Resource Localization Using Peer-To-Peer Technology for Network Enabled Server.

Eddy Caron, Frédéric Desprez, Franck Petit, Cédric Tedeschi

Laboratoire de l'Informatique du Parallélisme

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

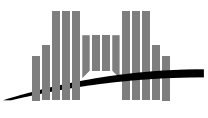
Resource Localization Using Peer-To-Peer Technology for Network Enabled Servers

Eddy Caron ,
Frédéric Desprez ,
Franck Petit ,
Cédric Tedeschi

Decembre 2004

Research Report N° 2004-55

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`



Resource Localization Using Peer-To-Peer Technology for Network Enabled Servers

Eddy Caron , Frédéric Desprez , Franck Petit , Cédric Tedeschi

Decembre 2004

Abstract

DIET (Distributed Interactive Engineering Toolbox) is a set of hierarchical components to design Network Enabled Server systems. These systems are built upon servers managed through distributed scheduling agents for a better scalability. Clients ask to these scheduling components to find servers available (using some performance metrics and information about the location of data already on the network). Our target architecture is the grid which is highly heterogeneous and dynamic. Clients, servers, and schedulers are better connected in a dynamic (or peer-to-peer) fashion.

One critical issue to be solved is the localization of resources on the grid. In this paper, we present the use of an asynchronous version of the Propagate Information with Feedback algorithm to discover computation resources in Network Enabled Servers with distributed schedulers and arbitrary networks. Resource discovery uses peer-to-peer connections between components. Our implementation is based on JXTA from Sun Microsystems and DIET developed in the GRAAL team from INRIA.

The algorithm and its implementation are discussed and performance results to show the benefit of this approach are given from experiments over the VTHD network which connects several supercomputers in different research institutes through a high-speed network.

Keywords: Resource Localization, P2P, Grid computing

Résumé

DIET (Distributed Interactive Engineering Toolbox) est un ensemble de composants hiérarchiques pour la conception de serveurs de calcul distants. Ces systèmes sont construits sur des serveurs gérés par des agents ou ordonnanceurs distribués pour une meilleure extensibilité. Les clients demandent à ces ordonnanceurs de trouver les serveurs disponibles (recherche prenant en compte les mesures de performance des serveurs et les informations concernant le temps d'accès aux données qui peuvent être déjà présentes sur le réseau). L'architecture cible est la grille qui est fortement hétérogène et dynamique. Les clients, les serveurs et les ordonnanceurs sont connectés via une approche dynamique utilisant une technologie pair-à-pair. Dans ce cadre, un problème critique à résoudre concerne la localisation des ressources sur la grille. Dans cet article, nous appliquons une version de l'algorithme de propagation d'informations avec retours pour découvrir les ressources de calcul pour des serveurs de calcul distants avec des ordonnanceurs distribués dans un réseau quelconque. Cette découverte de ressources utilise les connections pair-à-pairs entre les éléments. Nos expérimentations seront basées sur JXTA de Sun Microsystems et DIET de l'équipe GRAAL de l'INRIA. Les résultats obtenus et analysés montrant l'intérêt de cette approche ont été réalisés sur le réseau expérimental VTHD de France Telecom qui connecte les différents sites INRIA via un réseau à très haut débit.

Mots-clés: Localisation de ressource, P2P, Calcul sur la grille

1 Introduction

The use of distributed resources available through high-speed networks has recently gained a wide interest. So called grids [2, 11] 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 isssue. Many research projects have produced middlewares to cope with heterogeneity and dynamicity of the target platforms [9, 12, 13, 23] 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 [7, 16, 17] implement this model also called GridRPC [20]. Clients submit computation requests to a scheduler which goal is to find a server available on the grid. Scheduling has to be applied 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 the data and the request to solve a 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 issue can now be the scalability of the middleware itself. The scheduling can be made scalable by distributing the scheduler.

We thus designed DIET [5], a set of hierarchical components 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 using several hierarchies connected either statically or dynamically (in a peer-to-peer fashion). More information about DIET is given in Section 2.

One important issue to be solved is the discovery of resources available at a given time for a request. In this paper, we use the Propagate Information with Feedback (so called PIF) algorithm to gather information about resource available for a given request. Following our results for static hierarchical networks presented in [6], we extend our platform and our resource discovery scheme for arbitrary (and dynamic) networks. We have designed an asynchronous PIF and implemented it in a dynamic version of our software using peer-to-peer connection between hierarchies.

In a first section, we give an overview of DIET providing only static connections between components. In a second section, we introduce a version of DIET allowing dynamic connections between the hierarchies. Then, in Section 4 we describe the algorithms used to find resources in DIET. These algorithms are based on the Propagate Information with Feedback Algorithm (PIF). Finally, before a conclusion and some hints for our future work, we give the validation of our implementations on a grid based on clusters connected through a Wide Area Network.

2 DIET Overview

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

2.1 General Architecture

DIET is based on severals components. 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 PSE such as Matlab or Scilab, or from a program written in C or Fortran. The scheduler is scattered across a hierarchy of *Agents*. This hierarchy is made of one **Master Agent (MA)**, several **Agents (A)**, and **Local Agents (LA)**. Figure 1 shows a hierarchy built upon several DIET components.

[1] Remote Procedure Call

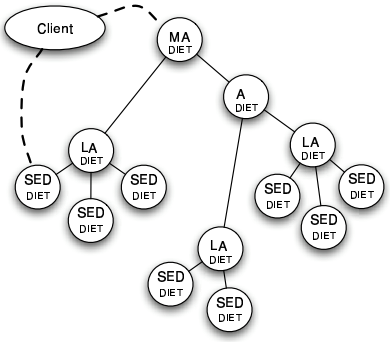


Figure 1: DIET hierarchical organization.

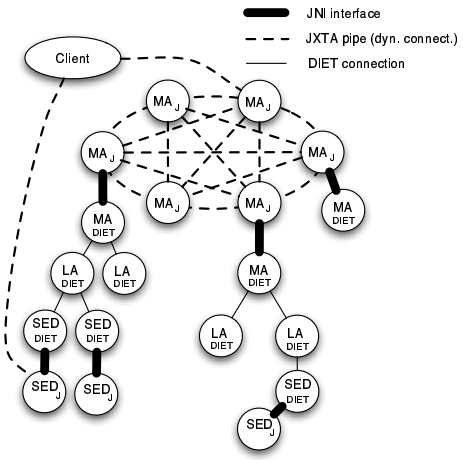


Figure 2: DIET$_J$ architecture

2.2 Scheduling Agents

A **Master Agent** is the entry point of our environment and thus it receives computation requests from clients attached to it. These requests refer to some DIET 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.

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 requests and the number of servers that can solve a given problem and information about the data distributed in this subtree. Depending on the underlying network topology, a hierarchy of Agents may be deployed between an MA and the LAs. The scheduling and the gathering of information is thus distributed in the tree.

2.3 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. For instance it can be located 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 declares the problems it can solve to its parent LA. A SeD can give performance predictions for a given problem using the performance evaluation module (FAST [18]).

2.4 Static Connections

In this version of DIET, connections between components are static. They are configured at the initialization of the platform or during the registration of new components. Deployment algorithms [4] are used to start components (and map them on the target architecture) depending of the capacity of the target platform and the history of previous requests.

3 Dynamic Connections between Clients, Schedulers and Servers

Offering a Multi-hierarchy level increases the scalability inside DIET. It can be achieved in different ways. In this section, we discuss a new and dynamic version, implemented on the JXTA virtual network [21].

3.1 The JXTA Project

JXTA is an open-source project initiated by Sun Microsystems, Inc. that defines a rich set of protocols for building peer-to-Peer (P2P) applications on top of the physical network architecture. The logic basic entity of the JXTA virtual network is the peer. Each peer is a potential client of any other peer, while offering services itself. A peer can be of one of the following types:

The edge peer. The edge peer is the basic peer on top of which users provide their services on the JXTA virtual network.

The rendezvous peer. The rendezvous peer is used to resolve the discovery queries submitted by edge peers.

The relay peer. 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. Due 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 unidirectionnal 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.

JXTA sockets. Final level of abstraction offered by JXTA, the JXTA sockets add reliability, bidirectionnality and transparency of communications.

3.2 P2P DIET Extension: DIET$_J$

DIET$_J$ is an extension of DIET allowing a DIET multi-hierarchy to be deployed (i.e., to have several static DIET hierarchies connected together) and providing to clients an entry point to computation resources put in common, in a dynamic and transparent way.

3.3 DIET$_J$ Architecture

The DIET$_J$ architecture shown in Figure 2 is divided into two parts. The JXTA part including the MA$_J$, the SeD$_J$ and the Client$_J$. All these components are peers on the JXTA virtual network and communicate together through it. The interface part: Java (JXTA native language) and C++(DIET native language) must cooperate. The technology used is JNI [15] that allows a Java program to call functions written in C++. We now introduce the different components built on top of JXTA.

The Client$_J$. This component communicates only with JXTA components and is written in Java. Its behavior is as follows: It launches a new peer, looks for the advertisement of a MA$_J$ (named "DIET_MA") and, once found, binds it via a JXTA pipe. Then, it encapsulates

the description of the problem to be solved in a JXTA message and sends it through the pipe. Then it waits for the MA_J's response. Once the response is received, it extracts from it the reference of the $SeD(s)_J$ able to solve the problem. It connects to one available SeD_J referenced in the response through a pipe and sends to it the problem to be solved by the SeD_{DIET}, encapsulated in a JXTA message. It waits for the SeD_J response message. Finally, the $Client_J$ extracts the result of the computation from the response.

The SeD_J. The purpose of the SeD_J is to allow the $Clients_J$ to send computation requests including data needed for the computation to the SeD_{DIET}. It also allows to pass through firewalls if any between the $Client_J$ and the SeD_J. Its behavior is as follows: It launches a new peer, loads the SeD_{DIET}, and waits for $Client_J$'s request messages. When a JXTA message is received, the SeD_J extracts the problem and the data to be computed from it and calls the SeD_{DIET} to solve the problem. The result returned by the SeD_{DIET} is then encapsulated in a JXTA message and sent back to the $Client_J$.

The Multi-MA_J. It is composed of all MAs_J running at a given time. The MA_J is able to dynamically connect to a $Client_J$ and to the other MAs_J running at the same time. Each MA_J is known on the JXTA network by an advertisement with a name common to all MAs_J ("DIET_MA") that is published at the beginning of its life. It is important to point out that this advertisement is published with a short lifetime in order to avoid $Clients_J$ or other MAs_J to try to connect to an already stopped MA_J, and thus to take into account the changes of the platform without failure. The behavior of the MA_J is as follows: It launches a new peer, loads the MA_{DIET}, launches a thread that regularly re-publishes the advertisement, and waits for request messages. When the MA_J receives a request coming from a client, it submits the problem description to DIET via the MA_{DIET}. If the submission to the DIET hierarchy returns a failure (no SeD found), the request is propagated to the other available MAs_J. When the MA_J has received the responses of all the other MAs_J, the responses obtained are encapsulated in a JXTA message and sent back to the $Client_J$.

3.4 Dynamic Connections

Important dynamic connections between the client and the Master Agents, between the client and the SeD, and between the Master Agents themselves allow to perform the resource localization in a dynamic multi-hierarchy, using JXTA pipes advertisements. The communication between other components of DIET (inside one hierarchy) are still static as we believe that small hierarchies will be installed within each administrative domain. At the local level, performances are not so fluctuent and new components are not frequently added.

According to [14], the throughput of the JXTA pipes is higher than the JXTA sockets one and close to endpoint service one. In addition, we believe pipes offer the right level of transparency we need for our architecture. Another evaluation in [14] shows that using the 2.3 version of JXTA minimizes the latency of the JXTA pipes. So, our implementation of the JXTA elements of $DIET_J$ is based on this version of JXTA and uses pipes as a communication tool.

4 Distributed Resource Localization Algorithm

4.1 Previous Work using the PIF Algorithm in Tree Networks

In [6], we have designed the hierarchical and distributed scheduler used in DIET. The scheduler is implemented using the *Propagation of Information with Feedback* scheme (the PIF scheme, for short) for (spanning) tree networks. The concept of PIF (also called *wave propagation*) was independently introduced by Chang [8] and Segall [19]. Figure 3 presents the behavior of the PIF algorithm in a fixed tree network. A process (sometime referred to a node) p initiates the first phase of the wave: the propagation or broadcast phase (phase (*ii*) in Figure 3). Every process, upon receiving the first broadcast message, chooses the sender of this message as its parent in the

PIF wave, and forwards the wave to its neighbors except its parent. When a process receives a feedback (acknowledgment) message from all its children with respect to the current PIF wave, it sends a feedback message to its parent(phase (*iii*) in Figure 3). So, eventually, the feedback phase ends at p (phase (*iv*) in Figure 3).

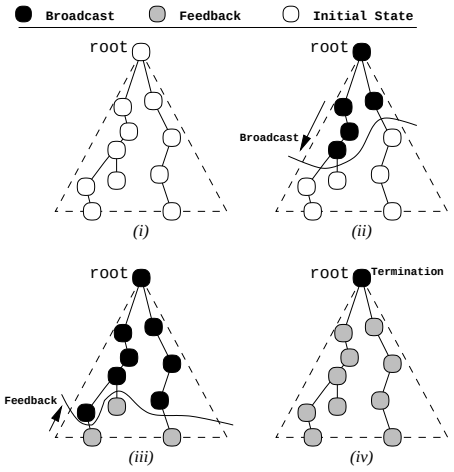


Figure 3: The PIF scheme.

The PIF scheme was also studied in the area of *self-stabilization*, e.g., in tree networks [3], in arbitrary networks [10]. In the next section, we briefly describe the *asynchronous PIF* scheme for arbitrary networks used in this paper.

4.2 Description of the Asynchronous PIF Algorithm in Arbitrary Networks

The PIF scheme can be informally described as follows: any process can be an initiator of a PIF wave, and several waves may run simultaneously. We assume that each wave is initiated by a process, called the root r. Informally, each wave is made of two consecutive phases (*broadcast* and *feedback* phases).

Starting from an initial configuration where no message has yet been broadcast, any process, thereafter referred as the root r, initiates the *broadcast* phase. The neighbors of r participate in this phase by forwarding the broadcast message, if possible. It is not possible for a process p to broadcast the message if all its neighbors have received the message from some other neighbor. So, step by step, a spanning tree rooted at r is dynamically built during the broadcast phase. Let us call this dynamic tree the $B\text{-}tree_r$. The processes which are not able to broadcast the message further are the leaves of $B\text{-}tree_r$. Once the broadcast phase reaches the leaf processes of $B\text{-}tree_r$, they notify to their parent in $B\text{-}tree_r$ of the termination of the broadcast phase by initiating the *feedback* phase. Then, the feedback phase eventually reaches the root r. This completes the current PIF Cycle. The complete formal algorithm is described in Algorithm 5.1.

We now describe how the PIF scheme is used to locate a DIET server in the network that will be able to solve a request according to some scheduling scheme.

5 Implementing the Traversal of the Multi-Hierarchy

5.1 Approaches

The search of the services inside the DIET_J multi-hierarchy is divided into two parts : the peer discovery, i.e., the discovery of the peers providing the DIET submission (the MAs_J) to be able to

establish the dynamic connection between hierarchies, and the service discovery, i.e., the traversal of the MAs_J discovered, each MA_J relying on a hierarchy possibly providing the service requested by the $Client_J$.

The JXTA 2.0 DHT Algorithm for Service Discovery

JXTA 2.0 provides a mechanism based on DHT [22] to achieve the discovery of advertisements stored on edge peers. An edge peer that publishes an advertisement sends it to a rendezvous peer (RDV) it knows, that stores an index of the advertisements, not the advertisements themselves. Moreover, each RDV maintains a RDV Peers View (RPV) referencing the other RDV it itself knows. The RDVs use a hash function to forward their indexes to other RDV. When an edge peer is looking for a given advertisement (for instance called "DIET_MA") it sends a discovery query to its known RDV that uses the hash function to retrieve the RDV caching the index of the wanted advertisement. Then,this RDV forwards the query on its turn to the edge peer providing the advertisement. At last, this edge peer sends the advertisement to the requesting peer.

One approach consists in integrating the DIET request submission inside the peer discovery. The discovery query request could contain not only the name of the peer service ("DIET_MA") but also the name of the service requested by the $Client_J$ (for example "MatrixSUM"). The JXTA DHT could submit the problem to its DIET subtree and send the result of the submission to the MA_J that forwarded the request, instead of its advertisement.

However, such an approach suppose to insert a part of DIET code inside the JXTA DHT discovery code, and thus to depend on the version of JXTA used. Moreover, the results of this algorithm are not meant to be exhaustive, and if the hash function fails retrieving the advertisement, the standard "walking" propagation method is used through the rendezvous peers.

To be JXTA-independant and to avoid the lack of the JXTA discovery based on DHT, we choose to first discover every MAs_J reachable from the MA_J contacted by the $Client_J$, and in a second time, to propagate the $Client_J$'s request using an algorithm optimizing the traversal of the MA_J graph.

5.2 Implementations

As said before, we already use the PIF for trees to collect the enabled servers in the DIET hierarchies. Now, the propagation of the request in the $DIET_J$ multi-hierarchy (i.e., between the tops of the hierarchies) has been implemented with two algorithms.

Propagation as an Asynchronous Star Graph Traversal

The propagation has been first implemented as an asynchronous star graph traversal. The MA_J that obtains a failure when submitting a request received from a $Client_J$ is the root of the star and propagates the request to all the other MA_J running when the failure occurs (its children) in 1 logical step in an asynchronous way: a listener collects the responses and merge the results in order to create the final response message to be sent to the client, when responses of all of its children have been received and processed. Every child of the root, when receiving the request, collects the servers able to solve the problem in their subtree, and send back the response to the root. The result of the propagation is systematically a star graph with MA initiating the propagation as root of the star. Let us call this algorithm "$STAR_{async}$" in the following of this paper.

Propagation as an Expanded Version of the Asynchronous PIF Scheme

The PIF is thus used at two levels: Each MA at the top of a hierarchy collects the enabled servers in its subtree using the PIF for trees, when receiving the request, as described in [6]. Then, if a failure is encountered, the request is propagated to other MAs using the asynchronous PIF described in Algorithm 5.1.

Figure 4 describes a scenario of propagation in a DIET multi-hierarchy, applying the two following phases:

Broadcast phase: The MA that received the request made by the client, initiates the wave and is the root r. It propagates the request to all other MAs it can reach and running at this time: the neighbors of r. It waits for a number of responses equal to the number of its neighbors. The MA receiving a propagated request checks if it has already processed it. If it is the case, it does not process it. Otherwise, the MA that sent the request to it becomes its parent. It collects the servers to solve the problem described in the request in its subtree and propagates the request in its turn to the neighbors of r.

Feedback phase: r waits for all the responses before to answer the client. The neighbors of r send the enabled servers found in their subtrees back to their parent and, when receiving a response from a child, send the response to their own parent.

Let us call this algorithm "PIF$_{async}$" in the following of this paper.

In Figure 4, (1.) The MA that received the request from the client encountered a failure in its own hierarchy when collecting servers able to process this request. It initiates the wave, and is the root. (2.) Some MAs have received the propagated request. They forward it on their turn, and initiate the asynchronous Feedback phase. (3.) All MAs have received the request. A spanning tree is built. The feedback phase goes on and ends. The connections during this phase occurs depending on the traffic load encountered during the brodcast phase, allowing an optimal propagation of feedback messages.

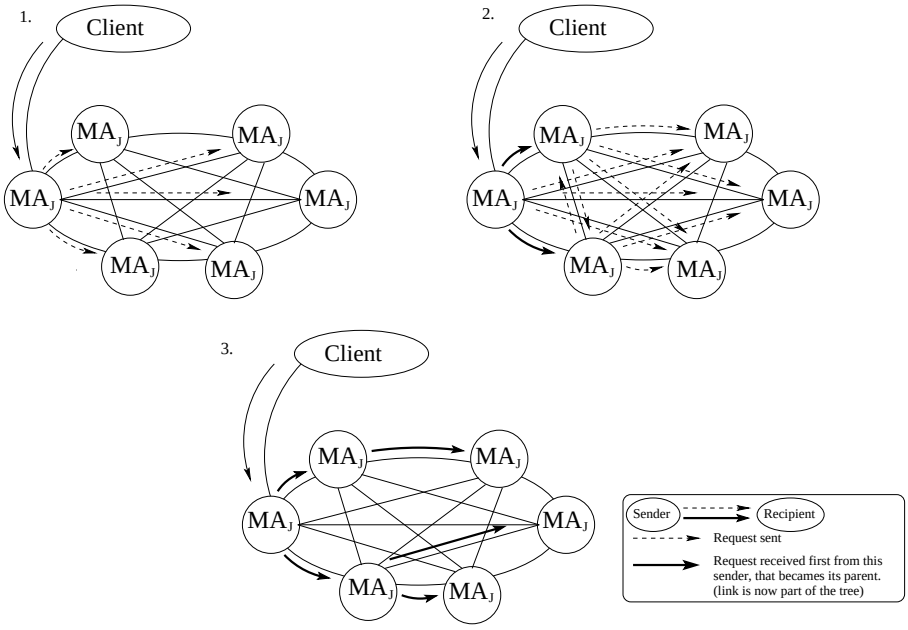


Figure 4: Propagation scenario in a DIET multi-hierarchy.

6 Experimenting the Traversal of the DIET$_J$ Multi-Hierarchy.

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

Algorithm 5.1 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 **FROM** 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) **FROM** 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) **FROM** q
$ListServer := ListServer \cup LS$;
Sync;

6.1 Experimental Platform

Our experimental platform is based on several clusters connected through the VTHD network. This Wide Area Network has a 2.5 Gb/s bandwidth. Clusters used for this experiments are the following. Paraci is a cluster composed of 64 nodes with Intel quadri-processors Xeon 2.4 GHz and Cristal is a cluster composed of 15 nodes with Intel bi-processors Xeon 2.8 GHz.

Only one MA (composed of one MA_J and one MA_{DIET}) runs per node. Only one $client_J$ runs at a given time and sends one or multiple requests to one MA that will be the propagation initializer for this request. Based on our previous experiments inside one hierarchy [6], we here just experiment connections of the MAs graph (without whole DIET hierarchies underneath).

6.2 Experiments with Homogeneous Network Performance

We experimented the propagation and feedback using both implementations described in the previous section. We started our experiments with a low and homogeneous traffic load, by varying the number of MAs in order to estimate the cost of using the PIF_{async} compared to the $STAR_{async}$

i.e, the cost of re-forwarding the request on non-root MAs.

Figures 5 and 6 show the time to initiate the propagation and to receive all the responses, using the STAR_{async} traversal and the PIF_{async} scheme, on two VTHD clusters. On such network, most of the time we obtain the same graph using either the PIF_{async} or the STAR_{async}. Using the PIF_{async}, the initial propagation from the root node will reach other nodes first, because of the homogeneous performance of the links. Thus, the trees obtained with the PIF_{async} are mostly stars.

6.3 Requests Flooding

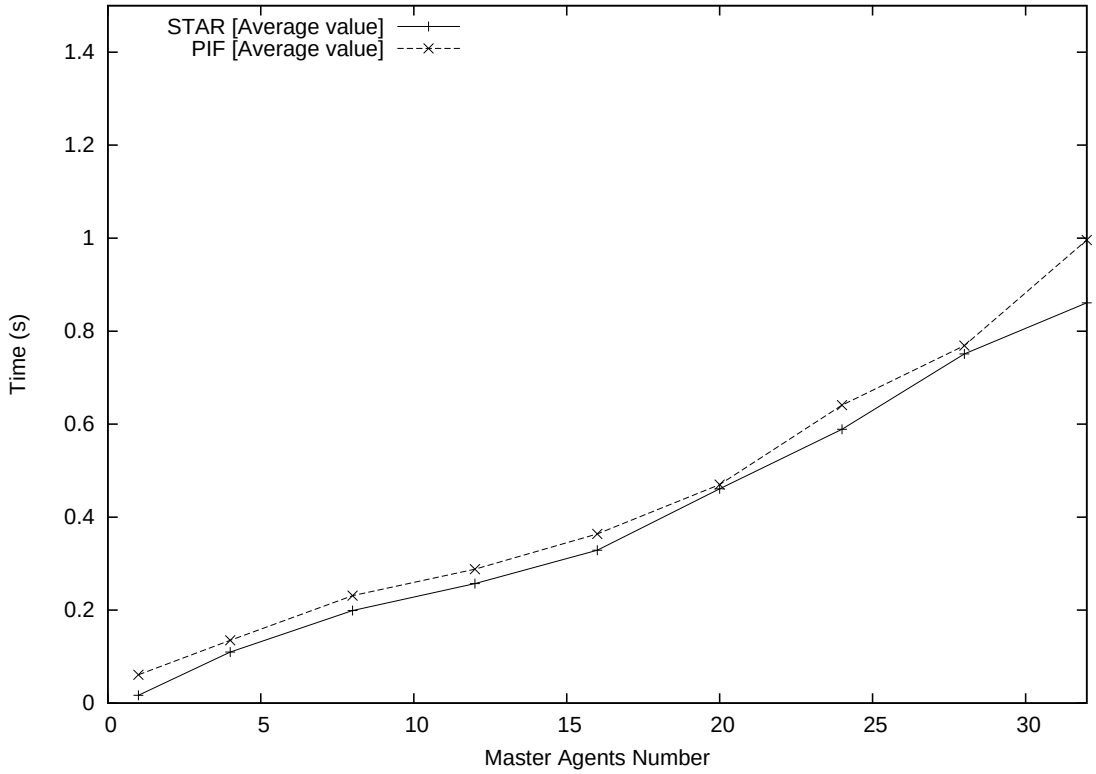


Figure 5: Evaluating the cost of using the PIF_{async} compared to the STAR_{async} on one cluster.

6.4 Requests Flooding

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

Figures 7 and 8 show the impact of processing multiple requests at the same time inside the graph of MAs, with the same root for every request. Better results are obtained by propagating requests with the PIF_{async}. Using the STAR_{async}, physical routes used by the logical JXTA pipes created to send responses are mostly the same for every requests, because the logical star traversal is systematic, and this strongly increases the load on these links. Using the PIF_{async}, logical path (and physical routes underneath) used during the feedback phase depands on the load of the links during the brodcast phase. The initial propagation of r does not reach all the neighbors of r first, because some links are averloded by other requests. Thus, each propagation builds a spanning tree

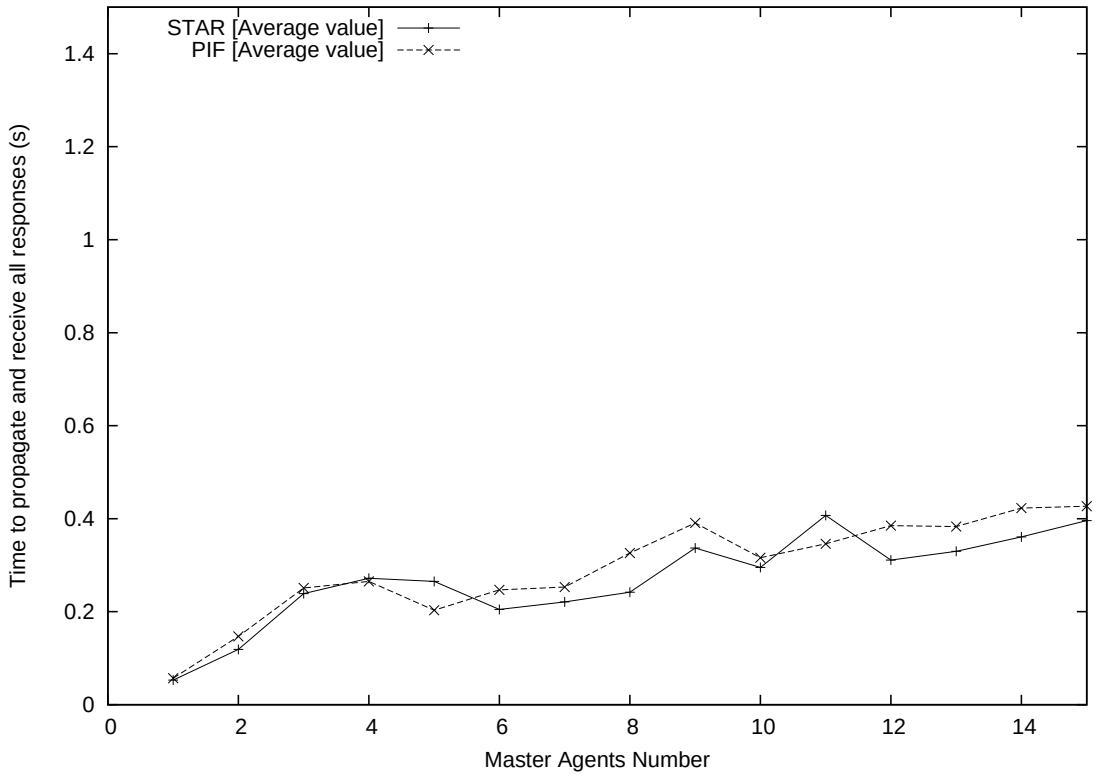


Figure 6: Evaluating the cost of using the PIF$_{async}$ compared to the STAR$_{async}$ on two clusters connected through the VTHD network.

during the broadcast phase minimizing the load of the links used during the feedback phase. The traffic is globally more distributed and bottlenecks are avoided. However, in these experiments, only few trees are built.

6.5 Experiments with Overloaded Links

Finally, we looked at the gain that can be obtained by using the PIF$_{async}$ algorithm that finds the optimal spanning tree on networks with heterogeneous loads on the links. We simulated a loaded traffic with loops of `scp` commands, especially around the MA initiating the propagation. Twelve other Master Agents are run. Figure 9 shows the performance of each algorithm, varying the number of links saturated by traffic load around the MA initiating the propagation. The STAR$_{async}$ will use the saturated links both during the feedback and feedback phases, increasing 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 the broadcast time, for each request. Now, a lot of trees are built, avoiding the bottleneck around the root.

7 Conclusion and Future Work

In this paper, we have presented an implementation of an asynchronous PIF algorithm used for resource discovery in peer-to-peer grids. A dynamic version of the DIET middleware that connects small hierarchies together using JXTA has been developed, which is able to dynamically adapt its

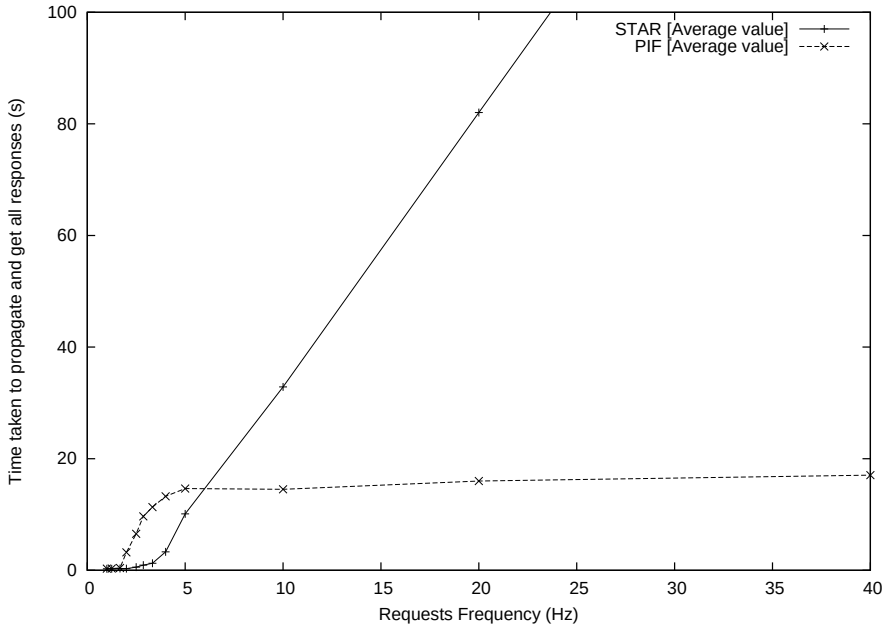


Figure 7: Sending 10 requests at various frequencies.

connections as the network performance evolves and as the number of requests increases. The use of JXTA and the asynchronous PIF algorithm allows a quick and efficient discovery of available servers.

Our experimental results show that the PIF_{async} algorithm has the same cost as the STAR_{async} algorithm when the network performance are homogeneous. Moreover, when the network traffic increases on some links of the target platform, our PIF_{async} algorithm outperforms the STAR_{async} one by choosing the less loaded links to build an optimal tree in the connection graph.

Our future work will consist in validating the algorithm at a larger scale using larger clusters connected through Wide Area Networks (within the Grid5000 project [1]) and to implement more fault tolerance features into DIET (as well as automatic deployment of static DIET hierarchies). On the development side, we plan to use the last version of JXTA-C to allow to directly integrate our algorithm in the C++ version of DIET.

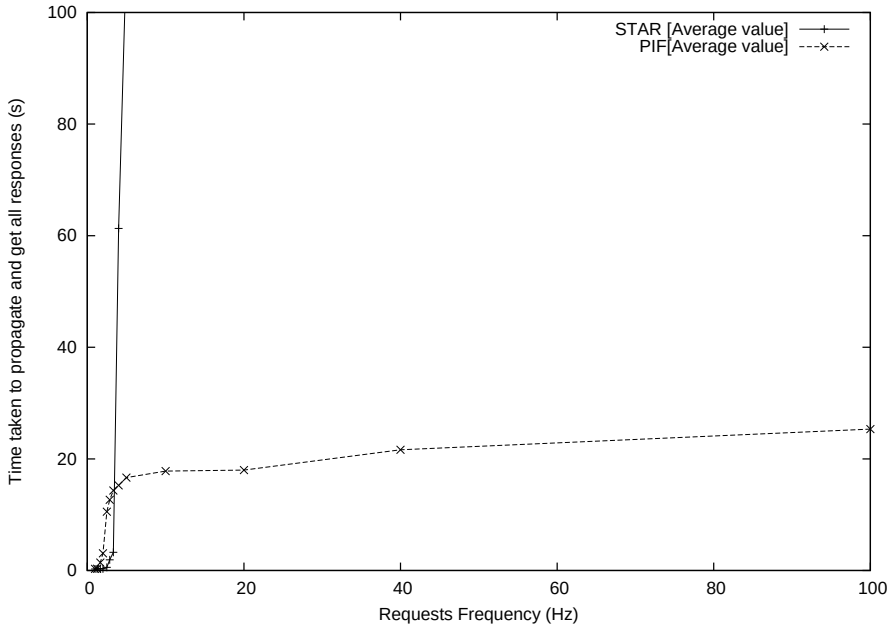


Figure 8: Sending 20 requests at various frequencies.

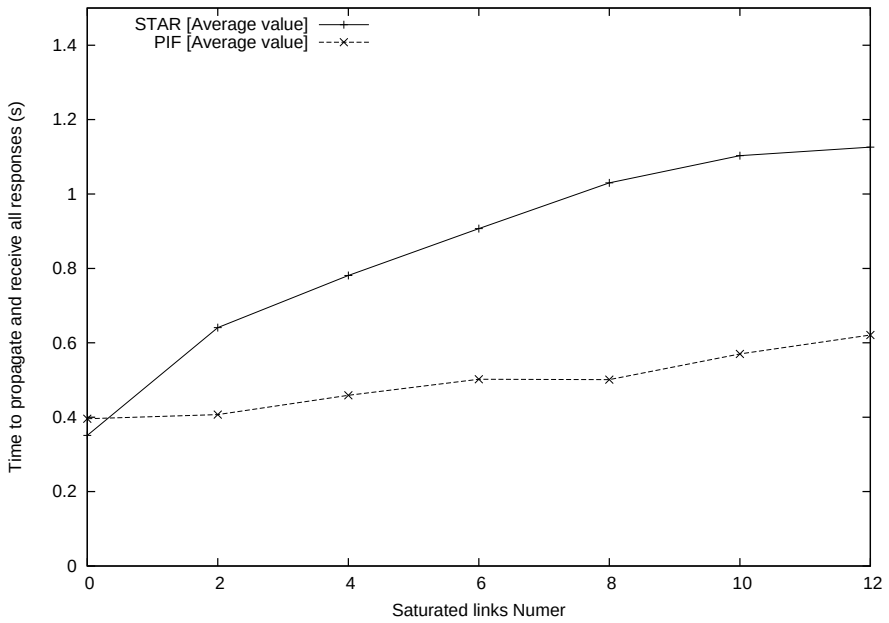


Figure 9: Experimenting the PIF_{async} and the STAR_{async} on a network with overloaded links.

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] A. Bui, A.K. Datta, F. Petit, and V. Villain. Optimal PIF in Tree Networks. In *Proc. of DIMACS Workshop on Dist. Data and Structures*, pages 1–16. Carleton Univ. Press, 1999.

[4] E. Caron, P.K. Chouhan, and A. Legrand. Automatic Deployment for Hierarchical Network Enabled Server. In *The 13th Heterogeneous Computing Workshop (HCW 2004)*, Santa Fe. New Mexico, USA, April 2004.

[5] 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.

[6] 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.

[7] 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.

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

[9] CONDOR. http://www.cs.wisc.edu/condor/.

[10] A. Cournier, A.K. Datta, F. Petit, and V. Villain. Snap-Stabilizing PIF Algorithm in Arbitrary Networks. In *Proceedings of the 22th International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 199–206. IEEE Computer Society, 2002.

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

[12] Globus. http://www.globus.org/.

[13] GRIDLAB. http://www.gridlab.org/.

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

[15] S. Liang. *Java(TM) Native Interface: Programmer's Guide and Specification*. Addison-Wesley Pub Co, 1999.

[16] Netsolve. http://www.cs.utk.edu/netsolve/.

[17] NINF. http://ninf.etl.go.jp/.

[18] 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.

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

[20] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. An Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *3rd International Workshop on Grid Computing*, November 2002.

[21] The JXTA project. http://www.jxta.org.

[22] Bernard Traversat, Mohamed Abdelaziz, and Eric Pouyoul. A Loosely-Consistent DHT Rendezvous Walker. Technical report, Sun Microsystems, Inc, March 2003.

[23] XtremWeb. http://www.xtremweb.org/.