A Peer-to-Peer Extension of Network-Enabled Server Systems*

Eddy Caron, Frédéric Desprez, Cédric Tedeschi
LIP Laboratory / GRAAL Project
UMR CNRS, ENS Lyon, INRIA, UCBL, France
46 Allée d'Italie, 69364 Lyon Cedex 07, France
FirstName.Name@ens-lyon.fr

Franck Petit
LaRIA Laboratory
University of Picardie - Jules Vernes, France
5, rue du Moulin Neuf, 80000 Amiens, France
Franck.Petit@laria.u-picardie.fr

Abstract

DIET (Distributed Interactive Engineering Toolbox) is a set of hierarchical components to design Network Enabled Server (NES) systems. In these systems, clients ask to agents (discovery and scheduling components) to find servers able to solve their problem using some performance metrics and information about the location of data already on the network. Today's NES middleware, in which agents are statically connected and potentially bottlenecks, don't cope with the dynamic and heterogeneous nature of future grid environments. In this paper, we present the design, the implementation and the experimentation of the first architecture extending traditional NES systems with an unstructured peer-to-peer network dynamically connecting distributed agents, to provide to clients an entry point to servers geographically distributed. Our implementation is based on DIET and the JXTA toolbox. Different algorithms have been implemented and experimented over the VTHD network which connects several supercomputers in different research institutes through a high-speed network, showing the scalability of the system.

1 Introduction

The use of distributed resources available through high-speed networks has recently gained a wide interest. So called grids [2, 8] are now widely available for many applications around the world. The number of resources made available grows every day and the scalability of middleware platforms becomes an important issue. Many research projects have produced middleware to cope with heterogeneity and dynamicity of the target platforms like Globus [9] or Condor [16] while trying to hide the complexity of the platform as much as possible to the user.

Among them, one simple, yet performant, approach consists in using servers available in different administrative domains through the classical client-server or RPC[1] paradigm. Network Enabled Servers [6] implement this model also called GridRPC [15] (like DIET, NetSolve, or Ninf). Clients submit computation requests to a scheduler whose goal is to find a server available on the grid, able to solve the client's problem. The agent applies scheduling mechanisms to balance the work among the servers and a list of available servers is sent back to the client which is in turn able to send a request and its data to the server to solve the given problem. Due to the growth of the network bandwidth and the reduction of the latency, small computation requests can now be sent to servers available on the grid. One important issue is now the scalability of the middleware itself. The agent's work (service discovery and scheduling) should be made scalable.

This has been first addressed by distributing the agent itself. We designed DIET [3, 4], a set of hierarchical elements to build applications using the GridRPC paradigm. This middleware is able to find an appropriate server according to the information given in the client's request (problem to be solved, size of the data involved), the performance of the target platform (servers load, memory available, communication performance), and the availability of data stored during previous computations. The scheduler is distributed in a hierarchy of agents. Agents are statically connected and can become bottlenecks as the number of requests grows.

Emerging Peer-to-Peer technologies allow the search of resources in large scale environments. Iamnitchi and Foster suggested in [10] that grids provide the infrastructure for sharing resources, but do not cope with the dynamic nature of today's large scale platforms, and so one would take advantage to adopt Peer-to-Peer tools. To date, very few grid middleware have implemented Peer-to-Peer technologies, and, to our knowledge, no NES systems, whose static nature still hinders its worldwide deployment, have used the

*This work was supported in part by the ACI GRID (ASP), RNTL (GASP), and the RNRT (VTHD++) of the french department of research.

[1] Remote Procedure Call

Peer-to-Peer technology.

In this paper, we present a new architecture extending a Network Enabled Server system connecting agents in a Peer-to-Peer fashion, thus dynamically aggregating servers known by several agents geographically distributed. Two algorithms have been implemented on top of this architecture for the propagation of clients' requests through the network of agents: an asynchronous star graph traversal and an expanded version of the Propagation of Information with Feedback (PIF) scheme dynamically adapting the logical connections of the Peer-to-Peer network to the heterogeneous load of links. Our architecture shows a good scalability as the number of agents and the number of requests grow and a good adaptability as some links become overloaded.

In following section, we give a brief overview of DIET. Then, in Section 3, we describe the design and the implementation of DIET$_J$, the new Peer-to-Peer extension of DIET. In Section 4, we describe the algorithms we used to validate our system. Finally, before a conclusion and some hints for future work, we give the validation of our architecture based on experiments on a grid connecting clusters through a Wide Area Network.

2 DIET overview

The DIET architecture has first been designed following a hierarchical approach [3]. Thus it provides a good scalability and can take into account the physical network constraints. In this section, we describe the DIET static hierarchical architecture.

DIET is based on several elements. First a **Client** is an application that uses DIET to solve problems in a RPC mode. Different kinds of clients should be able to connect to DIET from a web page, a Problem Solving Environment such as Matlab or Scilab, or from a program written in C or Fortran. Traditionally a centralized device in other NES systems such as NetSolve or Ninf, the DIET scheduler is scattered across a hierarchy of *Agents*. Figure 1 shows such a hierarchy.

2.1 Scheduling Agents

A **Master Agent** (MA) is the entry point of the DIET environment and thus receives computation requests from clients. These requests refer to some problems that can be solved by registered servers. These problems can be listed on a reference web page. A client can be connected to a MA by a specific name server or a web page which stores the various MA locations. Then the MA collects computation abilities from the servers and chooses the best one according to some scheduling heuristics (dead-line scheduling, shortest completion time first, minimization of the requests throughput, ...). A reference to the server chosen is sent back to the client.

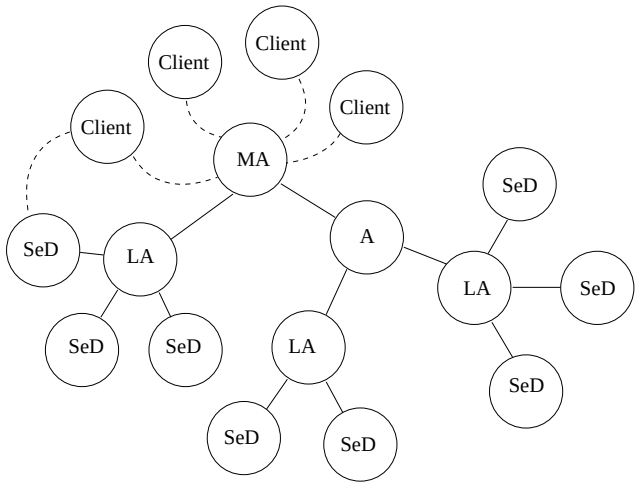

Figure 1. DIET hierarchical organization.

A Master Agent relies on a hierarchy of agents to gather information and scheduling decisions. An **Agent** aims at transmitting requests and information between MAs and LAs. A **Local Agent** (LA) aims at transmitting requests and information between Agents and several servers. The information stored on an Agent is the list of servers registered on Local Agents of its subtree, the problems they are able to solve and information about the data distributed in this subtree. Depending on the underlying network topology, Agents may be deployed between the MA and the LAs. The scheduling and the gathering of information is thus distributed in the tree.

2.2 Server Daemons

Computations are done by servers (both sequential and parallel) in front of which we have **Server Daemons (SeD)**. A SeD encapsulates a computational server, typically on the entry point of a parallel supercomputer. The information stored on a SeD is a list of data available on its server (with their distribution and the way to access them), the list of problems that can be solved on it, and all information concerning its load (memory and/or number of resources available, ...). A SeD registers to a Local Agent that becomes its parent and declares the problems it can solve to it. A SeD can give performance predictions for a given problem using the performance evaluation module (FAST [12]).

2.3 Limits of DIET

Such a hierarchy has three major drawbacks.

1. **Static configuration.** Such static hierarchies do not cope with the dynamicity of nodes at large scale, resulting in difficulties to deploy such hierarchies on large grids. As a consequence, most of those hierarchies are not deployed among more than one administrative domain. Moreover, the clients are given an entry point statically. One needs for the client is to dynamically choose its *best* MA considering metrics such as latency.

2. **Master Agent bottleneck.** The hierarchy has a unique entry point (the MA) for every clients. This involves a probability of finding a bottleneck growing with the number of requests submitted by clients.

3. **Service availability.** Real life use cases show that services are quite often deployed among only one hierarchy for many reasons (data locality, security, ...). One key purpose of computational grids is to make services available for clients anywhere in the wide area. So there is a strong need for making services available for clients, that does not necessarily know the entry point of the hierarchy providing the requested service.

3 DIET$_J$: A Peer-to-Peer Extension of DIET

The aim of DIET$_J$ is to dynamically connect together geographically distributed DIET hierarchies to gather services on-demand and improve the scalability of service discovery. This new architecture addresses the limits described before and have the following properties:

Dynamically connecting hierarchies for scalability To increase the scalability of DIET over the grid, we dynamically build a *multi-hierarchy* by connecting the entry points of the hierarchies (Master Agents) together. Note that the multi-hierarchy is built on-demand by a Master Agent only if it fails to find the service requested by a client inside its own tree. The clients are now given the ability to discover at time of requesting a service, one or several Master Agents, and thus connect the server with the best latency/locality.

Balancing the load among the Master Agents The entry point for each client being dynamically chosen, the bottleneck on the previous unique Master Agent is now avoided. Master Agents are connected in a pure unstructured Peer-to-Peer fashion (without any mechanism of maintenance, routing, or group membership).

Gathering services at large scale Whereas DIET hierarchies were unable to communicate together in the first DIET version, services are now gathered when processing a request thus providing to clients a front door to resources of hierarchies put in common in a transparent way.

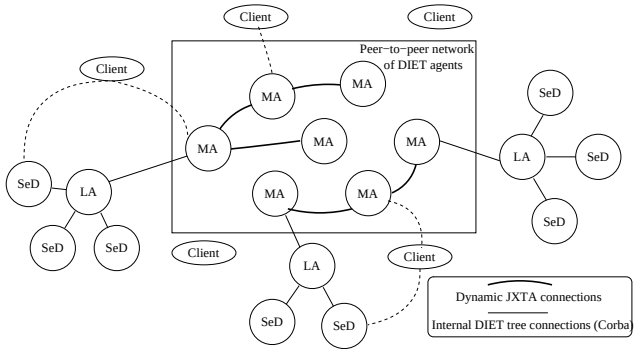

Figure 2. DIET$_J$ architecture.

3.1 The JXTA Project

JXTA is an open-source project initiated by Sun Microsystems that defines a rich set of protocols for building Peer-to-Peer (P2P) applications on top of the physical network. The basic logic entity of the JXTA virtual network is the peer. Each peer is a potential client of any other peer, while being a potential server for any peer. JXTA provides three types of peers. The **edge peer** is the basic peer on top of which users provide their applications on the JXTA virtual network. The **rendezvous peer** is used to resolve the discovery queries submitted by edge peers. The **relay peer** acts as a logical router passing through network protections (such as firewalls and NAT technologies.)

Each JXTA entity (peers, pipes, services) is uniquely identified by an **advertisement**. Thanks to the rendezvous peers allowing discovery of these advertisements, JXTA offers a dynamic discovery of any JXTA entity, thus allowing any peer to dynamically address any other peer on the JXTA virtual network. JXTA offers three communication layers:

The endpoint service. First level of abstraction, the endpoint service provides a unidirectional and unreliable communication between two edge peers.

The pipe service. Built on top of the endpoint service, the pipe is a virtual end-to-end communication channel. A pipe can be of *unicast* type, i.e., binding two peers in a unidirectional way, or of *propagate* type, allowing a peer to send messages to multiple recipients. Based on results presented in [11], we believe JXTA pipes offer the right trade-off between transparency and performance for our architecture. Our implementation is based on JXTA 2.3 which minimizes the latency of the JXTA pipes, according to [11].

JXTA sockets. Final level of abstraction offered by JXTA, the JXTA sockets are reliable, bidirectional and more transparent communication channels.

3.2 DIET$_J$ Architecture

The DIET$_J$ architecture, shown in Figure 2, connects several DIET hierarchies by a JXTA network of their root, i.e., Master Agents. The Master Agent's internal architecture, shown on Figure 3 is divided into three parts.

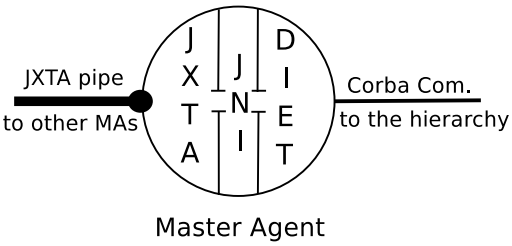

Figure 3. Master Agent Internal Architecture.

- **The JXTA part.** The JXTA part of the Master Agent is a peer on the JXTA virtual network. This part is its connection point to other Master Agents. This part is a java bytecode.
- **The DIET part.** The DIET part is the traditional DIET Master Agent, root of a DIET hierarchy of Agents and Local Agents, allowing the discovery of servers that registered to this hierarchy. This part is based on libraries generated from the DIET C code.
- **The interface.** To cooperate, Java (JXTA native language) and C (DIET native language) need an interface. We use the JNI technology allowing to call C functions from a Java program, and the data conversion between the two languages.

3.3 The Multi Master Agent system

One Multi-MA is composed of all MAs running at a given time over the network and reachable from a first MA. The MA is able to dynamically connect these other MAs. Each MA is known on the JXTA network by an advertisement with a name common to all of them ("DIET_MA") that is published at the beginning of its life. This advertisement is published with a short lifetime to avoid Clients and other MAs to try to bind an already stopped MA, and thus easily take into account the dynamicity of the platform.

At their loading time, the JXTA part loads the DIET part via JNI, periodically re-publishes its advertisement, while waiting for requests. When receiving a client's request, the DIET part submits the request to its own hierarchy. If the submission to the DIET hierarchy retrieves no SeD's references with the required service, the JXTA part builds a multi-hierarchy by discovering other MAs (thanks to their JXTA advertisements) and propagating the request to them. When the JXTA part has received responses from all other MAs or when a timeout is reached, the response is sent back to the client which is not aware that a multi-hierarchy has been temporarily built.

3.4 Dynamic Connections

Dynamic connections between the Master Agents allow to transparently perform the service discovery in a dynamic large scale multi-hierarchy, using JXTA advertisements. The communication between the agents inside one hierarchy are still static as we believe that small hierarchies are installed within each administrative domain. At the local level, performances are not very variable and new elements are not frequently added.

4 Traversing the Multi-Hierarchy

We now discuss approaches and algorithms implemented for propagating the clients' requests and gather information about servers of several hierarchies.

4.1 Approach

Discovering the MAs, then discovering the servers

It is important to note that the multi-hierarchy construction is divided into two parts.

1. **peers discovery**. The first step aims at discovering MAs reachable on the network, thanks to the JXTA discovery process. But once a peer has been discovered, i.e., got its advertisement (mainly containing its name and its address, under the shape of an input pipe advertisement), you still need to establish a connection with it.
2. **service discovery**. The second step consists in exploring the multi-hierarchy composed of the MAs discovered in the first step, looking for the requested service inside the DIET hierarchies.

JXTA discovery mechanisms

JXTA 2.x provides a hybrid mechanism based on DHT [17] and random walk to achieve the discovery of advertisements (e.g., advertisement named "DIET_MA"). Again, we choose

Algorithm 4.1 Asynchronous PIF for arbitrary networks.

Constants:
$IdSet$: set of IDs;
$Neigh$: set of outgoing links or neighbors Ids;

Variables:
$Ack[IdSet]$ of subset of $Neigh$;
$Father[IdSet]$ of $Neigh \cup \perp$, **initially** $\perp$;
$q,\ q' \in Neigh$;
$ListServer$: list of server names;

Macro $Sync$

$Ack[Id] := Neigh \setminus \{q\}$;
if $Ack[Id] = \emptyset$ **then**
  if $MyId = Id$ **then**
    SEND $ListServer$ TO the Application Layer;
  else
    SEND $(Id, ListServer)$ TO $Father[Id]$;
  endif
endif

Upon **RECEIPT** of $req$the Application Layer
$Father[MyId] := \top$;
$ListServer := \emptyset$;
$Ack[MyId] := Neigh$;
$\forall q' \in Ack[MyId]$: **SEND** $(MyId, req)$ TO q';

Upon **RECEIPT** of $(Id, req)q$
if $Father[Id] = \perp$ **then**
  $ListServer := MakeList(Id, req)$;
  $Father[Id] := q$;
  $Ack[Id] := Neigh \setminus \{q\}$;
  if $Ack[Id] = \emptyset$ **then**
    SEND $(Id, ListServer)$ TO $Father[Id]$;
  else
    $\forall q' \in Ack[Id]$: **SEND** (Id, req) TO q';
  endif
else
  $Sync$;
endif

Upon **RECEIPT** of $(Id, LS)q$
$ListServer := ListServer \cup LS$;
$Sync$;

not to use the hybrid DHT mechanism to avoid its maintenance overhead, its lack of exhaustiveness (when failing retrieving the advertisement by the hash function, it uses a less-efficient "walking" method) and cope with the unstructured fashion of the multi-hierarchy designed.

Thus, we first use the JXTA discovery mecanism based on flooding among the peers. Once the MA's references obtained, an algorithm optimizing the traversal of the multi-hierarchy (the MAs graph) is used to connect MAs together and propagate the request through the multi-hierarchy.

4.2 Implementations

The propagation of the request in the DIET$_J$ multi-hierarchy (i.e., between the MAs) has been implemented with two algorithms.

Propagation as an Asynchronous Star Graph Traversal

The propagation has first been implemented as an intuitive asynchronous star graph traversal. One MA r that found no SeD providing the service requested by a client in its own hierarchy discovers other MAs with the JXTA discovery process. Then, it forwards the request in an asynchronous way to all the MA previously discovered, using a simple JXTA multicast pipe instruction. On receipt of the forwarded request, each MA collects the servers able to solve the problem in its own hierarchy, and sends back the response to r that collects and merges responses to create the final response message, that it sends back to the client. Using this first algorithm, the propagation systematically builds a star graph, the MA initiating the propagation being the root of the star graph. In the following of the paper, this algorithm is called the "STAR$_{async}$" algorithm.

Propagation as an Expanded Version of the Asynchronous PIF Scheme

The propagation has also been implemented using an asynchronous version of the *Propagation of Information with Feedback* scheme (PIF) described in Algorithm 4.1, to have an unstructured, efficient and adaptative muti-hierarchy traversal. A complete description of the basic PIF can be found in [7, 14]. Figure 4 describes a scenario of propagation in a DIET multi-hierarchy, applying the two following phases:

The **Broadcast phase**: The MA that received the request from the client (and is unable to find a server providing the requested service) initiates the wave and so is the root r. As in the STAR$_{async}$ algorithm, it forwards the request to all other MAs it has previously discovered. Let M_r be the set of discovered MAs. r then waits for responses of MAs in M_r. Unlike in the STAR$_{async}$ algorithm, a MA $m \in M_r$ receiving a forwarded request checks if it has already received it. If yes, it ignores it. Otherwise, the MA that sent the request to it becomes its parent. Of course, m collects the servers to solve the problem described in the request in its hierarchy. Finally, m propagates the request in its turn to the MAs in M_r (that it knows from its parent), except those that are by the way taken by the request to reach m from r. Thus a time optimal tree rooted at r is built.

The **Feedback phase**: r waits for the responses of M_r during a finite time using a timeout. The MAs in M_r send the enabled servers found in their hierarchy back to their parent and, when receiving a response from a child, send the response to their own parent.

PIF scenario

Let us have a look at the Figure 4. The MA that received the request from the client found no SeD providing the requested service in its own hierarchy. After having discovered others MA, it initiates the wave (1). Some MAs have received the propagated request. They forward it in their turn, and initiate the asynchronous feedback phase (2). All MAs have received the request. A spanning tree is built. The feedback phase goes on and ends. The connections opened during this phase depends on the traffic load encountered during the broadcast phase, allowing an optimal feedback phase (3).

Quick analysis of the PIF scheme

Let us call this algorithm "PIF$_{async}$". Note that the PIF$_{async}$ builds an *on-demand optimal tree* for a given root for each request, thus balancing the load among the MAs graph as the number of requests increases and also avoiding overloaded links. It was shown in [13, 14] that in asynchronous environments, the PIF scheme is the fastest possible to reach every network nodes, messages following the fastest links during the broadcast phase. In other words, the dynamic tree built during the propagation is time optimal. It provides fault tolerance, because of the several retransmissions achieved by this algorithm and thus the several attempts to reach each MA. The number of messages can be very important ($O(n^2)$ in the worst case). Note that algorithm STAR$_{async}$ also provides a PIF scheme. However, it is not an adapting scheme, messages always following the same links, ignoring their heterogeneity and communication load.

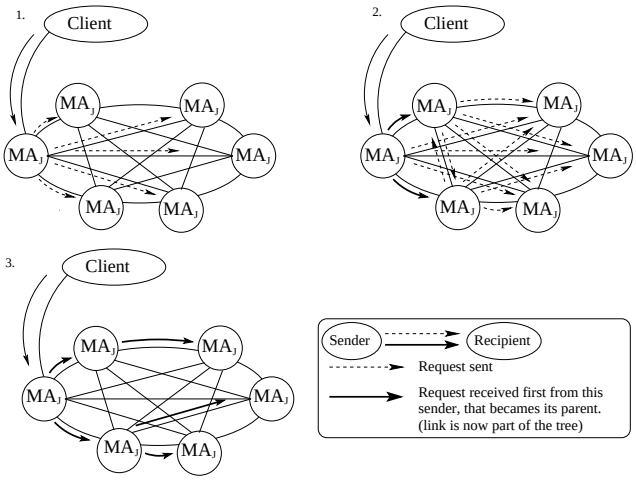

Figure 4. Propagation scenario in a DIET multi-hierarchy.

5 Experimenting DIET$_J$: Performance Results

In this section, we discuss experimental results of the implementation of DIET$_J$ with the algorithms previously described.

5.1 Experimental Platform

Our experimental platform relies on the VTHD network, a wide area network, which connects several clusters through links whose bandwidth is 2.5 Gb/s. The clusters used are equipped with Intel quadri-processors Xeon 2.4 GHz and Intel bi-processors Xeon 2.8 GHz. Only one MA runs per node and one client sends one or multiple requests to MAs. Based on our previous experiments inside one unique hierarchy [5], where we showed that a unique DIET Master Agent is able to have hundreds of servers in its own hierarchy and remain efficient with a very high number of simultaneous requests, we here experiment connections of the MAs graph without underlying hierarchies.

5.2 Experiments with Homogeneous Network Performance

We started our experiments with a low and homogeneous traffic load, by varying the number of MAs in order to estimate the response time of both algorithms.

Figures 5 and 6 show the time to initiate the propagation and to receive all the responses, using the STAR$_{async}$ and the PIF$_{async}$, on several VTHD clusters themselves connected by the VTHD WAN links, up to 32 Master Agents running at the same time. Note that on a homogeneous network, our architecture shows good results, in regard of the JXTA overhead, aggregating servers' references of 32 Master Agents in less than one second. Note that, under these conditions, most of the trees obtained with the PIF$_{async}$ are stars, the initial propagation from the root reaching other nodes first. It is interesting to see that using the PIF$_{async}$ involves quite few time overhead, in regard of the higher number of messages it generates.

5.3 Requests Flooding

Then we experimented both algorithms by varying the requests frequency still on a homogeneous network. Fifteen Master Agents are deployed for these experiments.

Figure 7 shows the impact of processing multiple requests at the same time inside the graph of MAs, with the same root for every requests. As expected, much better results are obtained by propagating requests with the PIF$_{async}$. Using the STAR$_{async}$, physical routes used by the JXTA pipes are mostly the same for every requests,

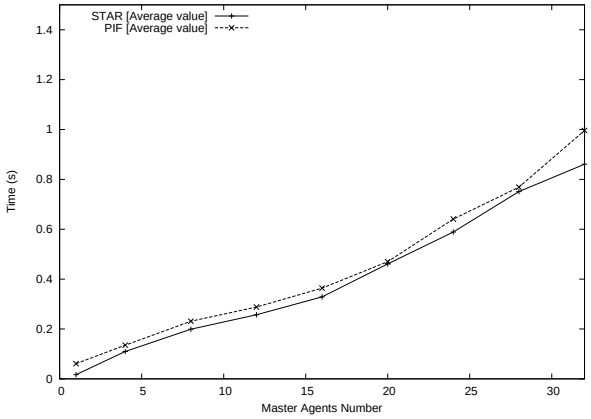

Figure 5. Evaluating the cost of using the PIF$_{async}$ compared to the STAR$_{async}$ on one cluster.

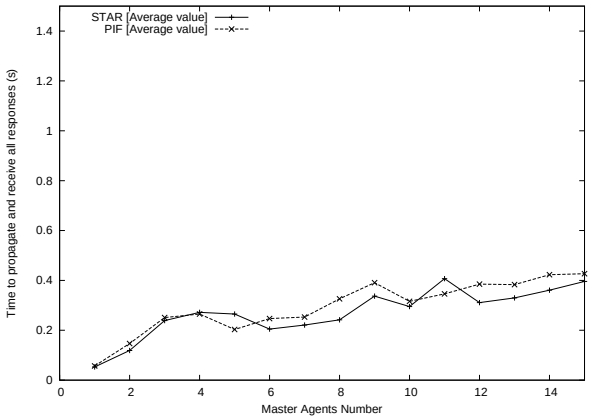

Figure 6. Evaluating the cost of using the PIF$_{async}$ compared to the STAR$_{async}$ on two clusters (located in different cities) connected through the VTHD network.

strongly increasing the load on these links. We believe the STAR algorithm performs so poorly because of the high cost of resolving JXTA pipes, especially when the same links are stressed. Using the PIF$_{async}$, logical path (and physical routes underneath) used during the feedback phase depends on the load of the links during the broadcast phase. Each propagation builds a spanning tree during the broadcast phase minimizing the communication time that is then used during the feedback phase. The traffic is globally more distributed and bottlenecks are avoided. The response time remains stable when the frequency of sending becomes quite high.

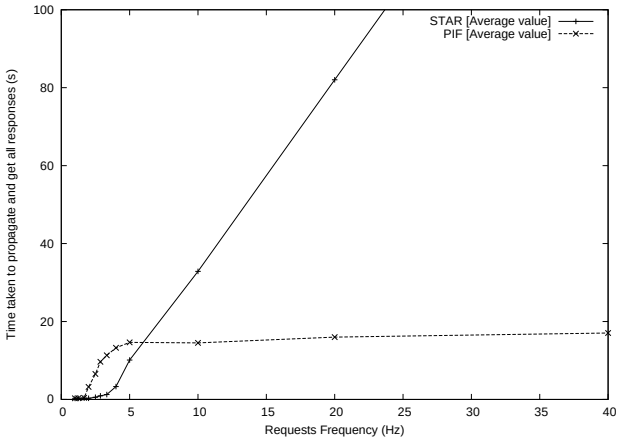

Figure 7. Sending 10 requests at various frequencies.

5.4 Experiments on Overloaded Links

Finally, we experimented our architecture on an overloaded network (thirteen MAs running). We simulated a loaded traffic with loops of `scp` commands, especially around the MA initiating the propagation. Figure 8 shows the performance of each algorithm, varying the number of saturated links around the MA initiating the propagation. The STAR$_{async}$ always uses the saturated links, during both the broadcast and the feedback phases, increasing again the load on the links. Using the PIF$_{async}$ algorithm allows to avoid most of the traffic around the root by building optimal trees for each request. The feedback phase uses the least overloaded route that has been discovered at broadcast time, for each request. The response time given by the PIF$_{async}$ is more stable than the STAR$_{async}$ one when the number of overloaded links increases, offering response time similar to those obtained under homogeneous conditions.

6 Conclusion and Future Work

In this paper, we have presented DIET$_J$, the first extension of a Network-Enabled Server system taking into account the dynamic and heterogeneous nature of today's platforms on which grids will inexorably take place.

The use of JXTA and the asynchronous PIF algorithm shows an efficient on-demand discovery of available servers at large scale. Our first experimental results demonstrate that our architecture remain efficient, providing quick response time, even when the network becomes overloaded.

Our future work will consist in validating our architecture at larger scale using larger clusters connected through Wide Area Networks (within the Grid5000 project [1]) and to implement other Peer-to-Peer algorithms in DIET$_J$. We

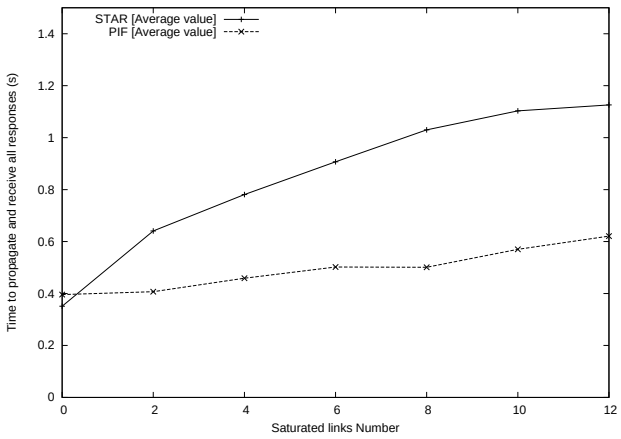

Figure 8. Experimenting the PIF$_{async}$ and the STAR$_{async}$ on a network with overloaded links.

are currently extending our architecture to other NES systems (NetSolve, Ninf). Each approach using its own internal mechanisms, there's a strong need to design bridges between them, to offer to Grid's users a tool hiding this heterogeneity.

References

[1] Grid 5000 project. http://www.grid5000.org.

[2] F. Berman, G.C. Fox, and A.J.H. Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003.

[3] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 2005. To appear.

[4] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In *Proc. of EuroPar 2002*, Paderborn, Germany, 2002.

[5] E. Caron, F. Desprez, F. Petit, and V. Villain. A Hierarchical Resource Reservation Algorithm for Network Enabled Servers. In *IPDPS'03. The 17th International Parallel and Distributed Processing Symposium*, Nice - France, April 2003.

[6] H. Casanova, S. Matsuoka, and J. Dongarra. Network-Enabled Server Systems: Deploying Scientific Simulations on the Grid. In *High Performance Computing Symposium (HPC'01)*, Seattle, Washington (USA), April 2001.

[7] E.J.H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. on Software Engineering*, SE-8:391–401, 1982.

[8] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[9] Ian Foster and Carl Kesselman. The globus toolkit. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 259–278. Morgan Kaufmann, San Francisco, CA, 1999. Chap. 11.

[10] Adriana Iamnitchi and Ian Foster. On fully decentralized resource discovery in grid environments. In *International Workshop on Grid Computing*, Denver, Colorado, 2001. IEEE.

[11] M. Jan and D.A. Noblet. Performance Evaluation of JXTA Communication Layers. Technical Report RR-5530, INRIA, IRISA, Rennes, France, october 2004.

[12] M. Quinson. Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment. In *International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02)*, April 15-19 2002.

[13] Danny Raz and Yuval Shavitt. New Models and Algorithms for Programmable Networks. *Computer Networks*, 38(3):311–326, 2002.

[14] A. Segall. Distributed Network Protocols. *IEEE Trans. on Information Theory*, IT-29:23–35, 1983.

[15] K. Seymour, C. Lee, F. Desprez, H. Nakada, and Y. Tanaka. The End-User and Middleware APIs for GridRPC. In *Workshop on Grid Application Programming Interfaces, In conjunction with GGF12*, Brussels, Belgium, September 2004.

[16] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[17] B. Traversat, M. Abdelaziz, and E. Pouyoul. A Loosely-Consistent DHT Rendezvous Walker. Technical report, Sun Microsystems, Inc, March 2003.