
20	`dgemm`	1699.05	1624.4	1666	114.7	4.39%
	Fibonacci	1738.56	1699.0	1735	114.0	2.28%

Table 8: Comparison between theoretical and experimental throughputs, for `dgemm` 100 Fibonacci 30. Relative error: $\frac{|Theoretical - Mean|}{Theoretical}$.

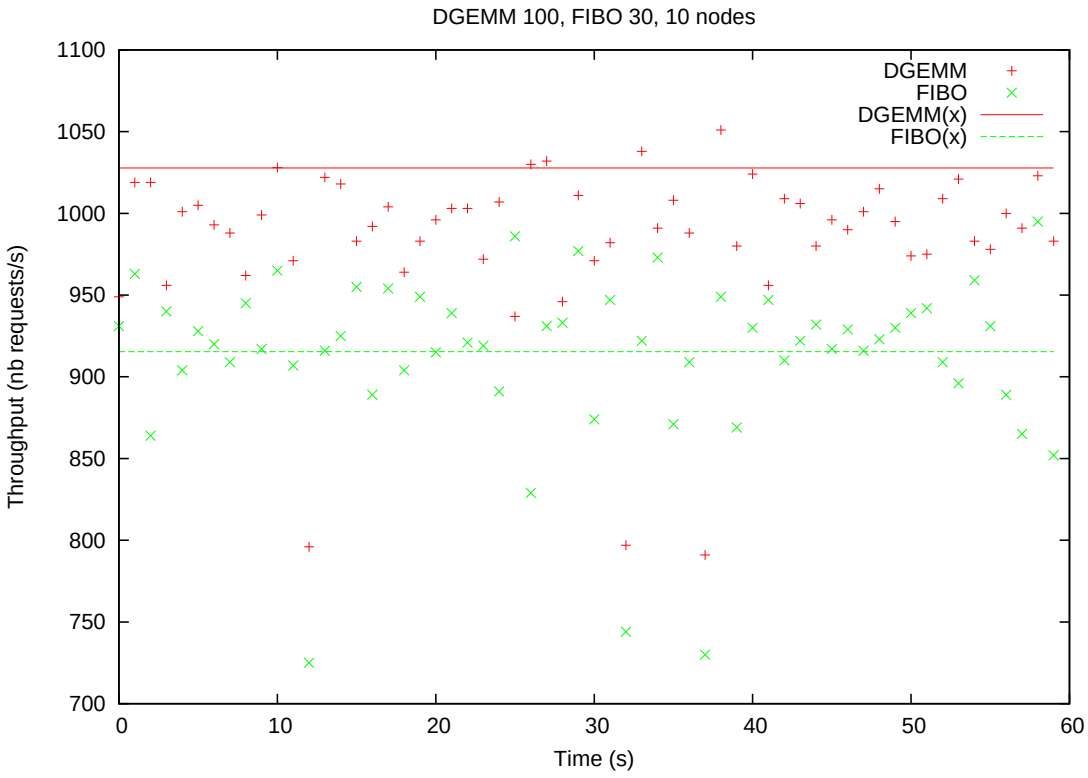

Figure 23: Throughputs for services `dgemm` 100 and Fibonacci 30, on 10 nodes.

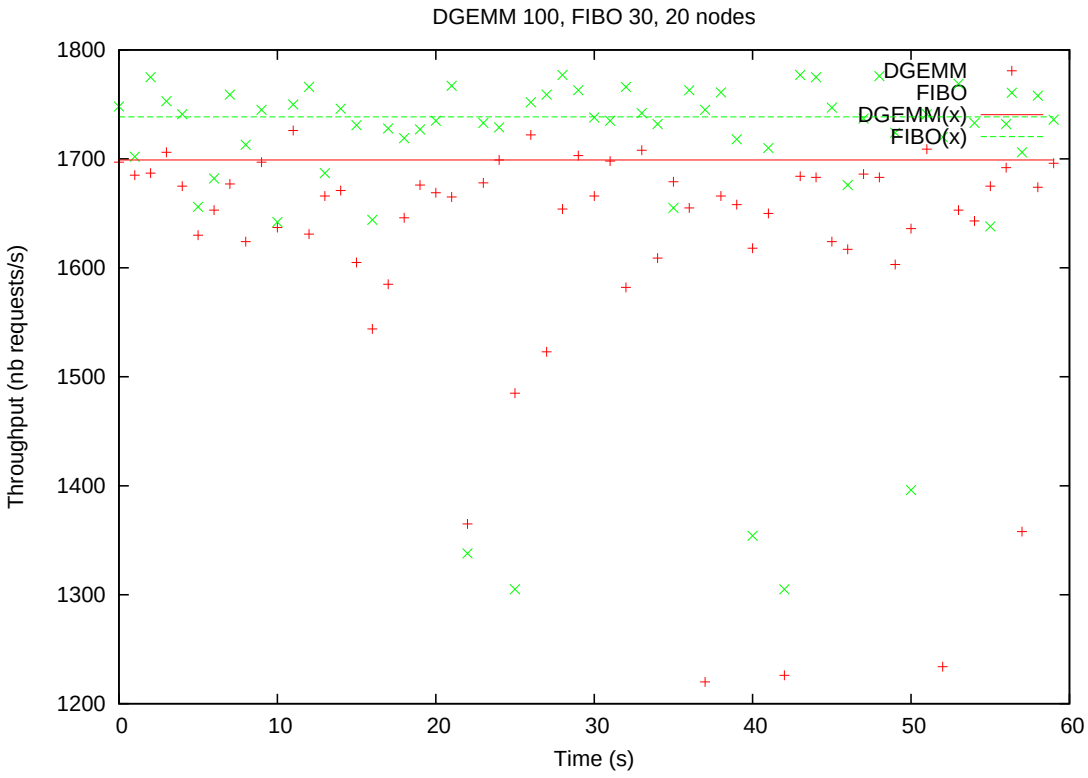

Figure 24: Throughputs for services `dgemm` 100 and Fibonacci 30, on 20 nodes.

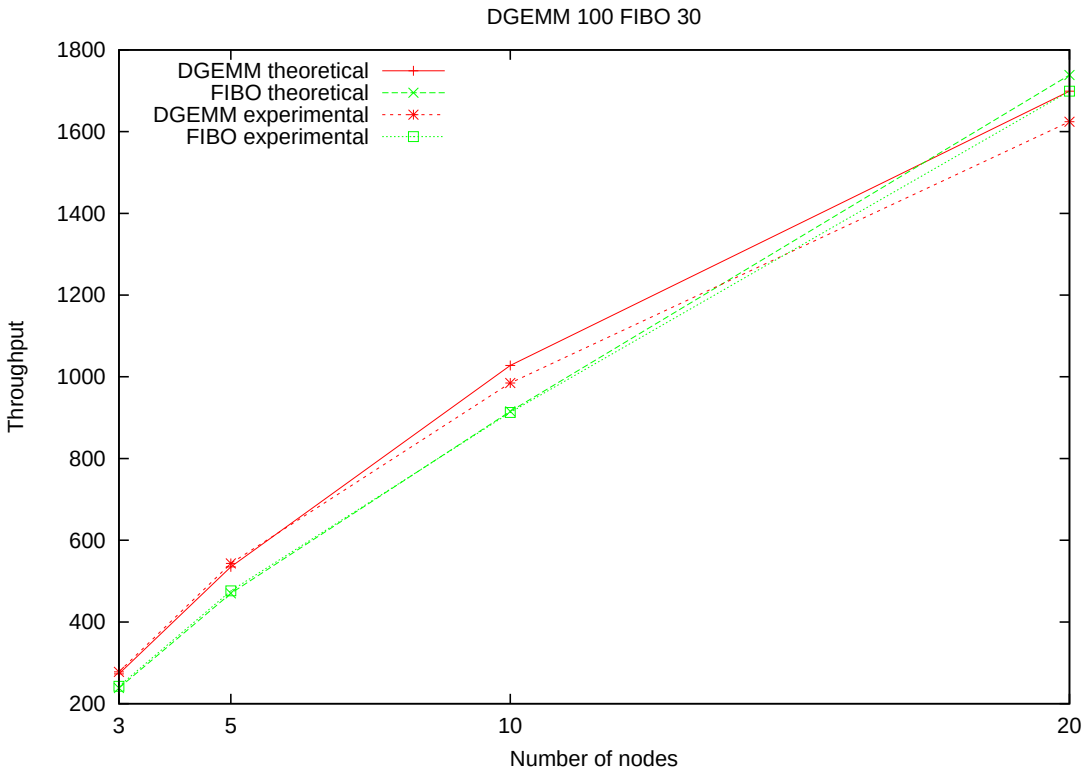

Figure 25: Comparison theoretical/experimental results, for `dgemm` 100 Fibonacci 30.

8.2 Results `dgemm` 10 Fibonacci 20

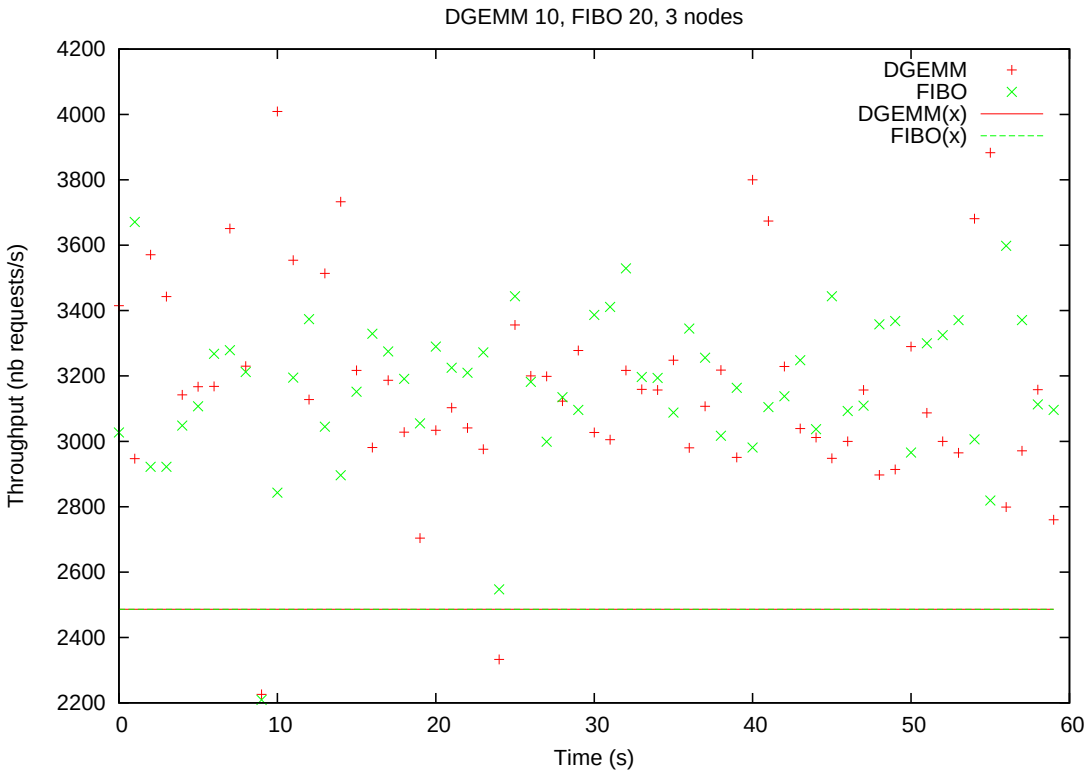

Figure 26: Throughputs for services `dgemm` 10 and Fibonacci 20, on 3 nodes.

No. nodes	Client	Theoretical	Mean	Median	Std Dev	Relative Error
3	`dgemm`	2486.33	3166.5	3157	319.7	27.4%
	Fibonacci	2486.33	3164.2	3191	231.3	27.3%

Table 9: Comparison between theoretical and experimental throughputs, for `dgemm` 10 Fibonacci 20. Relative error: $\frac{|Theoretical - Mean|}{Theoretical}$.

Figure 26 presents the results when requesting concurrently services `dgemm` 10 and Fibonacci 20. Table 8 sums up the results. For really small services such as presented here, the error increases greatly. Clearly here the problem lies in a bad estimation of the costs incurred by this kind of requests, *i.e.*, really small requests in terms of required computation at the server level are harder to benchmark correctly. The limiting factor is the agent, hence, over 3 nodes the hierarchy always contained only 3 nodes: 1 MA and 1 SeD for each service.

8.3 Results dgemm 10 Fibonacci 40

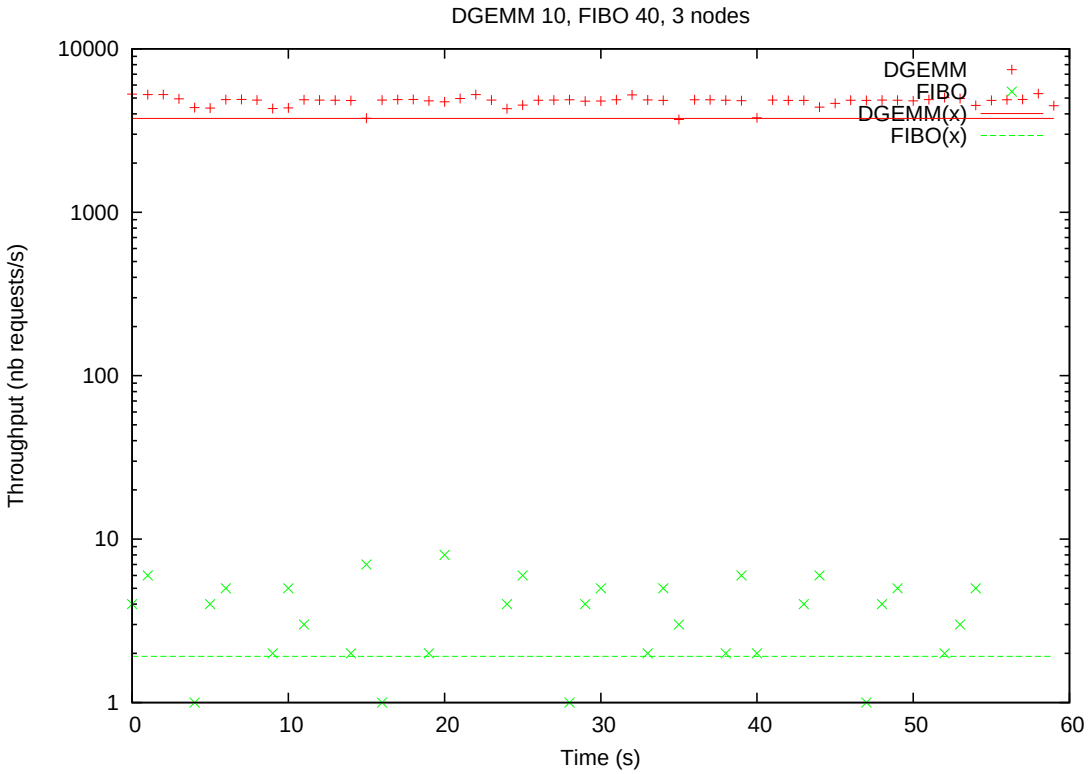

Figure 27: Throughputs for services dgemm 10 and Fibonacci 40, on 3 nodes.

No. nodes	Client	Theoretical	Mean	Median	Std Dev	Relative Error
3	dgemm	3752.54	4773.5	5853	325.5	27.2%
	Fibonacci	1.92	2.2	2	2.33	14.6%
5	dgemm	3752.54	4842.7	4856	212.2	29.1%
	Fibonacci	5.75	6.35	6	2.4	10.4%
10	dgemm	3752.54	4805.9	4854	292.5	28.1%
	Fibonacci	15.32	16.8	18	7.3	9.7%
20	dgemm	3752.54	4828.9	4859	279.4	28.7%
	Fibonacci	34.44	37.2	37	3.0	8.0%
30	dgemm	3752.54	4811.2	4882	358.2	28.2%
	Fibonacci	53.51	57.1	58	2.9	6.7%
40	dgemm	3752.54	4613.4	4739	386.2	22.9%
	Fibonacci	72.55	71.5	72	4.9	1.4%
50	dgemm	3258.15	4037.0	4072	433.9	23.9%
	Fibonacci	91.54	93.6	95	8.8	2.3%

Table 10: Comparison between theoretical and experimental throughputs, for dgemm 10 Fibonacci 40. Relative error: $\frac{|Theoretical-Mean|}{Theoretical}$.

Figures 27 to 33 present the results when requesting concurrently services dgemm 10 and Fibonacci 40. Table 10 sums up the results. The Fibonacci service closely follows the model, whereas

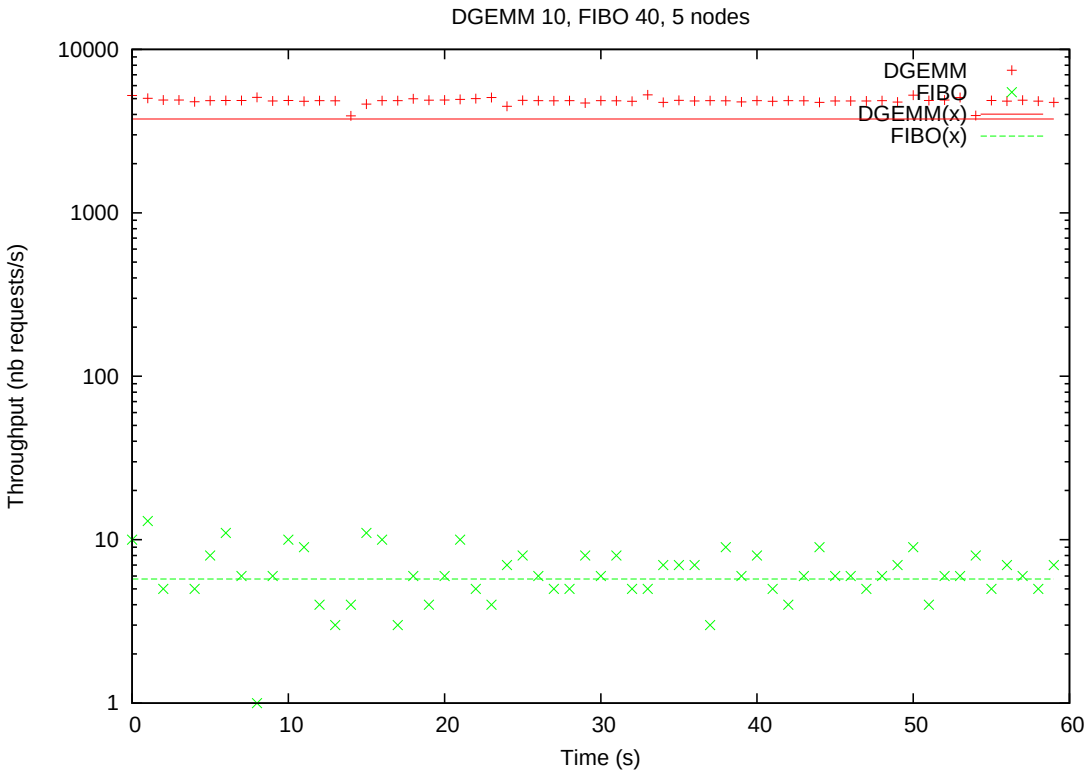

Figure 28: Throughputs for services `dgemm` 10 and Fibonacci 40, on 5 nodes.

the DGEMM service parts from it due to benchmarking problems. Figure 34 presents graphically the results of Table 10.

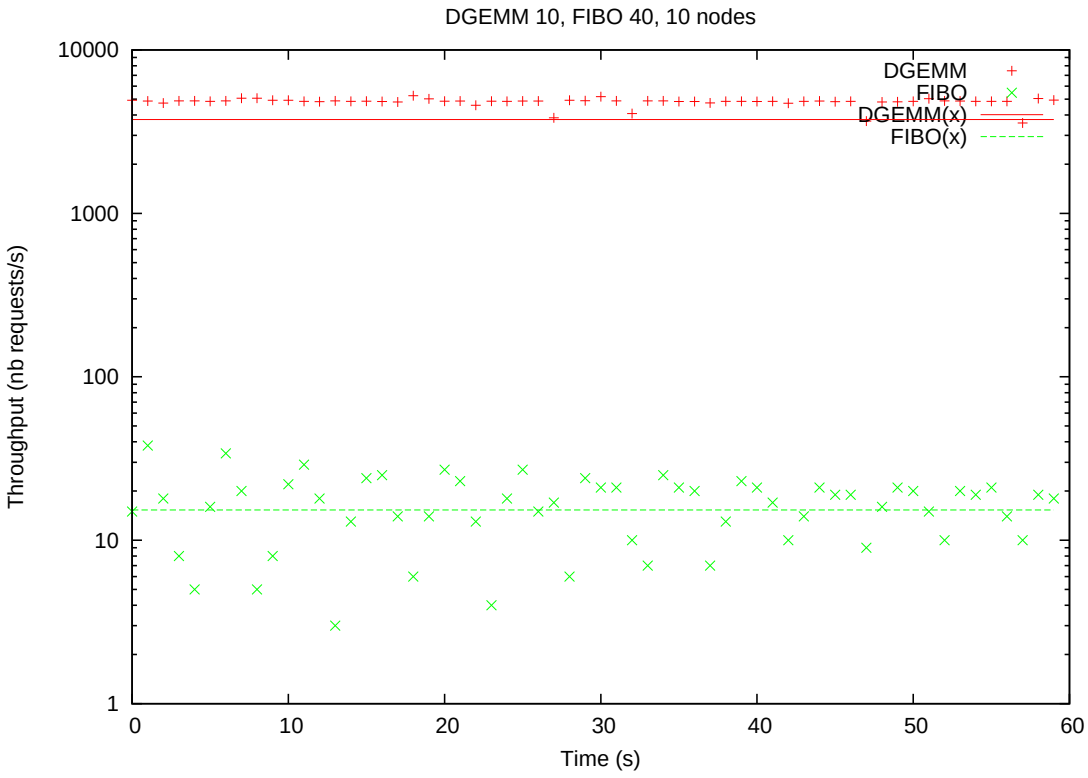

Figure 29: Throughputs for services `dgemm` 10 and Fibonacci 40, on 10 nodes.

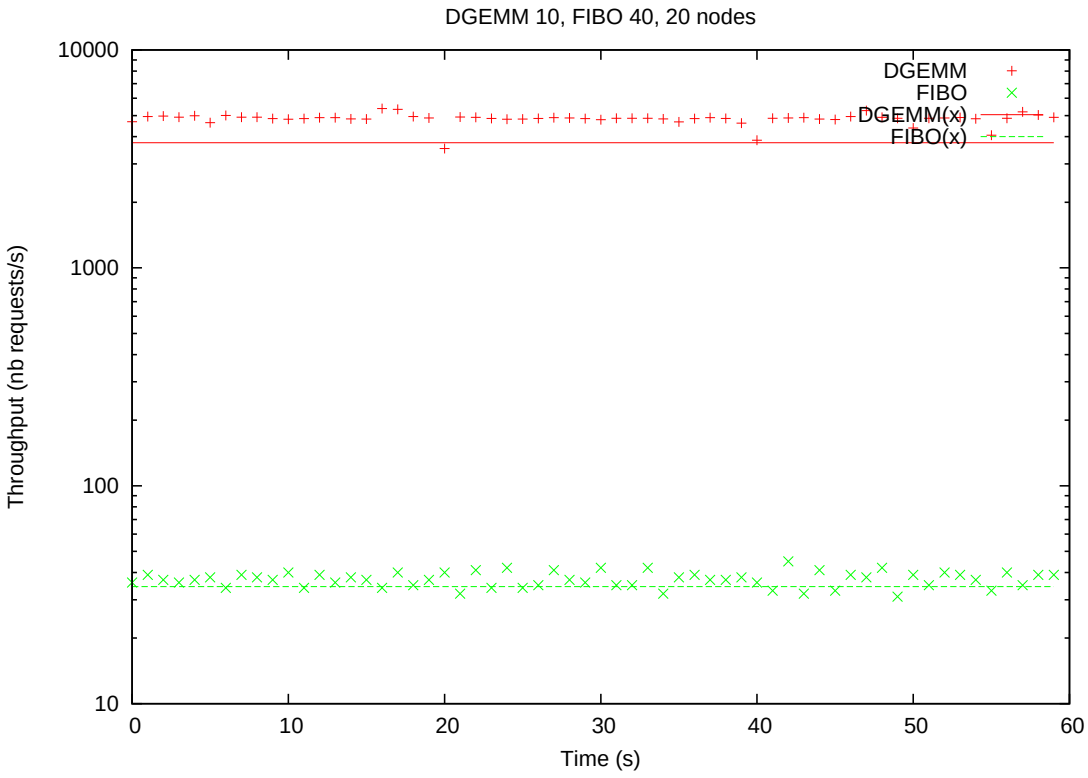

Figure 30: Throughputs for services `dgemm` 10 and Fibonacci 40, on 20 nodes.

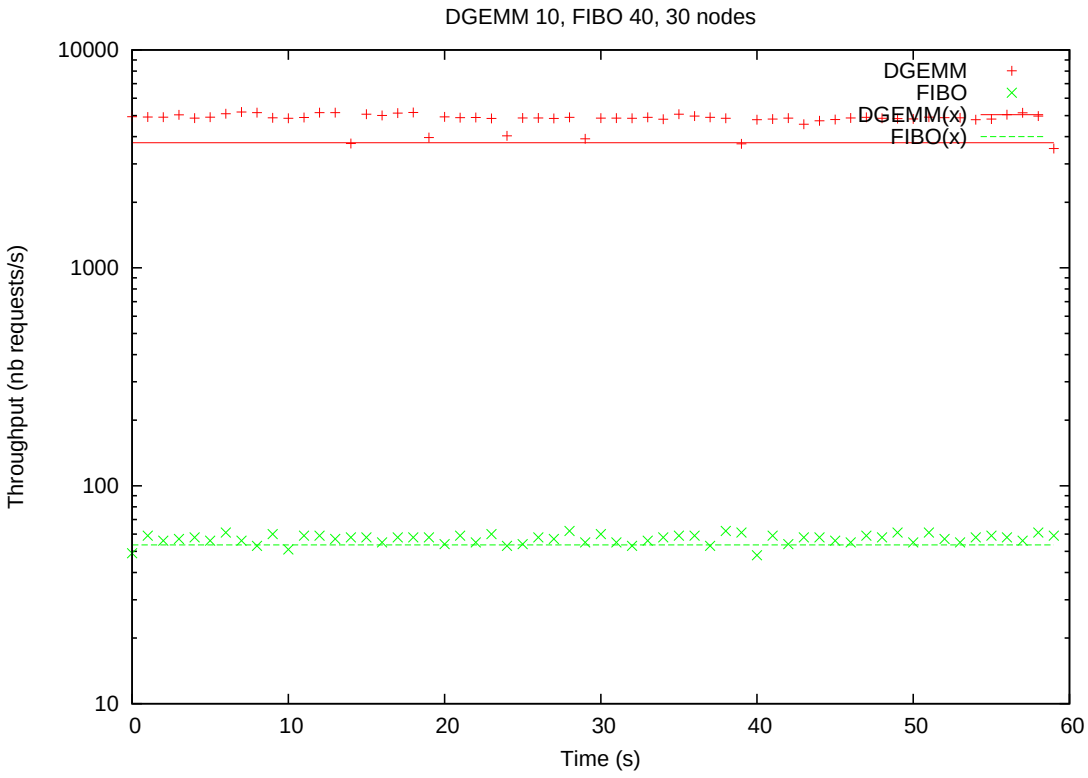

Figure 31: Throughputs for services `dgemm` 10 and Fibonacci 40, on 30 nodes.

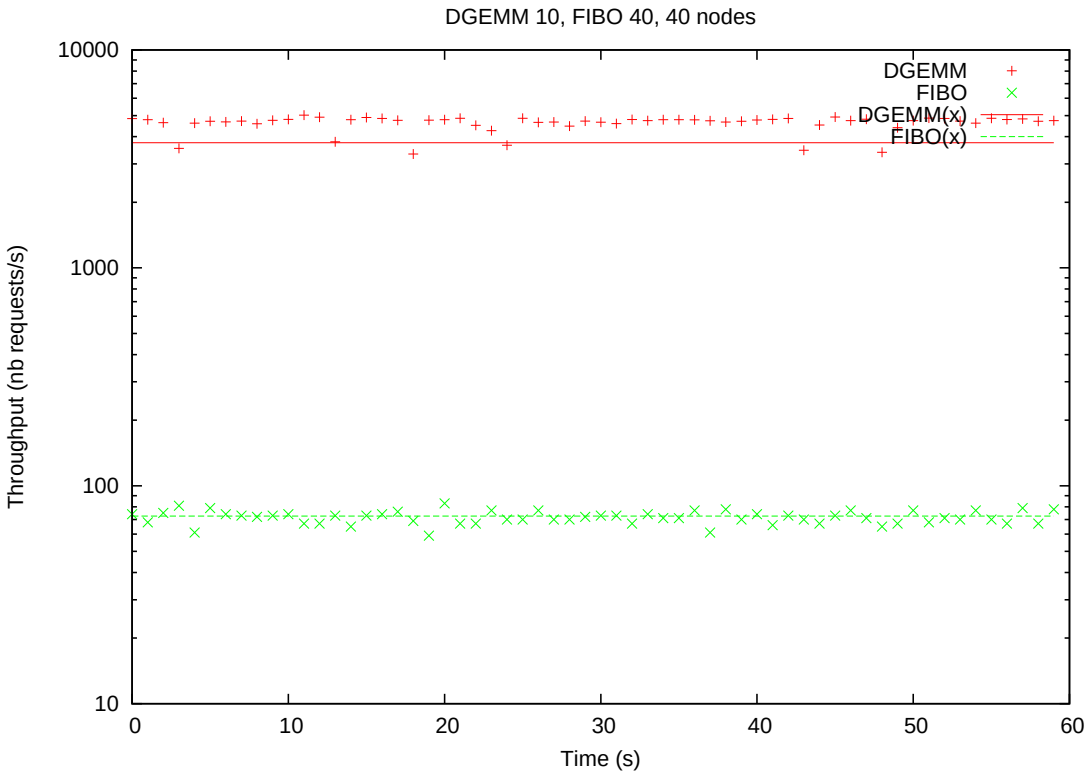

Figure 32: Throughputs for services `dgemm` 10 and Fibonacci 40, on 40 nodes.

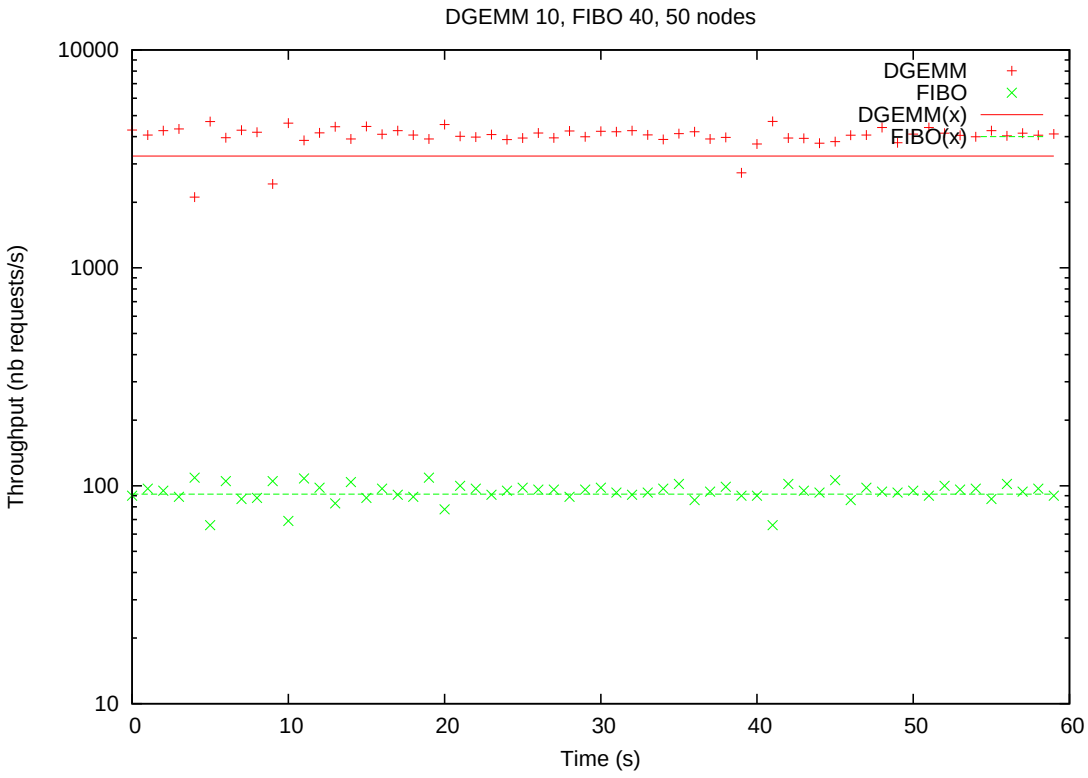

Figure 33: Throughputs for services `dgemm` 10 and Fibonacci 40, on 50 nodes.

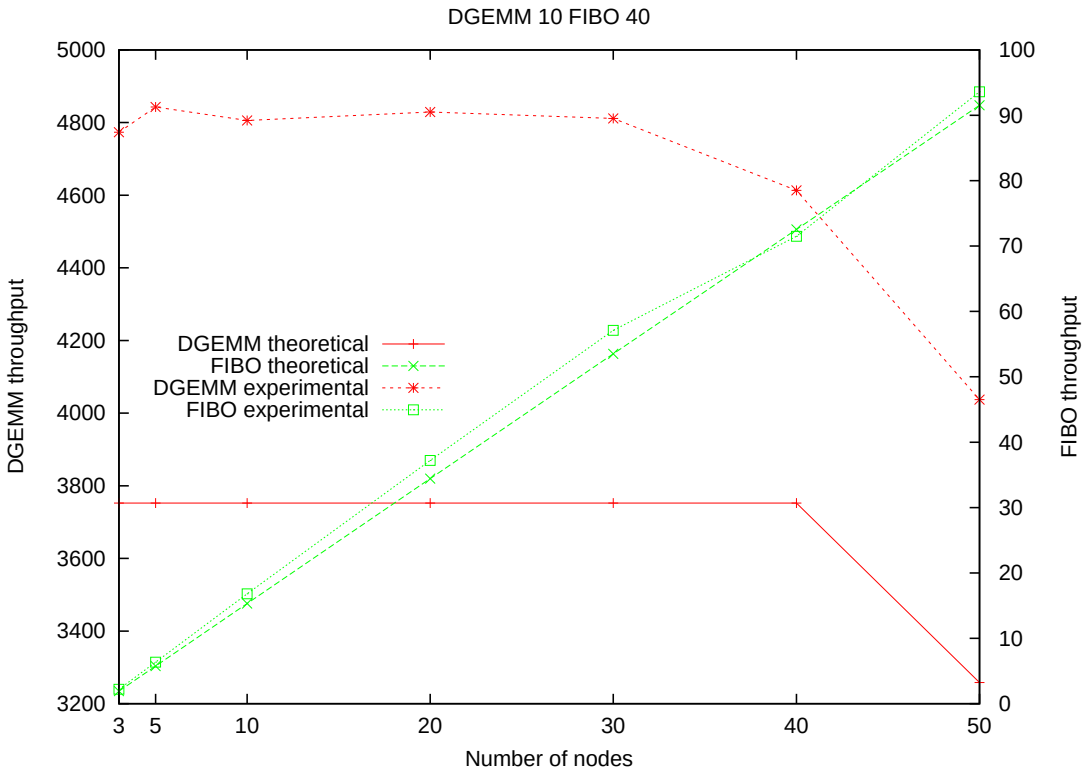

Figure 34: Comparison theoretical/experimental results, for `dgemm` 10 Fibonacci 40.

8.4 Results dgemm 500 Fibonacci 20

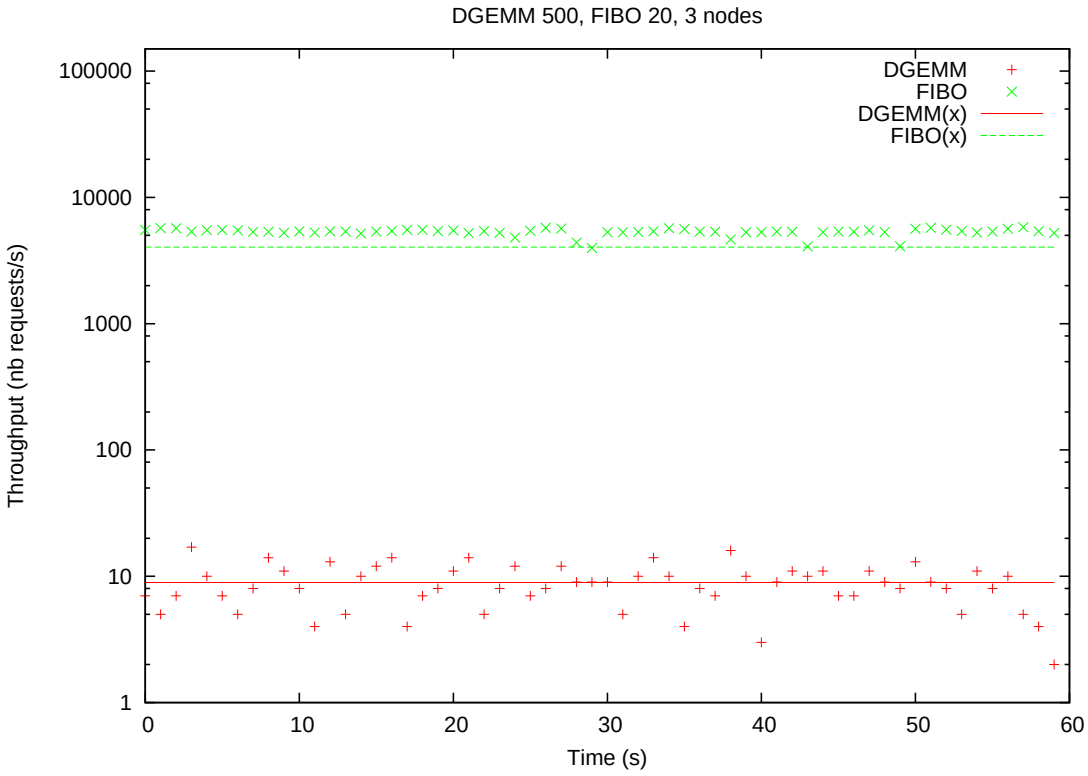

Figure 35: Throughputs for services dgemm 500 and Fibonacci 20, on 3 nodes.

No. nodes	Client	Theoretical	Mean	Median	Std Dev	Relative Error
3	dgemm	8.9	8.8	9	3.3	1.1%
	Fibonacci	4034.6	5318.1	5366	372.4	31.8%
5	dgemm	26.7	26.5	27	3.6	0.7%
	Fibonacci	4034.6	5351.8	5328	206.9	32.6%
10	dgemm	70.95	68.8	69	4.8	3.0%
	Fibonacci	4034.6	5221.0	5258	264.4	26.4%
20	dgemm	158.1	153.3	153	10.9	3.0%
	Fibonacci	3856.3	4851.8	4865	339.1	25.8%
30	dgemm	235.2	224.0	228	23.8	4.7%
	Fibonacci	4034.6	4845.85	4946	289.3	20.1%
40	dgemm	311.0	281.35	291	55.6	9.5%
	Fibonacci	2539.3	2391.6	2466	473.8	5.8%
50	dgemm	385.5	311.7	328	63.2	19.1%
	Fibonacci	2694.3	2973.3	2960	352.1	10.4%

Table 11: Comparison between theoretical and experimental throughputs, for dgemm 500 Fibonacci 20. Relative error: $\frac{|Theoretical - Mean|}{Theoretical}$.

Figures 35 to 41 present the results when requesting concurrently services dgemm 500 and Fibonacci 20. Table 11 sums up the results. Here the small service also suffers from the benchmarking

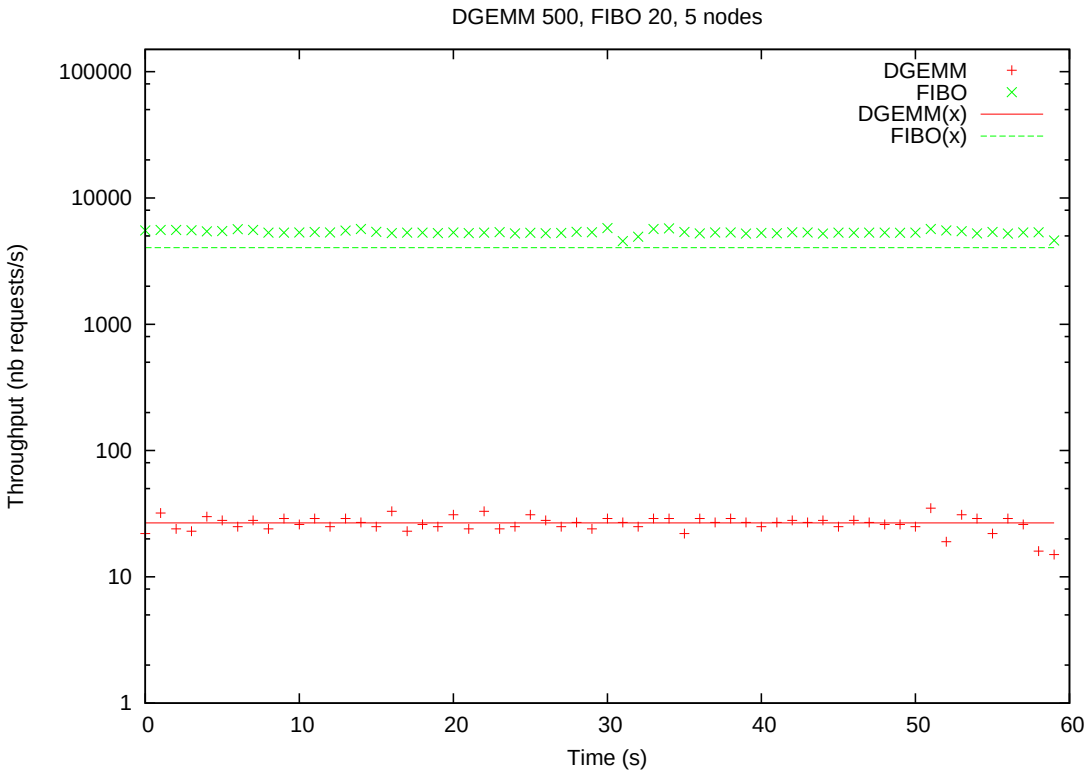

Figure 36: Throughputs for services `dgemm` 500 and Fibonacci 20, on 5 nodes.

problems. `dgemm` results diverges a bit from the theoretical predictions. This is certainly due to the fact that our model does not explicitly take into account the client/server communications (they are implicitly taken into account in the "time" required by the server to serve a request). Figure 42 presents graphically the results of Table 11.

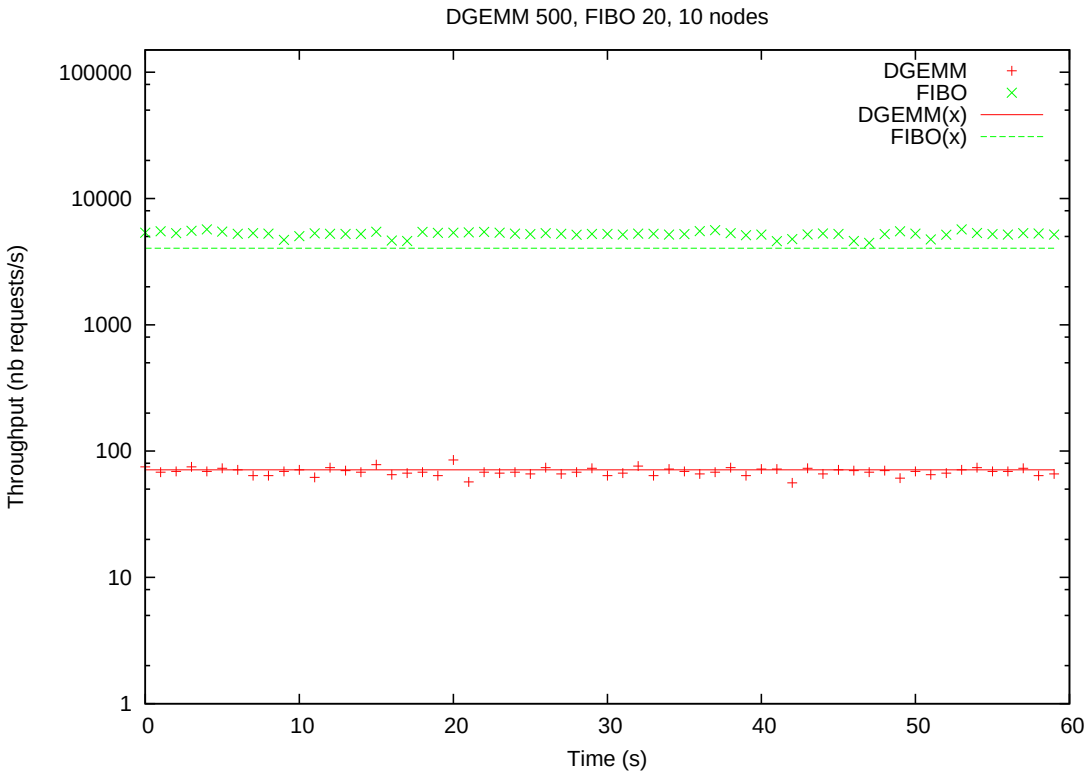

Figure 37: Throughputs for services `dgemm` 500 and Fibonacci 20, on 10 nodes.

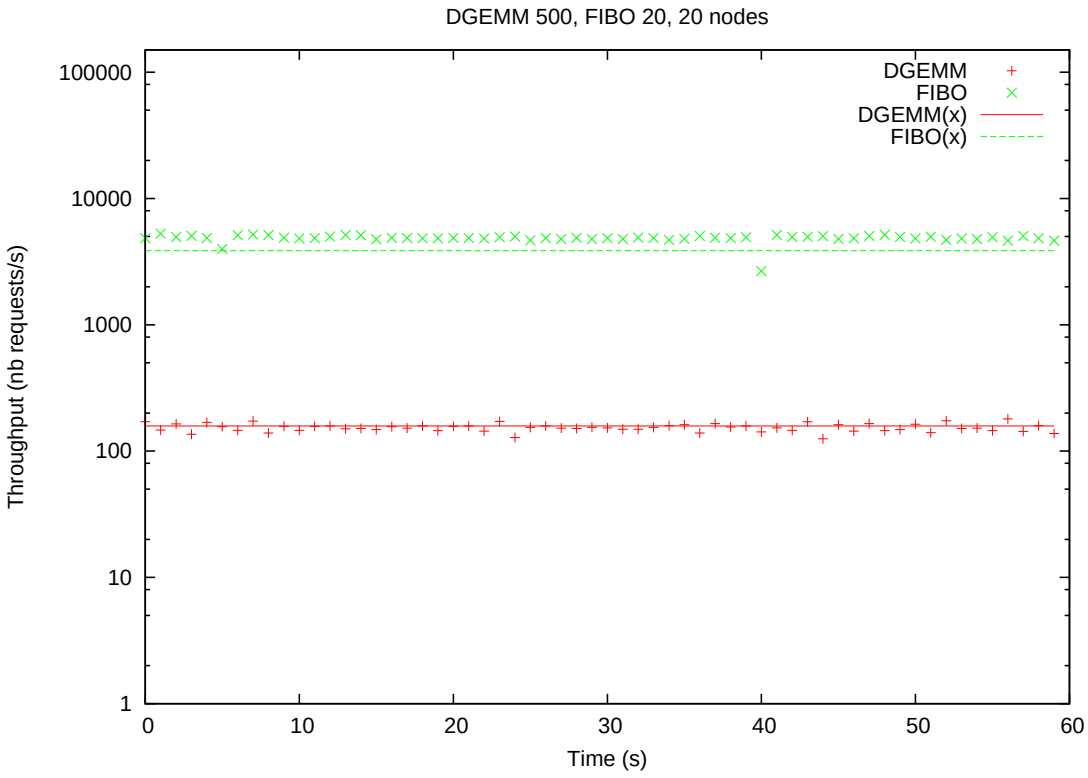

Figure 38: Throughputs for services `dgemm` 500 and Fibonacci 20, on 20 nodes.

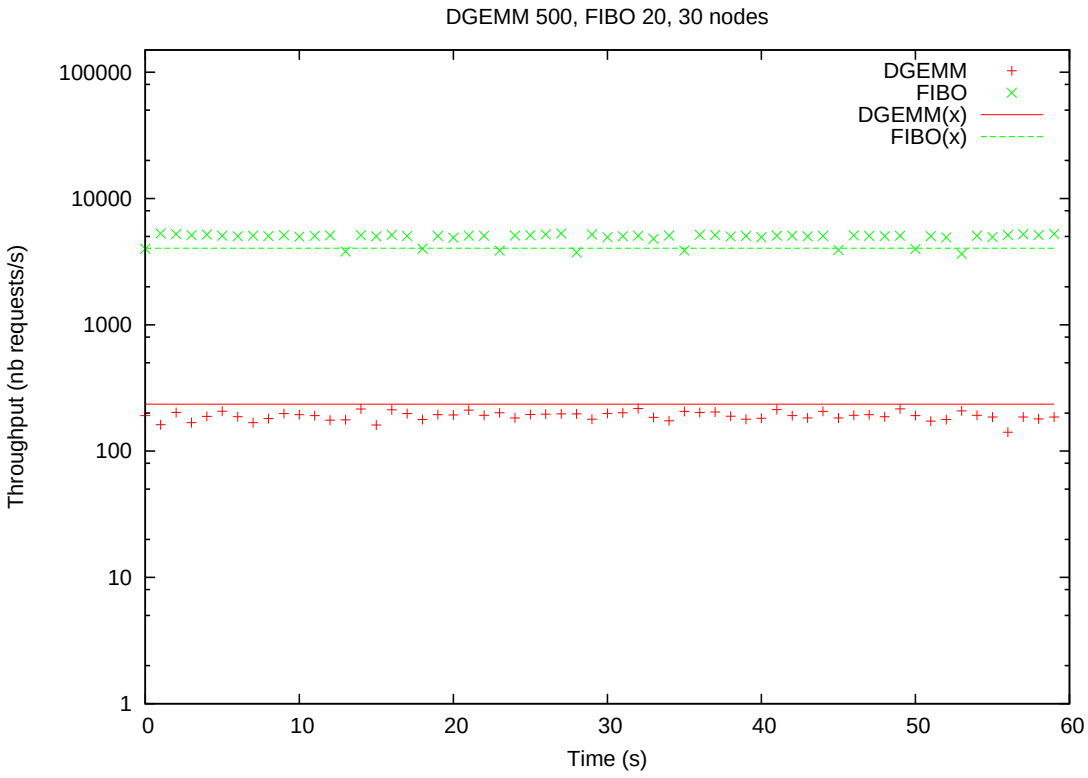

Figure 39: Throughputs for services `dgemm` 500 and Fibonacci 20, on 30 nodes.

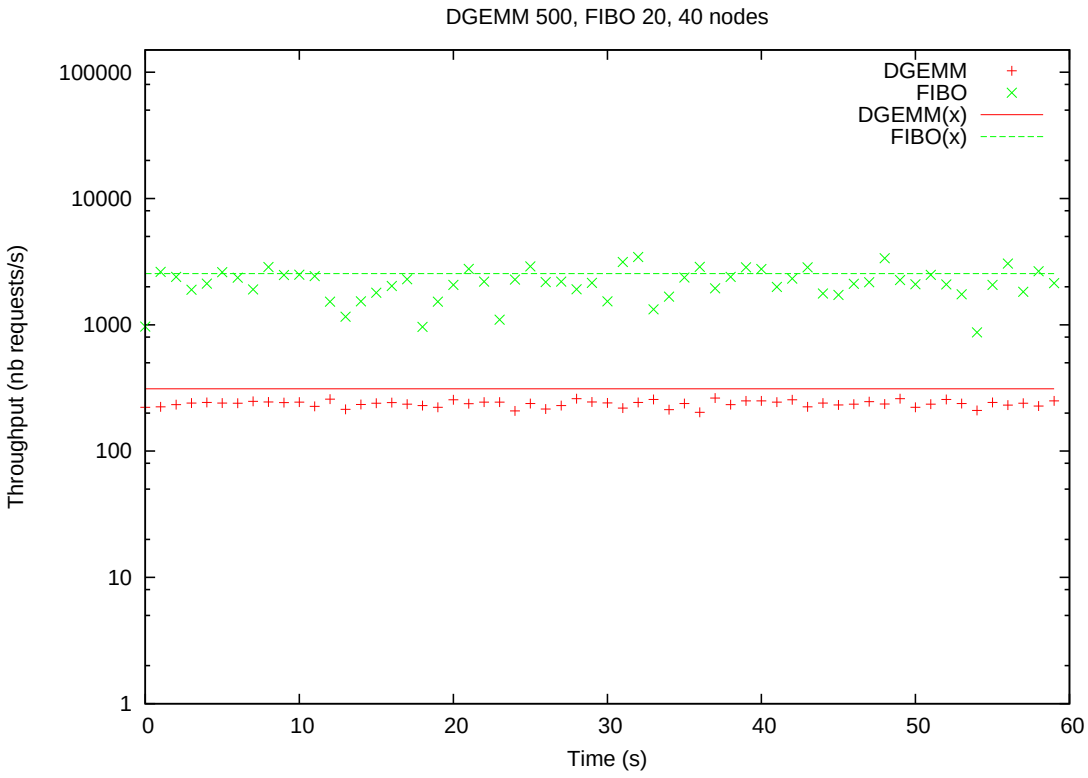

Figure 40: Throughputs for services `dgemm` 500 and Fibonacci 20, on 40 nodes.

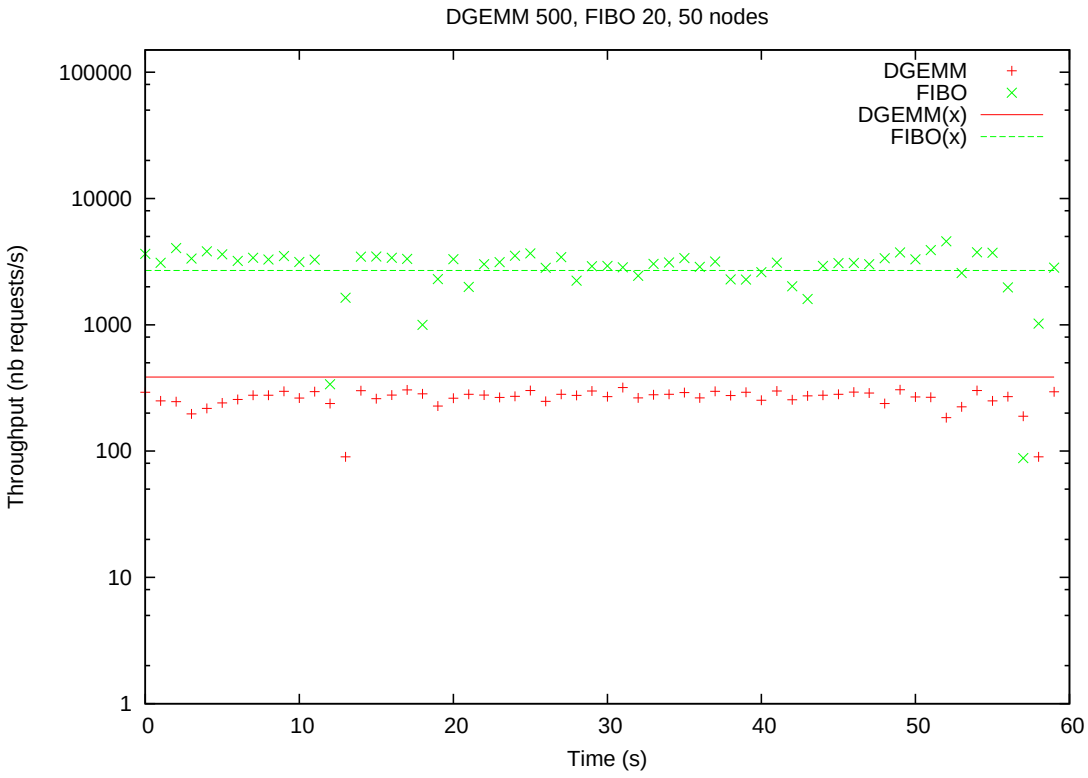

Figure 41: Throughputs for services `dgemm` 500 and Fibonacci 20, on 50 nodes.

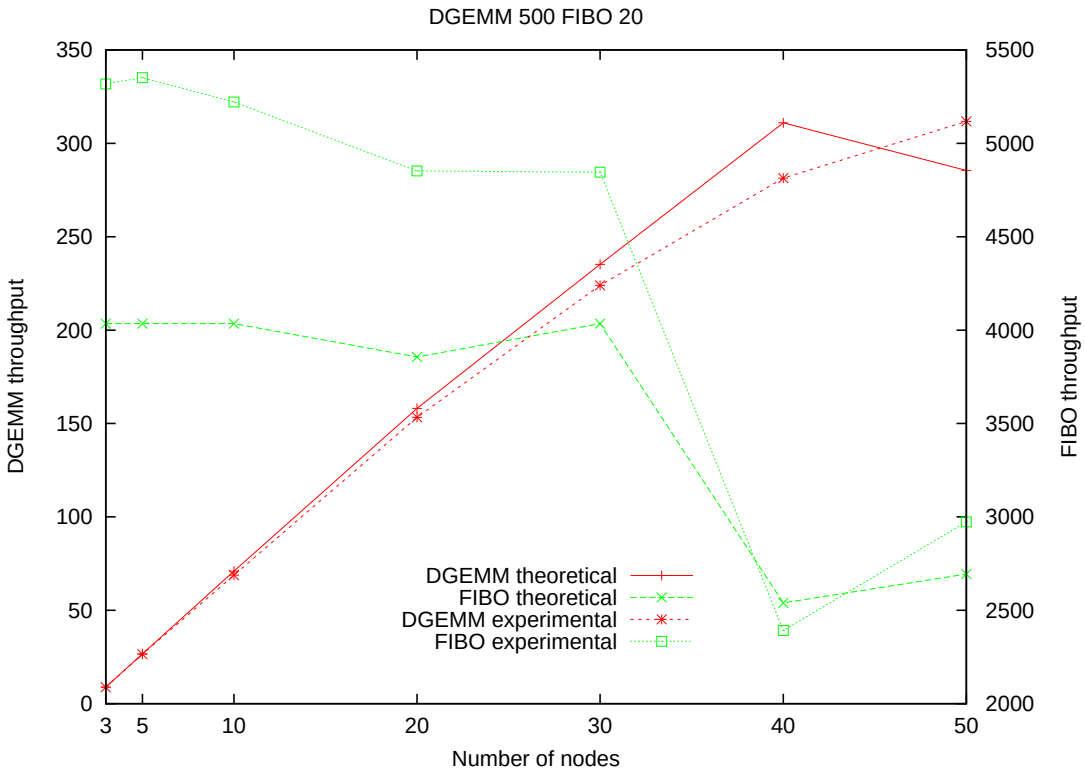

Figure 42: Comparison theoretical/experimental results, for `dgemm` 500 Fibonacci 20.

8.5 Results dgemm 500 Fibonacci 40

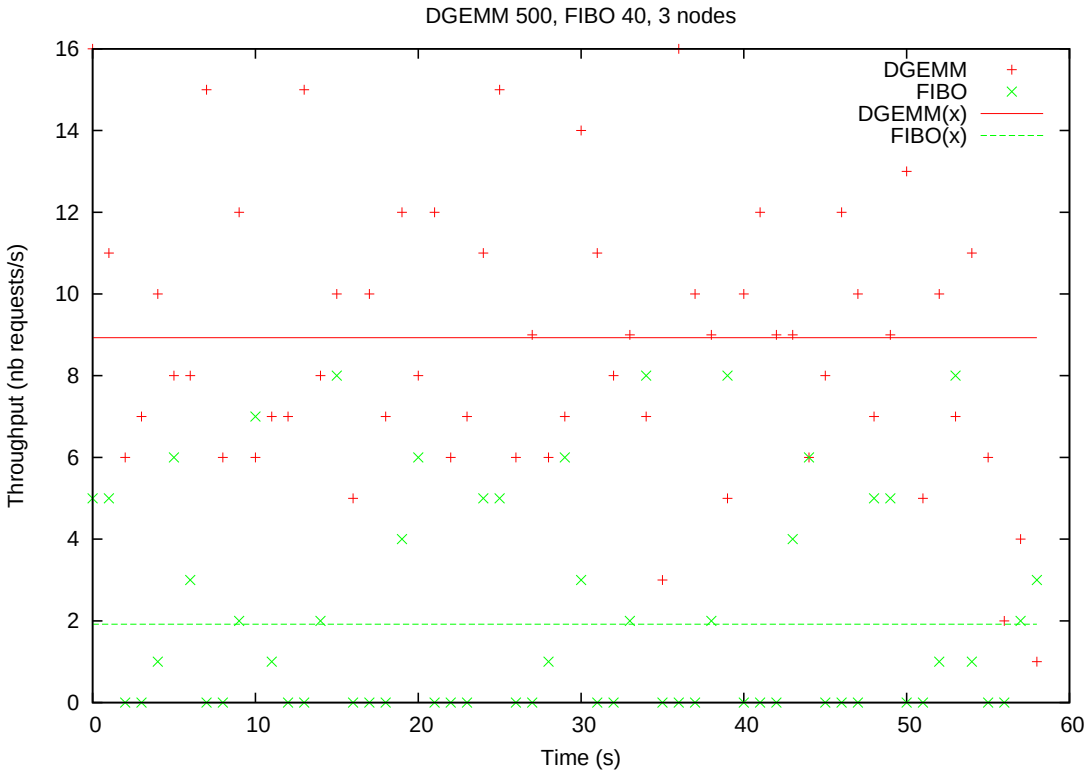

Figure 43: Throughputs for services dgemm 500 and Fibonacci 40, on 3 nodes.

No. nodes	Client	Theoretical	Mean	Median	Std Dev	Relative Error
3	dgemm	8.9	8.7	8	3.3	2.2%
	Fibonacci	1.9	2.1	1	2.7	10.5%
5	dgemm	8.9	8.2	8	3.8	7.9%
	Fibonacci	5.7	6.3	7	3.2	10.5%
10	dgemm	17.8	17.1	17	4.6	3.9%
	Fibonacci	13.4	14.5	15	4.2	8.2%
20	dgemm	35.6	34.6	36	6.0	2.8%
	Fibonacci	28.7	31.2	31	4.4	8.7%
30	dgemm	44.5	44.5	45	4.7	0.0%
	Fibonacci	45.9	48.5	48	3.4	5.7%
40	dgemm	62.1	61.8	62	6.3	0.5%
	Fibonacci	61.1	63.9	64	3.6	4.6%
50	dgemm	79.7	79.5	80	5.1	0.3%
	Fibonacci	76.4	76.6	77	3.9	0.3%

Table 12: Comparison between theoretical and experimental throughputs, for dgemm 500 Fibonacci 40. Relative error: $\frac{|Theoretical - Mean|}{Theoretical}$.

Figures 43 to 49 present the results when requesting concurrently services dgemm 500 and Fibonacci 40. Table 12 sums up the results. In these experiments, the experimental results closely

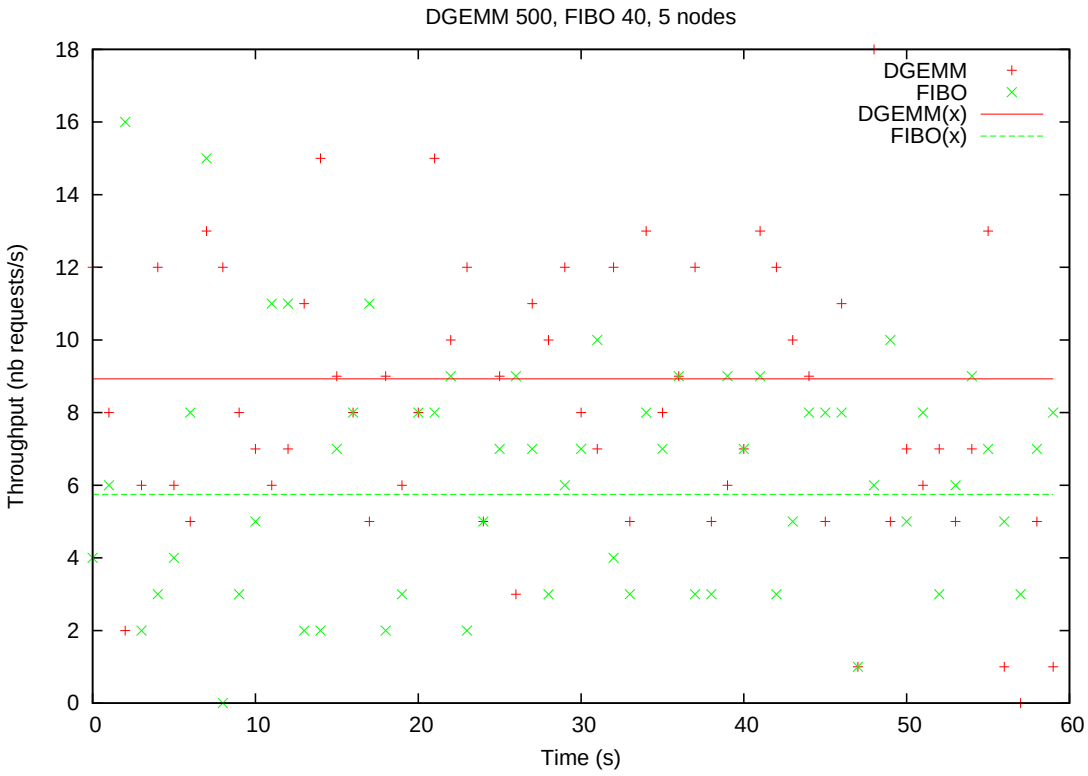

Figure 44: Throughputs for services `dgemm` 500 and Fibonacci 40, on 5 nodes.

follows the model, with sometimes less than 0.5% of relative error. Figure 50 presents graphically the results of Table 12.

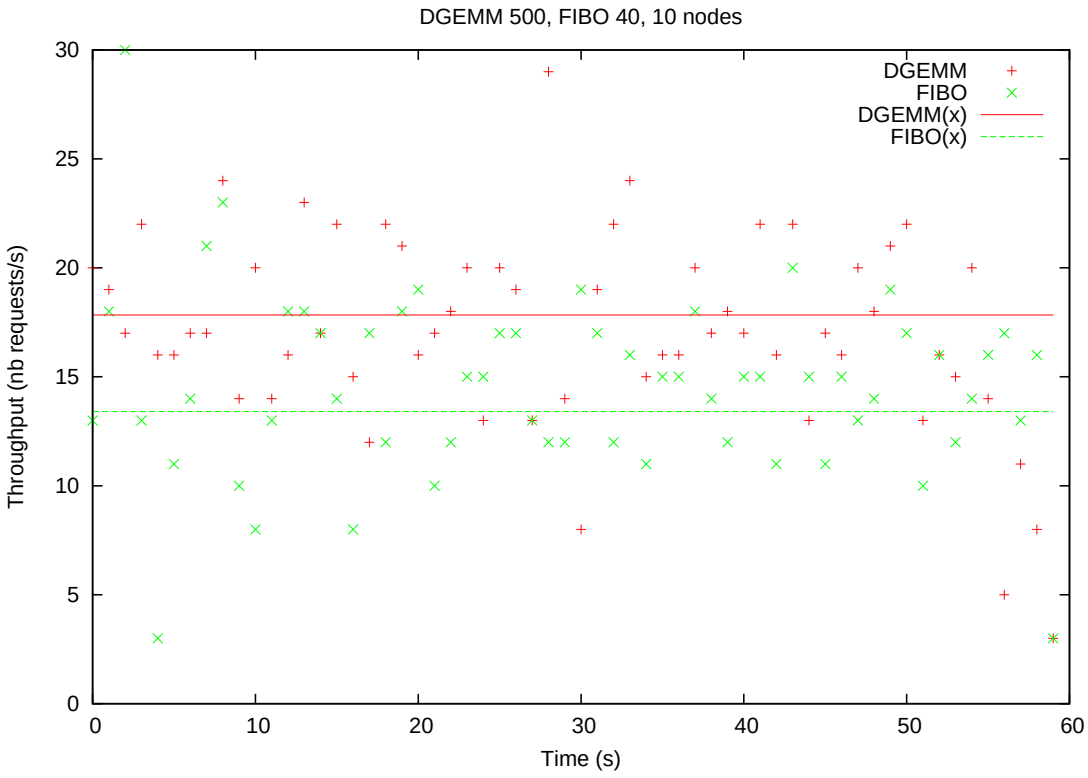

Figure 45: Throughputs for services `dgemm` 500 and Fibonacci 40, on 10 nodes.

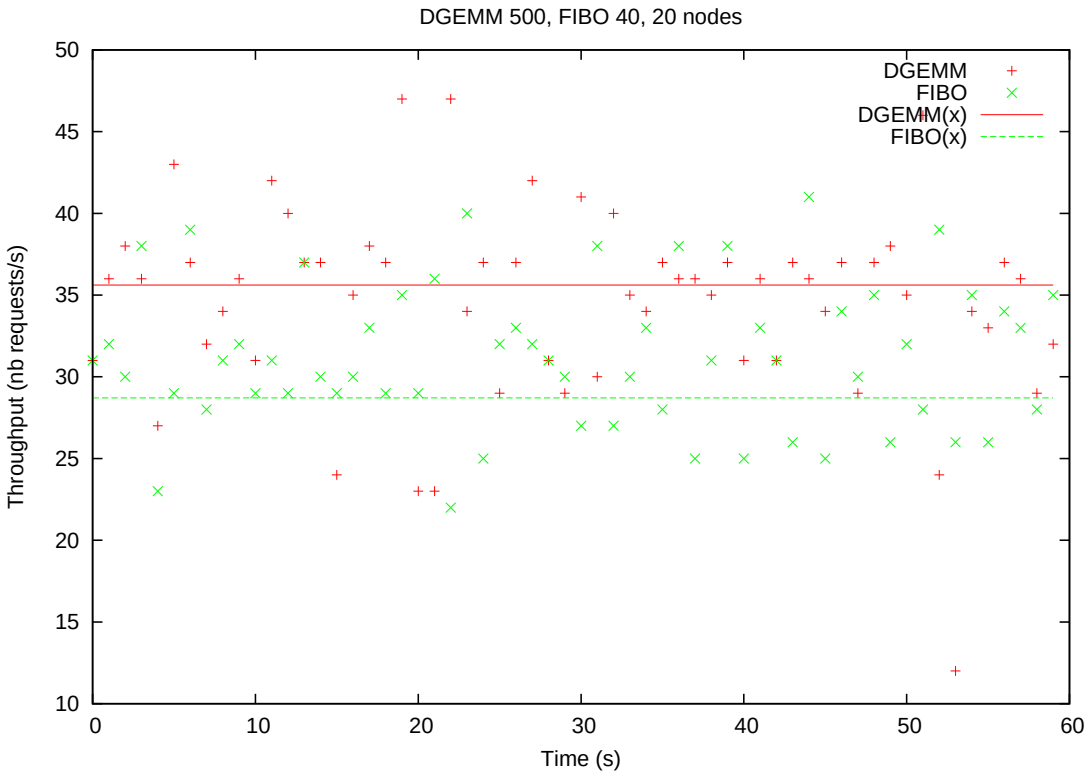

Figure 46: Throughputs for services `dgemm` 500 and Fibonacci 40, on 20 nodes.

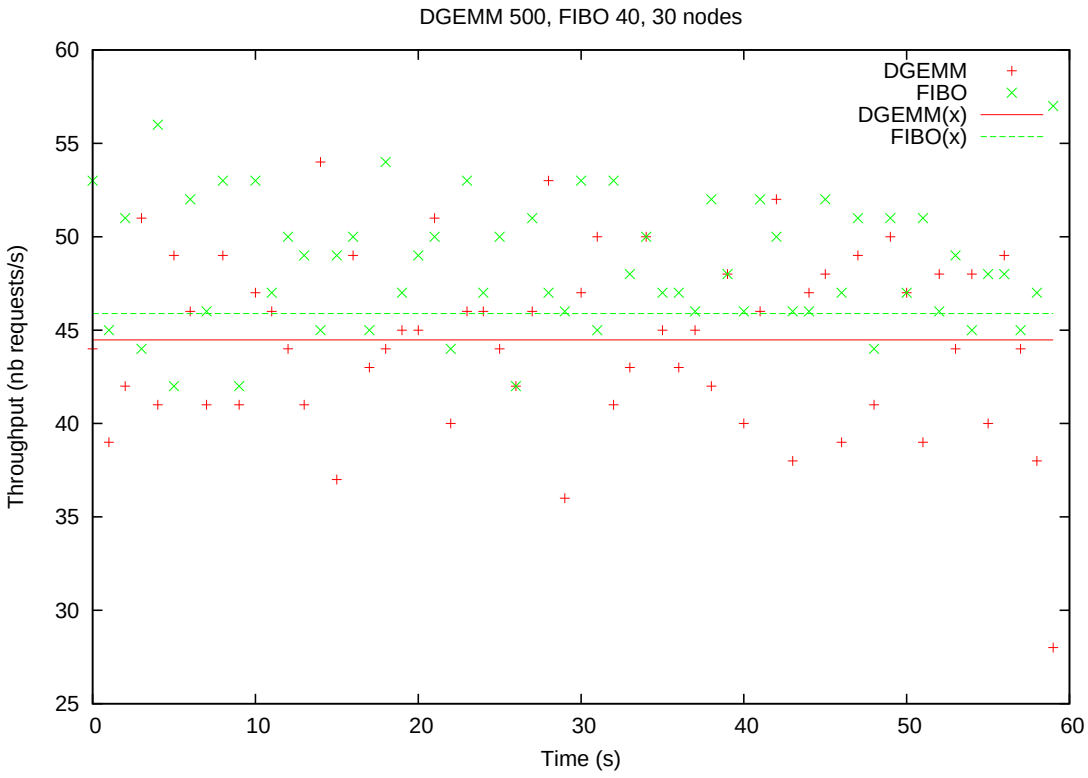

Figure 47: Throughputs for services `dgemm` 500 and Fibonacci 40, on 30 nodes.

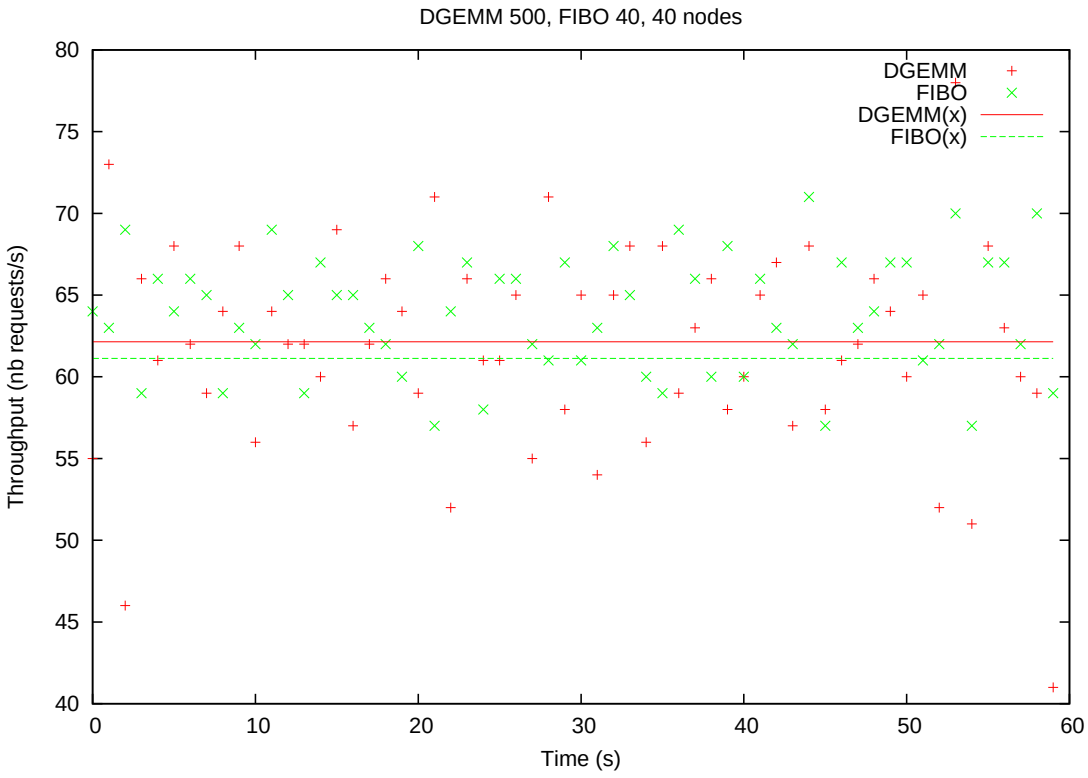

Figure 48: Throughputs for services `dgemm` 500 and Fibonacci 40, on 40 nodes.

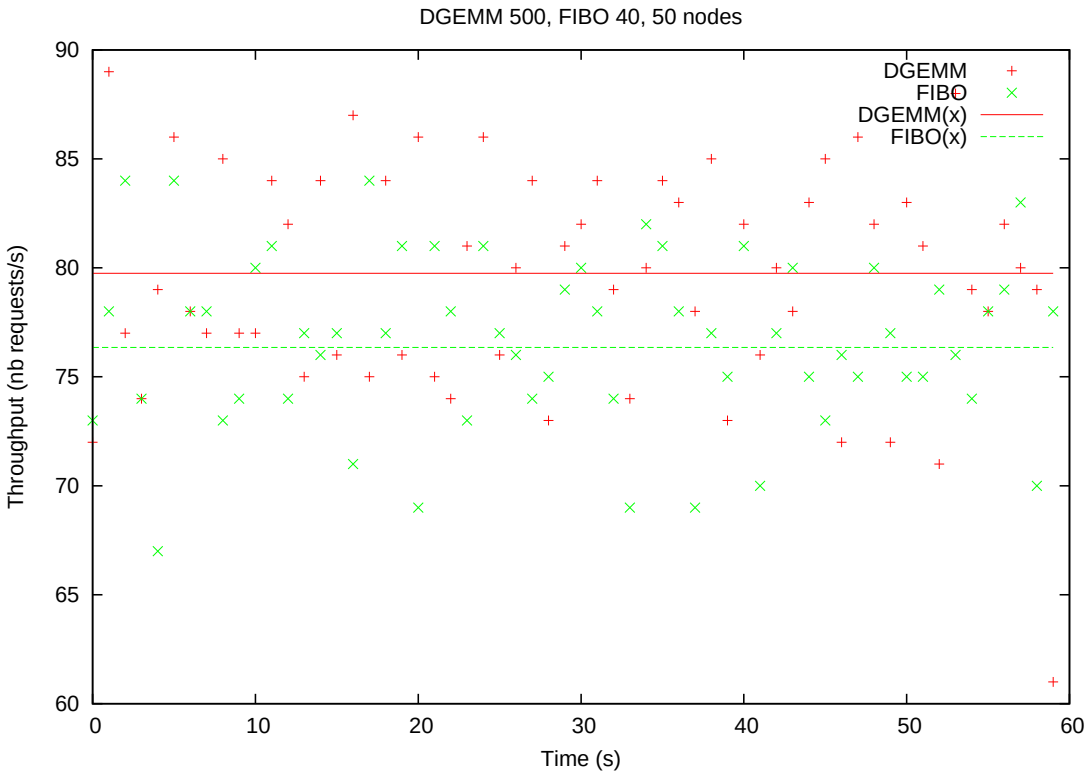

Figure 49: Throughputs for services `dgemm` 500 and Fibonacci 40, on 50 nodes.

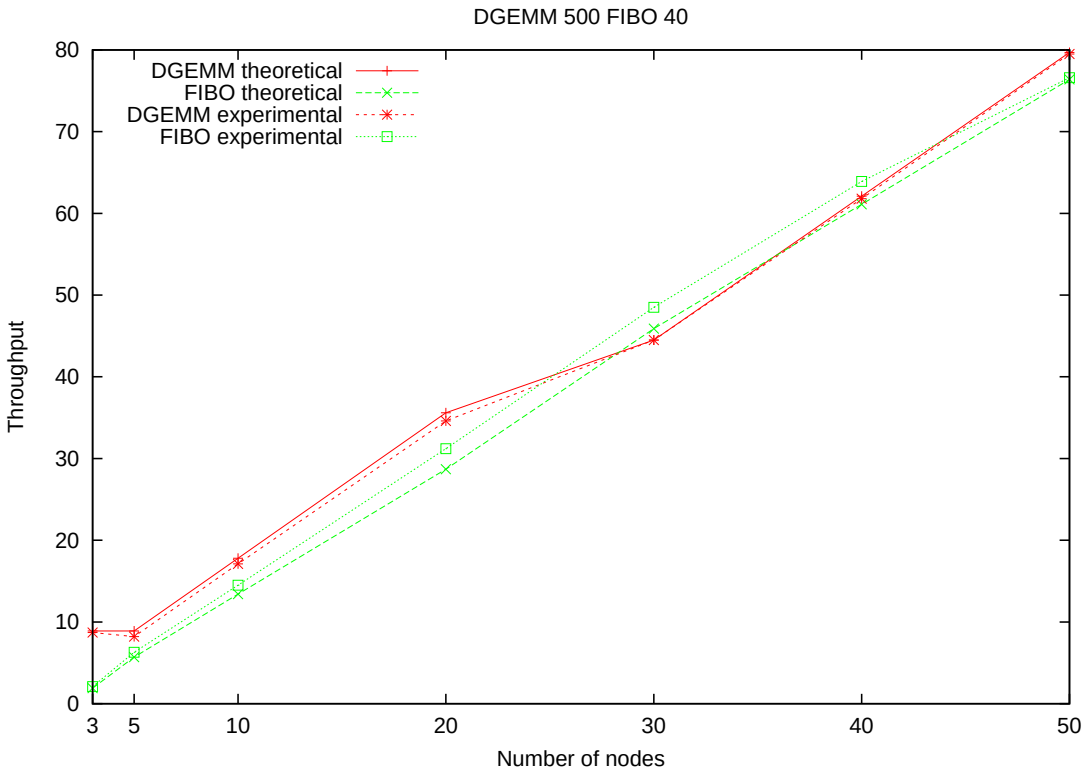

Figure 50: Comparison theoretical/experimental results, for `dgemm` 500 Fibonacci 40.

8.6 Clients parameters used for the experiments

In this section, we list the parameters used for the clients for each of the previously presented experiments. Table 13 presents the number of multithreaded clients used for each experiment: one client was launched on a physical node, and each client had a certain number of threads. Each thread continuously sent requests to the DIET hierarchy.

Experiment	Client	3	5	10	20	30	40	50
dgemm 100, Fibonacci 30	dgemm	2/20	4/20	4/40	6/32	-	-	-
	Fibonacci	2/25	4/25	4/50	4/50	-	-	-
dgemm 10, Fibonacci 20	dgemm	3/29	-	-	-	-	-	-
	Fibonacci	2/29	-	-	-	-	-	-
dgemm 10, Fibonacci 40	dgemm	2/35	2/35	2/35	2/35	2/35	2/35	2/35
	Fibonacci	1/10	1/30	1/80	1/80	1/80	1/10	1/10
dgemm 500, Fibonacci 20	dgemm	5/9	8/9	10/9	28/7	40/8	70/7	64/8
	Fibonacci	5/34	5/34	5/34	3/30	2/30	2/30	2/30
dgemm 500, Fibonacci 40	dgemm	5/9	8/9	10/9	10/10	10/10	10/12	10/12
	Fibonacci	1/10	1/30	4/20	4/20	4/20	5/20	5/20

Table 13: Number of nodes and threads per node used for the experiments. Format is the following: nodes/threads, '-' means that the experiment wasn't conducted as the hierarchy was the same as with fewer nodes.

8.7 Relevancy of creating new agent levels

In order to validate the relevancy of our algorithm to create the hierarchies, we compared the throughput obtained with our hierarchies, and the ones obtained with a star graph having exactly the same repartition of servers obtained with our algorithm. Thus, we aim at validating the fact that our algorithm sometimes creates several levels of agents whenever required. In fact in the previous experiments, this happens only for the following deployments:

- dgemm 100 Fibonacci 30 on 20 nodes: 1 MA and 3 LA
- dgemm 500 Fibonacci 20 on 30, 40 and 50 nodes: 1 MA and 2 LA

It is clear that with more nodes, the number of levels would increase.

Here are the results we obtained with such star graphs. We present the gains/losses obtained by the star graph deployments compared to the results obtained with the hierarchies our algorithm computed:

- dgemm **100 Fibonacci 30 on 20 nodes:** we obtained a throughput of 911.95 for dgemm and 779.07 for Fibonacci. Hence a loss respectively of 43.8% and 54.1%.
- dgemm **500 Fibonacci 20 on 30 nodes:** we obtained a throughput of 211.65 for dgemm and 3490.58 for Fibonacci. Hence a loss respectively of 5.5%, and 28%.
- dgemm **500 Fibonacci 20 on 40 nodes:** we obtained a throughput of 238.1 for dgemm and 2476.9 for Fibonacci. Hence a loss of 15.4% for dgemm, and a gain of 3.6% for Fibonacci.
- dgemm **500 Fibonacci 20 on 50 nodes:** we obtained a throughput of 184.68 for dgemm and 2494.0 for Fibonacci. Hence a loss respectively of 40.7% and 16.1%.

We can see from the above results that our algorithm creates new levels of agents whenever required: without the new agent levels the obtained throughput can be much lower. This is due to the fact that the MA becomes overloaded, and thus do not have enough time to schedule all requests.

8.8 Relevancy of the servers repartition

We also compared the throughputs obtained by our algorithm, and the ones by a balanced star graph (*i.e.,* a star graph where all services received the same number of servers). A balanced star graph is the naive approach that is generally used when the same throughput is requested for all services, which is what we aimed at in our experiments. Figures 51 to 55 present the comparison between the throughput obtained with both methods. As can be seen our algorithm gives better results: the throughput is better on all but one experiment (`dgemm` 500 in the `dgemm` 500 Fibonacci 40 experiment), no more resources than necessary are used (in Figure 51, no more than 20 nodes are required to obtain the best throughput, and in Figure 52 only 3 nodes). Our algorithm also tries to balance the $\frac{\rho_i}{\rho_i^*}$ ratio without degrading the performances, whereas with the balanced star graph adding more nodes can degrade the performances of both services.

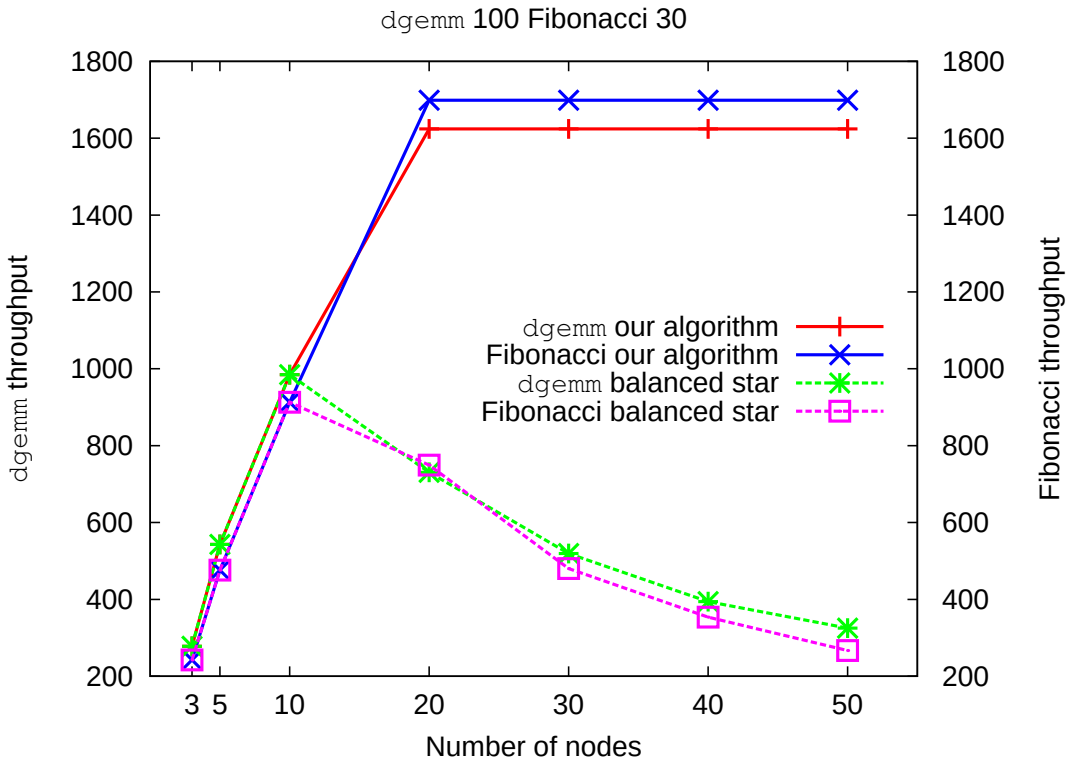

Figure 51: Comparison: our algorithm and balanced star for `dgemm` 100, Fibonacci 30.

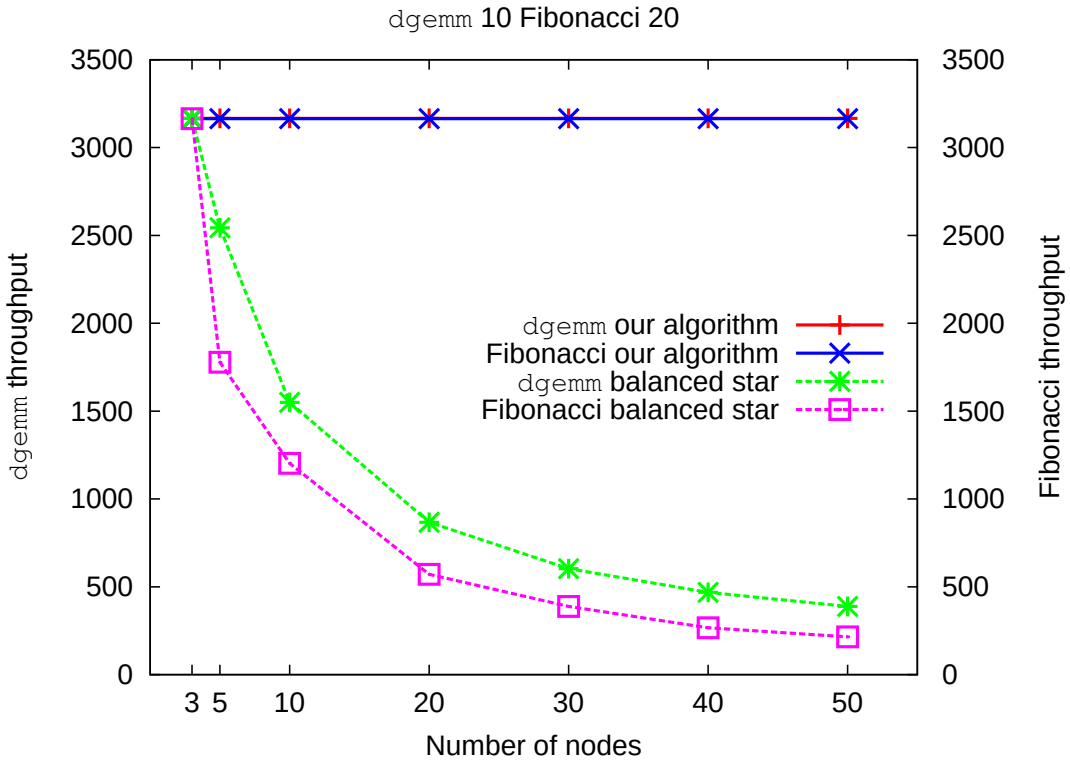

Figure 52: Comparison: our algorithm and balanced star for `dgemm` 10, Fibonacci 20.

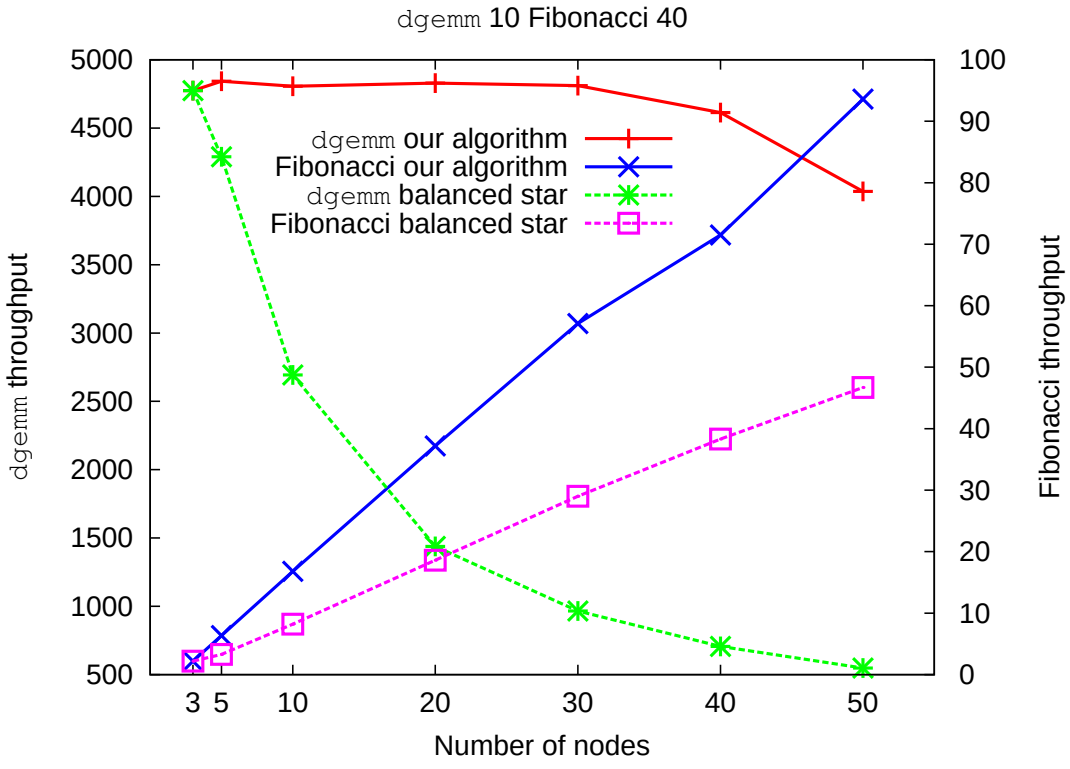

Figure 53: Comparison: our algorithm and balanced star for `dgemm` 10, Fibonacci 40.

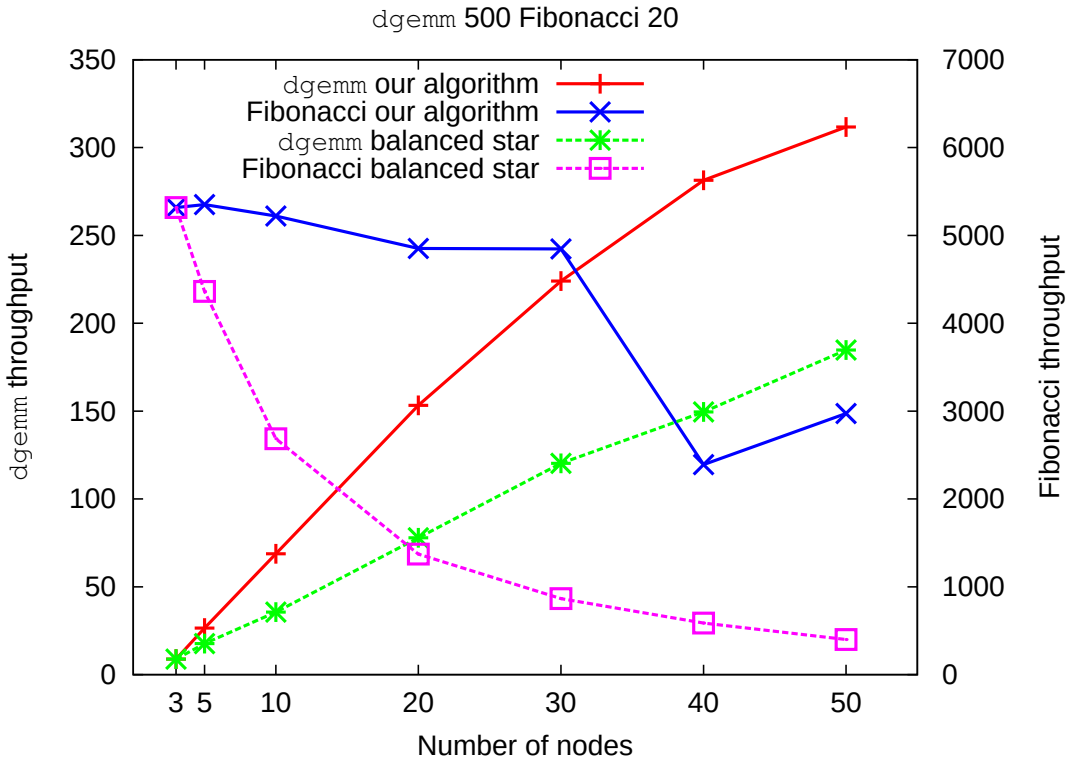

Figure 54: Comparison: our algorithm and balanced star for `dgemm` 500, Fibonacci 20.

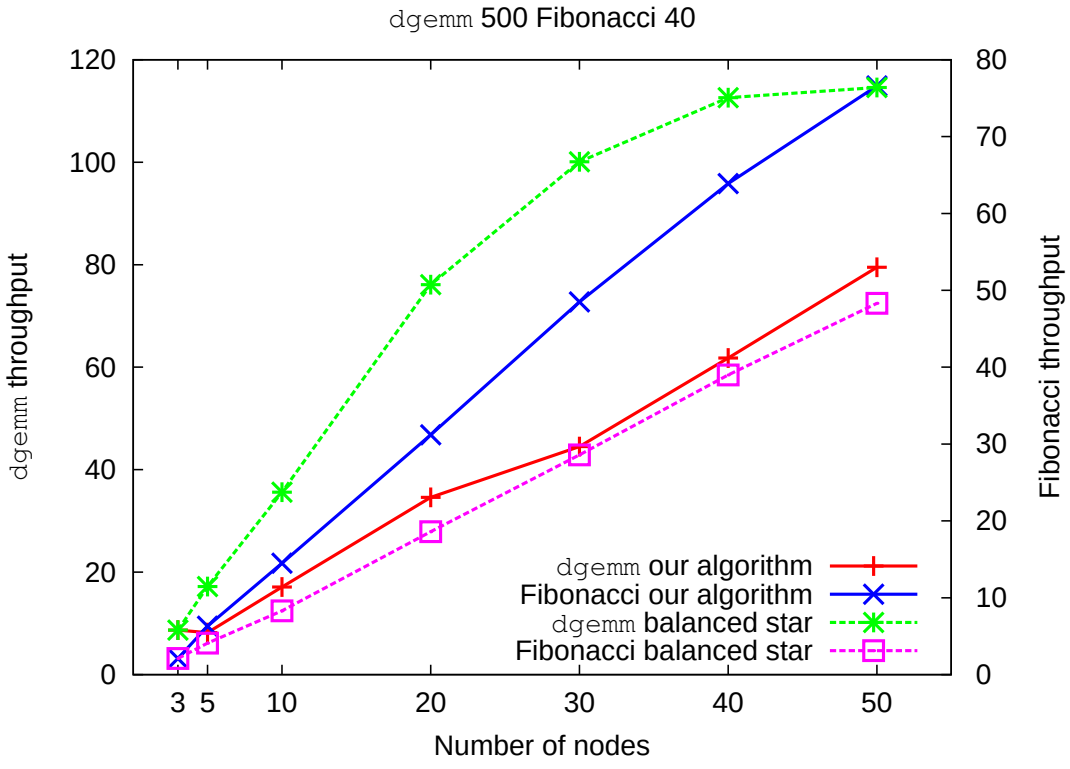

Figure 55: Comparison: our algorithm and balanced star for `dgemm` 500, Fibonacci 40.

8.9 Elements influencing the throughput

It isn't straightforward to obtain the best throughput out of a given hierarchy. Several parameters have to be taken into account in order to obtain the best throughput. We give here a list of some of the problems we encountered while trying to obtain the best throughput.

Bad scheduling The basic scheduling implemented within DIET relies on the time of the last request a SED has solved: when a request is sent down the hierarchy, each SED replies the *time since last solve*, *i.e.,* the time elapsed since the SED has solved a request. Hence, when multiple requests are sent in the hierarchy, the SED can reply the same value of *time since last solve* for multiple requests, and thus the same SED is likely to be chosen for a lot of requests. This of course, may overload some SED and do not give any load to the other SED. Hence, in order to cope with this, we need to activate an option of DIET which allows a client to make its own scheduling: the client keeps a local list of available SED for the service, and choose the server in a round robin fashion (which is what is implied by our model, as all servers are meant to have the same amount of load). Figures 56 and 57 presents the throughput obtained on a mixed hierarchy containing both Fibonacci and `dgemm` services, when using or not the scheduling at the client level. Exactly the same number of clients have been used for both experiments, and the same hierarchy has been used as well. We can see that the throughput is better when using the scheduling at the client level in two ways: first it is higher, and secondly it is less perturbed.

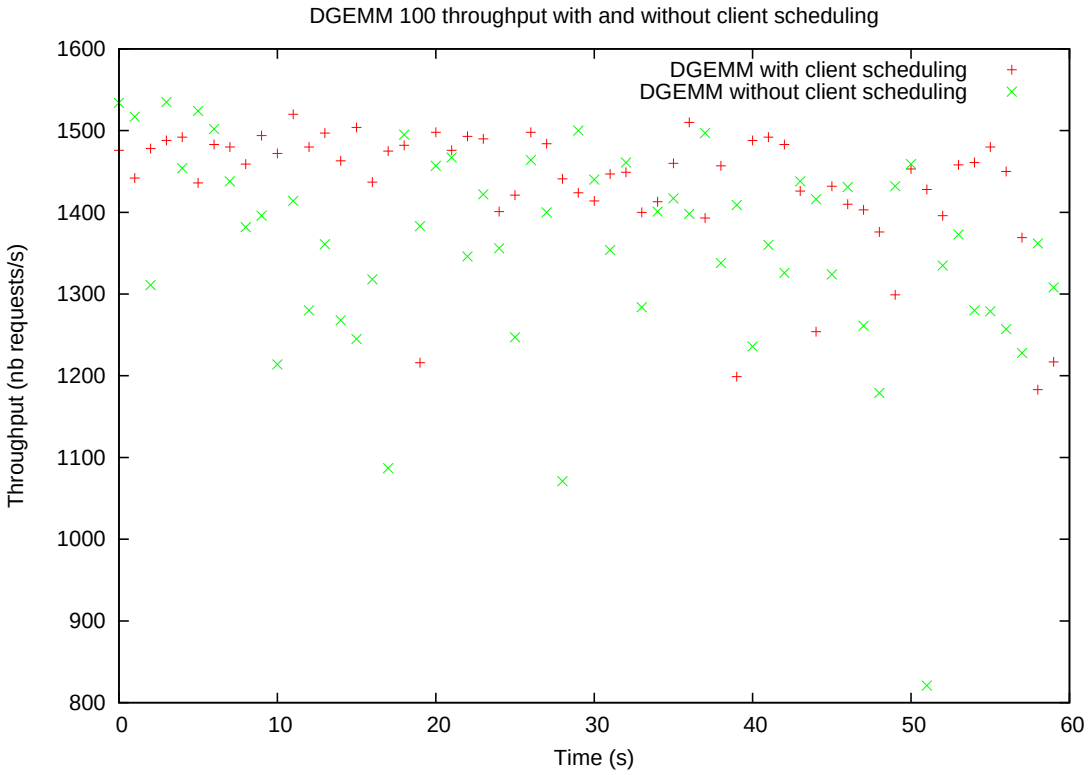

Figure 56: Throughput for `dgemm` 100 with and without client scheduling.

Logging Turning on or off the logging in DIET greatly influences the obtained throughput. Figure 58 presents the throughput obtained when turning on the logging on a platform on 30 nodes. Figure 59 presents the throughput obtained on the same platform, with the same number

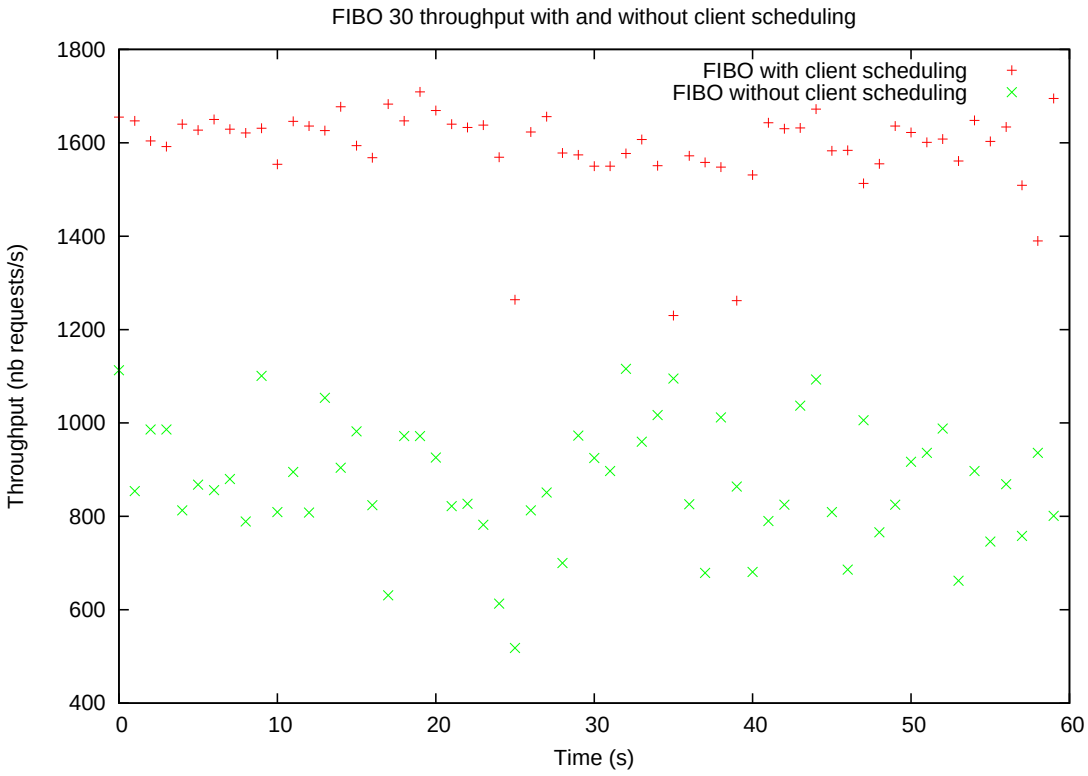

Figure 57: Throughput for Fibonacci 30 with and without client scheduling.

of clients, but when turning off the logging. As can be seen the throughput is greatly influenced by the logging facility. Indeed, several log messages are sent by each element of the hierarchy whenever a request is sent and solved. We observe a peak between 40 and 50 seconds on Figure 58, this behavior is typical from configurations with logging on, on some other experiments we obtained such peaks several times.

Number of clients The number of clients involved in the system also influences the throughput. The first point is of course the fact that too few clients won't send enough requests in the system to fully load it. Conversely, too many clients will overload the system as no queuing is done at the SeD level. Hence, in order to find the best throughput we need to find the correct number of clients. One more point has to be taken into account, is that having lots of clients with each sending a request at a time is more *expensive* than having less clients, but threaded clients that are able to send several requests in parallel. Figure 60 presents the throughput obtained for services `dgemm` 100 and Fibonacci 30 with 150 clients per service, all clients being deployed on 40 nodes. Figure 61 presents the throughput obtained on the same platform, but with threaded clients: 10 `dgemm` clients with 15 threads each, and 2 Fibonacci clients with 75 threads each, a node is used for each client. Each thread can send a single request in the system at a given time (*i.e.,* a new request can only be sent when the previous one has been solved). Hence, the two configurations are equivalent in terms of number of "clients", *i.e.,* the number of requests that can be sent to the hierarchy at a given time. What can be seen is that with threaded clients, the throughput is higher, and more stable.

omniORB configuration omniORB has by default some limits on the number of threads CORBA clients and servers can use at the same time. This can limit the throughput of the

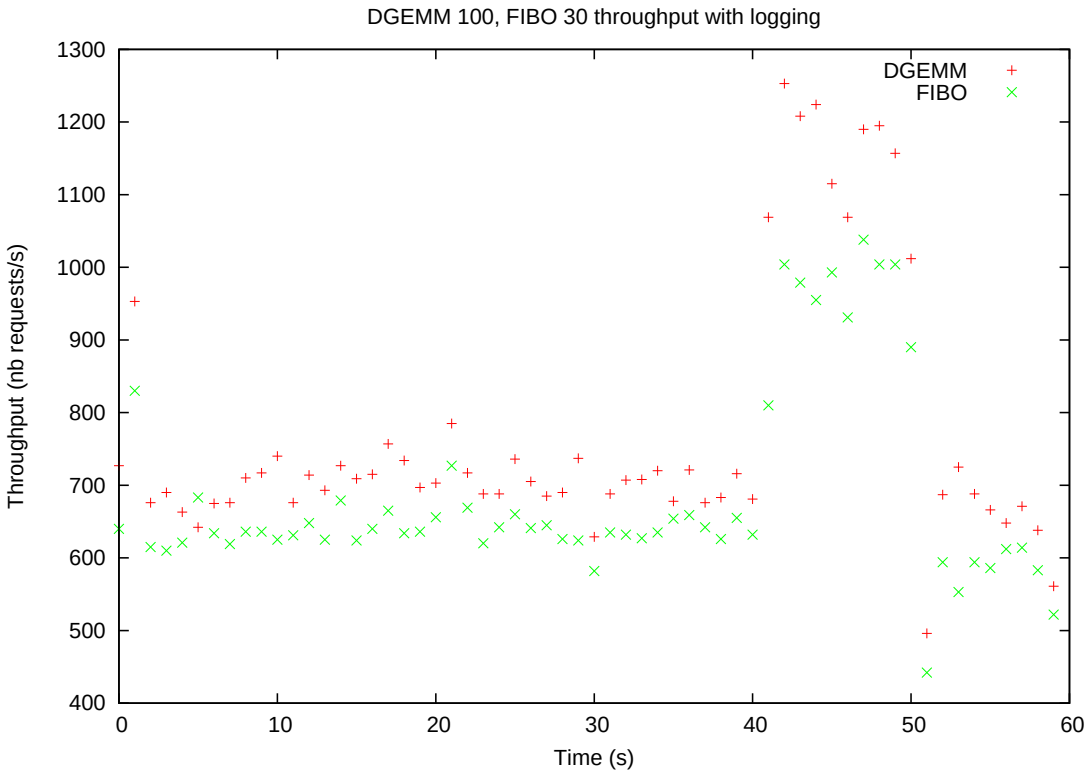

Figure 58: Throughput for `dgemm` 100, Fibonacci 30. With logging on.

platform, especially if "large" data transfers have to take place (this is the case for example with `dgemm` 500). Thus, in order to improve the throughput, we need to increase the number of threads clients and servers can use. This is done respectively with `maxGIOPConnectionPerServer` (default value: 5), and `maxServerThreadPoolSize` (default value: 100). We set those values respectively to 100 and 1000. With these values, we gained around 17% - 18% on the number of requests per seconds.

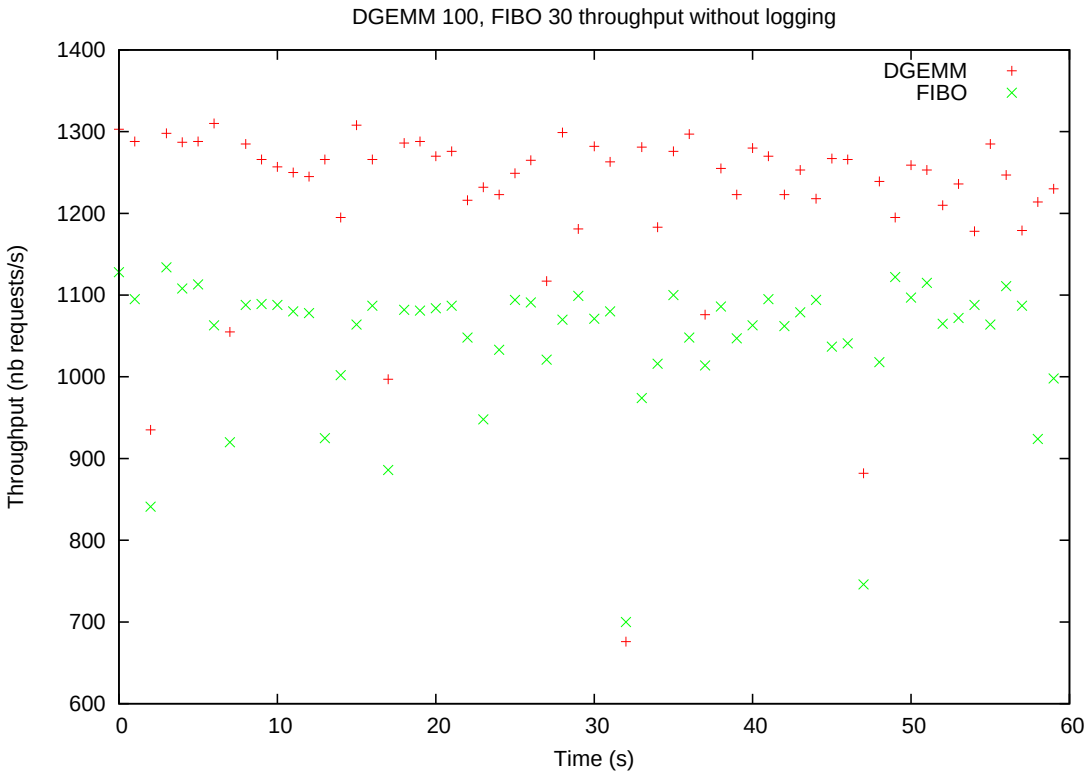

Figure 59: Throughput for `dgemm` 100, Fibonacci 30. With logging off.

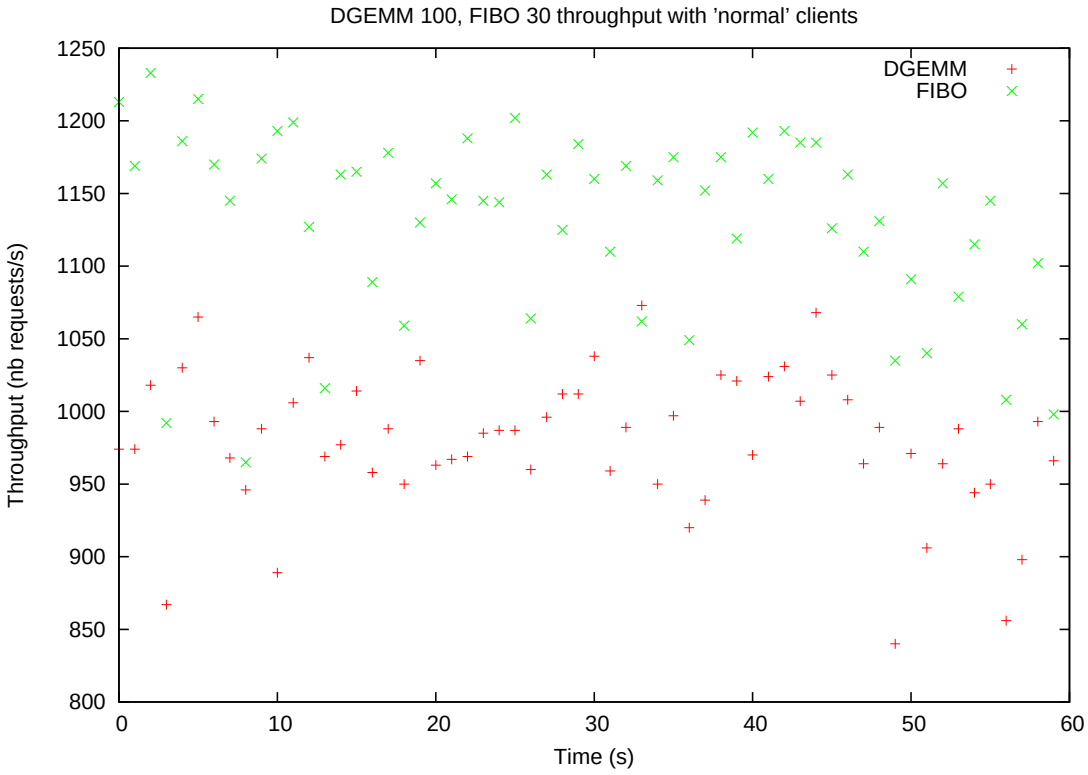

Figure 60: Throughput for `dgemm` 100, Fibonacci 30. 150 `dgemm` and FIBO "normal" clients on 40 different nodes.

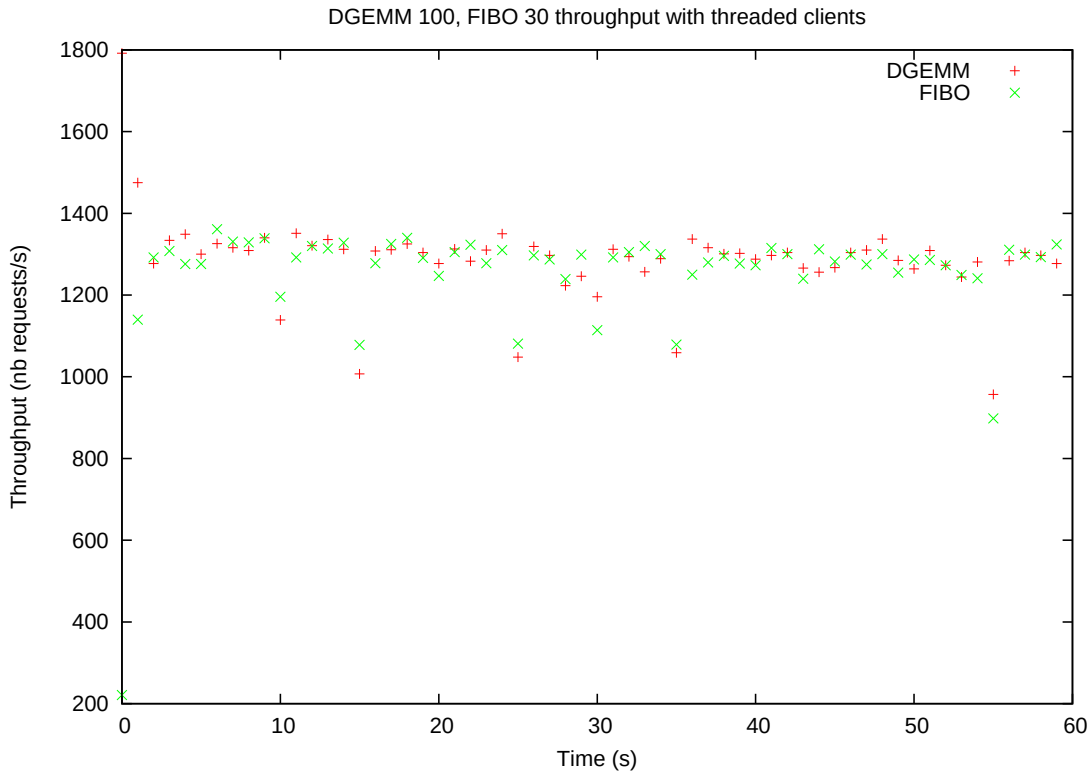

Figure 61: Throughput for `dgemm` 100, Fibonacci 30. 10 `dgemm` threaded clients (15 threads each), and 2 Fibonacci threaded clients (75 threads each). Each client is on a separate node.

9 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 MILP 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 and that our bottom-up algorithm provides excellent performances. We clearly showed that it adds new levels of agents whenever required, and that it outperforms the classical approach of deploying the middleware as a balanced star graph. Finally, the experiments allowed us to determine several elements that have a great impact on the throughput.

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.

10 Acknowledgment

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] Abelkader Amar, Raphaël Bolze, Yves Caniou, Eddy Caron, Benjamin Depardon, Jean-Sébastien Gay, Gaël Le Mahec, and David Loureiro. Tunable scheduling in a GridRPC framework. *Concurrency and Computation: Practice and Experience*, 20(9):1051–1069, 2008.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[3] Fran Berman, Geoffrey Fox, and Anthony J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[4] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frederic Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touché Irena. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[5] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel Depalma, and Suzy Temate. Autonomic management policy specification in tune. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663, New York, NY, USA, 2008. ACM.

[6] Yves Caniou, Eddy Caron, FrÃ©dÃ©ric Desprez, Hidemoto Nakada, Keith Seymour, and Yoshio Tanaka. *Recent Developments in Grid Technology and Applications*, chapter High performance GridRPC middleware. Nova Science Publishers, April 2009. To appear.

[7] E. Caron, P.K. Chouhan, and A. Legrand. Automatic deployment for hierarchical network enabled servers. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 109–, April 2004.

[8] Eddy Caron, Pushpinder Kaur Chouhan, and Holly Dail. GoDiet : A deployment tool for distributed middleware on grid'5000. In IEEE, editor, *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15.*, pages 1–8, Paris, France, June 19th 2006.

[9] Eddy Caron and Frédéric Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[10] Henri Casanova and Jack Dongarra. Netsolve: a network server for solving computational science problems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 40, Washington, DC, USA, 1996. IEEE Computer Society.

[11] P.K. Chouhan. *Automatic Deployment for Application Service Provider Environments*. PhD thesis, PhD thesis, Ecole Normale Supérieure de Lyon, 2006.

[12] Pushpinder Kaur Chouhan, Holly Dail, Eddy Caron, and Frédéric Vivien. Automatic middleware deployment planning on clusters. *Int. J. High Perform. Comput. Appl.*, 20(4):517–530, 2006.

[13] J. Dongarra et al. Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing*, 16(1):1–111, 2002.

[14] Areski Flissi and Philippe Merle. 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'06)*, volume 4279 of *Lecture Notes in Computer Science*, pages 1402–1411, Montpellier, France, nov 2006. Springer-Verlag.

[15] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[16] Sébastien Lacour, Christian Pérez, and Thierry Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, Seattle, WA, USA, November 2005. Springer-Verlag.

[17] J. P. Morrison, B. Clayton, D. A. Power, and A. Patil. Webcom-G: grid enabled metacomputing. *Neural, Parallel Sci. Comput.*, 12(3):419–438, 2004.

[18] Keith Seymour, Craig Lee, Frédéric Desprez, Hidemoto Nakada, and Yoshio Tanaka. The end-user and middleware apis for GridRPC. In *Workshop on Grid Application Programming Interfaces, In conjunction with GGF12, Brussels, Belgium*, 2004.

[19] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-g: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing*, 1(1):41–51, 03 2003.

[20] Yoshio Tanaka, Hiroshi Takemiya, Hidemoto Nakada, and Satoshi Sekiguchi. Design, implementation and performance evaluation of gridrpc programming middleware for a large-scale computational grid. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 298–305, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Asim YarKhan, Jack Dongarra, and Keith Seymour. Gridsolve: The evolution of a network enabled solver. *Grid-Based Problem Solving Environments*, pages 215–224, 2007.