Modelization and Performance Evaluation of the DIET Middleware

Eddy Caron, Benjamin Depardon and Frédéric Desprez

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

1 Introduction

Using distributed resources to solve large problems ranging from numerical simulations to life science is nowadays a common practice [1,2]. Several approaches exist for porting these applications to a distributed environment; examples include classic message-passing, batch processing, web portals or GridRPC systems [3]. 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 [4] 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 middleware only rely on basic hierarchies of elements, a star graph, such as Ninf-G [3] and NetSolve [6] and GridSolve [7]. Others, in order to divide the load at the agents level, can have a more complicated hierarchy shape: WebCom-G [8] and DIET [9]. In this latter case, a problem arises: what is the best shape for the hierarchy?

Modelization of middleware 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. The literature do not provide much papers on the modelization and evaluation of distributed middleware. In [10], Tanaka et al. present a performance evaluation of Ninf-G, however, no theoretical model is given. In [11] P.K. Chouhan presents a model for hierarchical middleware, and algorithms to deploy a hierarchy of schedulers on clusters and grid environments. She also compares 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 focus on one particular hierarchical NES: DIET 1 (Distributed Interactive Engineering Toolbox.) The DIET component architecture is structured hierarchically as a tree to obtain an improved scalability. DIET comprises several components. Clients that use DIET infrastructure to solve problems using a remote procedure call (RPC) approach. SeDs, or server daemons, act as service providers, exporting functionalities via a standardized computational service interface. 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 potentially several Local Agents (LA) (internal nodes.)

We first briefly present in Section 2 our model for hierarchical middleware, and a description of our algorithm to automatically build a hierarchy. Then, experimental results are presented in Section 3, before concluding.

1 http://graal.ens-lyon.fr/DIET
2 Model for hierarchical middleware

We do not detail the whole model and the algorithm in this paper. Instead we concentrate on the experimental results presented in the next section. Hence, we strongly encourage the reader to refer to our research report [12]. In this section, we give an overview of our model.

2.1 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: $V$ is the set of nodes, $E$ is the set of edges, $w$ is the processing power in $Mflops/s$, and all links have the same bandwidth $B$ in $Mb/s$. We do not take into account contentions in the network.

2.2 Server and agent model

We aim at deploying a hierarchy of agents and servers that is able to support several services. As we have multiple services in the hierarchy, our goal cannot be to maximize the global throughput of completed requests (number of completed requests per time unit) regardless of the kind of services. Hence, our goal is to obtain for each service a throughput such that all services receive almost the same obtained throughput to requested throughput ratio (of course we try to maximize this ratio), while having as few agents in the hierarchy as possible, so as not to use more resources than necessary.

Our model is divided into two parts: the servers and the agents models. Requests are sent from the root of the hierarchy to the servers where they evaluate the time they will take to execute the service. Then, replies are sent up the tree, and each agent aggregates the replies and selects the best server. In the end, the client receives the “best” server to execute the request. Then, it can directly contact this server to send the relevant data and execute the request. Our model takes into account the communications costs within the hierarchy (requests forwarding, and replies) as well as the computations costs at the agents level (time to treat a request, and time to aggregate the results) and the servers level (time to evaluate a request, and time to effectively execute a request).

2.3 Building a DIET hierarchy

Based on the server and agent models, we designed an algorithm to automatically build a DIET hierarchy. The algorithm is based on a bottom-up approach. We first distribute the available nodes between the different services. With this repartition we can compute the throughput the servers can offer. Then, with the remaining nodes, we try to build a hierarchy of agents that is able to support this load. The construction of the hierarchy is done level by level: we first try to add enough agents to support the servers level, i.e., allocate children to the agents (as the load of an agent is linear in the number of children), and verify that the computations and communications do not exceed the platform capacity $w$ and $B$ (resource constraints.) If only one agent is enough to support all the servers, then our hierarchy is built. Otherwise, we need to add at least a new level of agents on top of the previous one. We repeat this schema as long as we have enough nodes to build new levels, and for as long as we do not have only one agent at the top of the hierarchy.

If no suitable hierarchy can be found, we reduce the number of nodes allocated to the servers, which incidentally reduces the services throughput, and repeat the same schema, i.e., we try to build a hierarchy of agents with the remaining nodes. Note that the load of an agent depends on the number of children of each service type and the service throughputs. The agent level construction (i.e., nodes allocation, and children repartition) is done using a Mixed Integer Linear Program (MILP.) We do not detail the algorithm in this paper, instead we encourage the reader to refer to our research report [12].
3 Experimental Results

In this section, we present comparisons between the theoretical predictions and real experiments with the DIET middleware. We also show that our bottom-up approach based on an MILP to build the hierarchy gives really good results compared to classical deployments. We deployed two kinds of services: a matrix multiplication using \texttt{dgemm} [13] (for various matrix size), and the computation of the Fibonacci number using a naive algorithm. We will denote by \texttt{dgemm} $x$ the call to a \texttt{dgemm} service on matrices of size $x \times x$, and Fibonacci $x$ the call to a Fibonacci service to compute the Fibonacci number for $n = x$.

3.1 Theoretical model / experimental results comparison

In order to validate our model, we conducted experiments with DIET on the French experimental testbed Grid’5000 [14]. Using benchmarks made on the Sagittaire cluster, and our bottom-up algorithm, we generated for various combinations of \texttt{dgemm} and Fibonacci services, for platforms with 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 the 35 hierarchies for our experiments. Our goal here is to stress \texttt{DIET}, so we use relatively small services. We compared the throughput measured and predicted for various services combinations: \texttt{dgemm} 100 and Fibonacci 30 (medium size services), \texttt{dgemm} 10 and Fibonacci 20 (small size services), \texttt{dgemm} 10 and Fibonacci 40 (small size \texttt{dgemm}, large size Fibonacci), \texttt{dgemm} 500 and Fibonacci 20 (large size \texttt{dgemm}, small size Fibonacci), and \texttt{dgemm} 500 and Fibonacci 40 (large size \texttt{dgemm}, large size Fibonacci.) We tried to obtained the same throughput for each service: our target throughput is 10000 (which is higher than what we can expect on the platform.)

The experimental protocol is the following: we first deploy the hierarchy with \texttt{GoDIET} [15], then we run clients for both services and wait 10 seconds so that the load stabilizes. Finally, we measure the throughput during 60 seconds before killing all clients and the \texttt{DIET} hierarchy. Each client runs on a separate node, and sends one request at a time, once a request is completed, a new one is sent, this is repeated indefinitely.

We used a 79-nodes cluster: the Sagittaire cluster from the Lyon site. Each node has an AMD Opteron 250 CPU at 2.4GHz, with 2GB of memory. All those nodes are connected on a Gigabit Ethernet network supported by an Extreme Networks Blackdiamond 8810. The operating system deployed using \texttt{kadeploy3} on the Sagittaire nodes was a Debian 2.6.24.3 x86_64. Finally, the chosen CORBA implementation was omniORB 4.1.2, as \texttt{DIET} relies on CORBA for communications; and we used gcc 4.3.2. For the \texttt{dgemm} service, we relied on the Altas library [16].

Table 1 presents the relative error between the theoretical and experimental throughput: a dash in the table means that the algorithm returned the same hierarchy as with fewer nodes, and hence the results are the same. As can be seen, the experimental results closely follow what the model has predicted. Higher errors can be observed for really small services (\texttt{dgemm} 10 and Fibonacci 20.) This is due to the fact that really small services are harder to benchmark: the benchmarked server time to predict the execution time, and the time to execute a service are higher than what they should be.

3.2 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 1a to 1e 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, and no more resources than necessary are used (in Figure 1a, no more than 20 nodes are required to obtain the best throughput, and in Figure 1b only 3 nodes.)

The only experiment where the balanced star graph produced a higher throughput is for \texttt{dgemm} 500 in the \texttt{dgemm} 500 Fibonacci 40 experiment (see Figure 1e.) However, this is at the expense of the second service

\[\text{http://omniorb.sourceforge.net/}\]
which has a lower throughput. The \texttt{dgemm} throughput loss observable in Figure 1d is due to the load at the agent level (this phenomena is predicted by our model.)

Using our algorithm, we are able to increase the throughput to up to 700\% (see Figure 1b for instance.)

### 3.3 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: \texttt{dgemm} 100 Fibonacci 30 on 20 nodes (1 MA and 3 LA), and \texttt{dgemm} 500 Fibonacci 20 on 30, 40 and 50 nodes (1 MA and 2 LA.)

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:

- \texttt{dgemm 100 Fibonacci 30 on 20 nodes}: we obtained a throughput of 911.95 requests per seconds for \texttt{dgemm} and 779.07 requests per seconds for Fibonacci, compared to respectively 1624.4 and 1699.0 with our algorithm. Hence a loss of 43.8\% and 54.1\% respectively.

- \texttt{dgemm 500 Fibonacci 20 on 30 nodes}: we obtained a throughput of 211.65 requests per seconds for \texttt{dgemm} and 3490.58 requests per seconds for Fibonacci, compared to respectively 224.0 and 4845.85 with our algorithm. Hence a loss of 5.5\%, and 28\% respectively.

- \texttt{dgemm 500 Fibonacci 20 on 40 nodes}: we obtained a throughput of 238.1 requests per seconds for \texttt{dgemm} and 2476.9 requests per seconds for Fibonacci, compared to respectively 281.35 and 2391.6 with our algorithm. Hence a loss of 15.4\% for \texttt{dgemm}, and a gain of 3.6\% for Fibonacci.

- \texttt{dgemm 500 Fibonacci 20 on 50 nodes}: we obtained a throughput of 184.68 requests per seconds for \texttt{dgemm} and 2494.0 requests per seconds for Fibonacci, compared to respectively 311.7 and 2973.3 with our algorithm. Hence a loss of 40.7\% and 16.1\% respectively.

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.

### 4 Conclusion and future work

We presented in this paper a model for hierarchical middleware based on a tree structure, and briefly described our algorithm for automatically finding a suitable hierarchy. Our objective, when deploying several services in the hierarchy is to provide the best provided to required throughput ratio.
We conducted several experiments on the Grid'5000 platform with the DIET middleware. The results show that the experimental results closely follow what the model predicted, and that our bottom-up algorithm provides excellent performances. The experiments showed that our algorithm adds new levels of agents whenever required, and that it clearly outperforms the classical approach of deploying the middleware as a balanced star graph.

Several points now need to be taken into account. So far we only studied the case of a fully homogeneous platform. Even though our model and algorithm can easily be extended to computation heterogeneous /
communication homogeneous platforms; the case of a fully heterogeneous platform is much more complicated. The model also needs to be extended with client/server communications, but for this we need to know where the clients are, whereas in this paper we considered that we did not know anything on the clients, this assumption do not seem to impact our model. Finally, as the MA can become a bottleneck, and as DIET can have several hierarchies interconnected at the MA level, we could also study this latter case.

5 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