Intelligence Integration in Distributed Knowledge Management

Dariusz Król
Wroclaw University of Technology, Poland

Ngoc Thanh Nguyen
Wroclaw University of Technology, Poland

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE
Hershey · New York

Director of Editorial Content:	Kristin Klinger
Managing Development Editor:	Kristin M. Roth
Senior Managing Editor:	Jennifer Neidig
Managing Editor:	Jamie Snavely
Assistant Managing Editor:	Carole Coulson
Copy Editor:	Lanette Ehrhardt
Typesetter:	Jeff Ash
Cover Design:	Lisa Tosheff
Printed at:	Yurchak Printing Inc.

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue, Suite 200
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

and in the United Kingdom by
Information Science Reference (an imprint of IGI Global)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 0609
Web site: http://www.eurospanbookstore.com

Library of Congress Cataloging-in-Publication Data

Intelligence integration in distributed knowledge management / Dariusz Krol and Ngoc Thanh Nguyen, editors.

p. cm.

Includes bibliographical references and index.

Summary: "This book covers a broad range of intelligence integration approaches in distributed knowledge systems, from Web-based systems through multi-agent and grid systems, ontology management to fuzzy approaches"--Provided by publisher.

ISBN 978-1-59904-576-4 (hardcover) -- ISBN 978-1-59904-578-8 (ebook)

1. Expert systems (Computer science) 2. Intelligent agents (Computer software) 3. Electronic data processing--Distributed processing. I. Krol, Dariusz. II. Nguyên, Ngoc Thanh.

QA76.76.E95I53475 2009

006.3--dc22

2008016377

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter XVIII
Example-Based Framework for Propagation of Tasks in Distributed Environments

Dariusz Król
Wroclaw University of Technology, Poland

ABSTRACT

In this chapter, we propose a generic framework in C# to distribute and compute tasks defined by users. Unlike the more popular models such as middleware technologies, our multinode framework is task-oriented desktop grid. In contrast with earlier proposals, our work provides simple architecture to define, distribute and compute applications. The results confirm and quantify the usefulness of such ad-hoc grids. Although significant additional experiments are needed to fully characterize the framework, the simplicity of how they work in tandem with the user is the most important advantage of our current proposal. The last section points out conclusions and future trends in distributed environments.

INTRODUCTION

The main goal of this project is to create an exemplary system that would allow a network of computers to serve as distributed computer, allowing a client to send computational tasks to this network. The task would be later split into smaller tasks and processed by computers in the network. The reason behind creating a network able to process tasks in distributed way is obvious (Lanunay & Pazat, 2001). Creating a network from many low-end computers in most cases gives us processing capability much greater than one high-end computer of the same cost as many slower computers (Haeuser et al., 2000; Laure, 2001; Matsuoka & Itou, 2001). Also, distributed computing allows for the more effective use of resources of many idle computers in the network (Mitschang, 2003). Creating a task-oriented network is also a good way to understand the basics of remoting and reflection, two mechanisms in C# that are crucial for such solution to work (Gybels, Wuyts, Ducasse, & Hondt, 2006).

On the one hand, to ensure communication between elements of the framework, .NET remoting technology is used. Remoting facilitates method invocation on remote objects works exactly the same way as on local objects. On the other hand, reflection provides necessary mechanisms that allow computer programs to modify themselves during runtime. C# implementation of reflection allows us to load assembly code, create objects, obtain information about assembly code, object, methods, properties and fields, and invoke a method of object. In our work, reflection is necessary to divide tasks into task portions. If a task cannot be divided into portions beforehand, this kind of task cannot be effectively distributed and our framework will be unable to facilitate parallel execution of that task.

The division of tasks into task portions is done during the coding of the task itself. The user needs to specify two classes for each task. One class serves as an "initiator" of task portions. The other class is a task portion, and needs a parameter and returns partial result. Every time a new task is run, an initiator class uses reflection to create instances of task portion class and puts instantiated objects into readyJobs array. This is done by a method called readyJobs.Add (i, Activator.CreateInstance (taskPortionType, parameters)). The first parameter to this method is task portion type and it needs to be declared inside of the "initiator" class. To create such a type of variable, reflection is needed. If the class, which type is set to the type of the task portion, is not loaded yet, reflection can be used to load appropriate assembly and then get the type of the class.

The rest of the chapter is organized as follows. The following section (Section "Background") introduces the cross-platforms for desktop grids. The next section (Section "Elements of the Multinode Framework") details the elements of the multinode framework. The fourth section (Section "Communication between Elements via .NET Remoting") proposes a communication schema between elements via .NET remoting. The next section (Section "Defining a Task to Distribute and Compute") describes how to define a task to distribute and compute. The next section (Section "Study of the Framework Performance") studies the performance. The last section (Section "Conclusion") points out conclusions and future trends in distributed environments.

BACKGROUND

There are many solutions for task propagation in distributed environments. The most known is the BOINC (Berkeley Open Infrastructure for Network Computing) project (Anderson, 2004). The system consists of a set of applications running in a Linux environment. They mostly have the form of independent daemons communicating with each other using a database or shared memory. Every project based on BOINC needs to have its own server. In order to take part in a project, community members need to install dedicated client software. This software connects to server and downloads data and executables necessary for carrying out tasks. The client may be connected to several projects at the same time. It is some form of reusability; however, it comes within client's administrator duties to connect manually to a new server. In this approach, there is no knowledge sharing or projects coordination. Each project runs independent. It may have some advantages when system failures are taken into consideration. In BOINC architecture, when one project is shut down, the others can continue to operate normally. The autonomy of projects increases the probability of error detection. Every project can define its own assertions to the data being the result of processing.

Recently, some .NET grids have been created. OGSI.net, developed at the University of Virginia (OGSI.net, 2007), is an implementation of the

OGSI specification on Microsoft's .NET platform. However, the OGSI.net project is committed to interoperability with other OGSI compliant frameworks (such as the Globus Toolkit 3) which run primarily on Unix systems and so represents a bridge between grid computing solutions on the two platforms. OGSI.NET provides tools and support for an attribute-based development model in which service logic is transformed into a grid service by annotating it with metadata. OGSI.NET also includes class libraries that perform common functions needed by both services and clients.

The Alchemi project from the University of Melbourne (Alchemi, 2007) is an open source software framework that allows you to painlessly aggregate the computing power of networked machines into a virtual supercomputer (desktop grid) and to develop applications to run on the grid. It has been designed with the primary goal of being easy to use without sacrificing power and flexibility. Alchemi includes the runtime machinery (Windows executables) to construct computational grids, a .NET API, and tools to develop .NET grid applications and grid-enable legacy applications.

The motivations of our work have similarities with BOINC, OGSI.net, and Alchemi, and differences from work of Poshtkohi, Abutalebi, and Hessabi (2007). This work proposes the DotGrid project to share, select and aggregate distributed resources in an integrated way based on Microsoft .NET in Windows and MONO .NET in Linux. This has come true via implementing a layer over the chosen operating systems. This approach eliminates the dependency of grid to the native system. The .NET programming environment includes features that are suitable for simplicity and efficiency computing: multithreading, remoting, and reflection. We use the last one to divide tasks into task portions.

Grid computing is one of the most innovative aspects in recent years. Multi-agent computing now becomes a promising solution in many domains. Hence, it is a natural choice to combine these two technologies together. Although grid technology heavily relies on efficient computation with interaction, most of the current systems or applications lack the vision of utilizing computer interaction. Recently, there has been a shift toward agent-based grid computing, with many researchers contributing to the field.

In the following section, we briefly present some of the main elements of the multinode framework, which addresses the problems described previously.

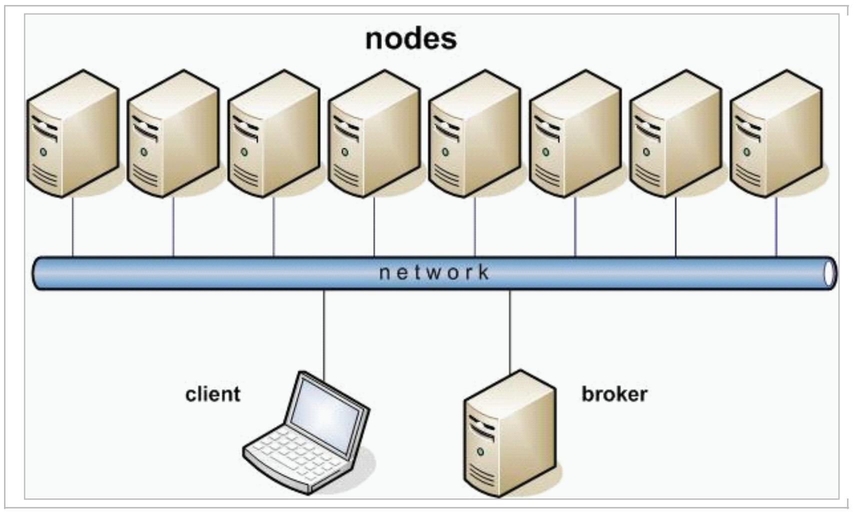
ELEMENTS OF THE MULTINODE FRAMEWORK

The framework that enables distributed code propagation (Deng, Han, & Mishra, 2005; Król, 2005; Król & Kukla, 2006; Lauzac & Chrysanthis, 2002; Lin & Kuo, 2000) is composed of the main elements (shown in Figure 1):

- The broker represents the central part of the system; it is the application which receives tasks, processes them and distributes parts of the tasks to remote nodes, then fetches the partial results and combines them into final result;
- The client is the application which connects to the broker and sends tasks with appropriate format; and
- The nodes are background applications running on many computers, constantly receiving task portions from the broker, processing them and finally returning the partial result to the broker.

Even though all elements of the system are connected with each other, all communication is channeled through the broker and there is no direct connection between client and any of the other nodes.

Figure 1. Elements of the project



COMMUNICATION BETWEEN ELEMENTS VIA .NET REMOTING

Within our framework, communication is needed to:

- Send tasks from client to broker;
- Send task portions from broker to nodes;
- Send task results from nodes to broker; and
- Send other signals (login and logouts) between client, broker and nodes.

The communication between system elements can be described with an example of sending tasks from client to broker. Elements of the system that need to receive something become a server. The application that is sending something becomes a client. The broker creates a well-known service of MyTaskReceiver with method receive-Task (object Task). To pass a task instance, a client needs just to call remote method and give task as parameter. The method will return true of false value, depending on the result of the method. If a communication problem happens, a particular exception will be thrown. To allow two applications to communicate through remoting, one of them must be configured to be a server and another as a client.

Server needs to:

1. Register HTTP server port.
2. Register well-known service with remote object type.

Client needs to:

1. Register HTTP client port.
2. Create object of remote object type.
3. Activate remote object.
4. Run remote method.

Our project uses the following elements implemented as interfaces (shown in Figure 2).

As we can see in the Figures 3, 4, and 5, the interfaces define only what methods are available within the remote objects. The interfaces themselves do not define what is exactly executed by method. This is computed entirely on the server side. An application which is defined as server needs to have specific method definitions. In this system, this is done by:

Figure 2. Interfaces used in the framework

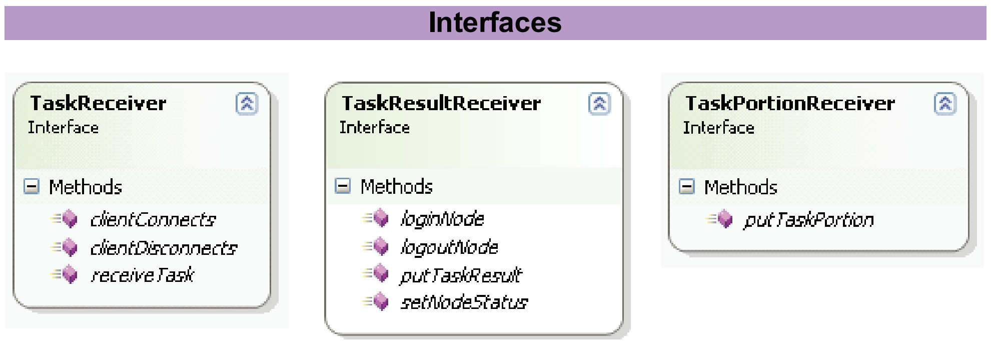


Figure 3. TaskReceiver interface

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Nestor
{
  public interface TaskReceiver
  {
    bool clientConnects();
    bool clientDisconnects();
    string receiveTask(object task);
  }
}
```

- Class MyTaskReceiver on broker, extending TaskReceiver interface;
- Class MyTaskResultReceiver on broker, extending TaskResultReceiver interface; and
- Class MyTaskPortionReceiver on nodes, extending TaskPortionReceiver interface.

DEFINING A TASK TO DISTRIBUTE AND COMPUTE

To facilitate the process of sending tasks from a client to the broker, a generic task class needs to be defined. A task should be defined this way, so the broker will be able to divide a task into the smaller tasks (called here task portions), which will be later sent to the nodes. After gathering the

Figure 4. TaskResultReceiver interface

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Nestor
{
  public interface TaskResultReceiver
  {
    /// <summary>
    /// Adds node to logged nodes list
    /// </summary>
    /// <returns>Returns assigned number of node (>=0) or error.</returns>
    int loginNode(string host);

    /// <summary>
    /// Removes node from logged node list
    /// </summary>
    /// <returns>True on success, false otherwise.</returns>
    bool logoutNode(int loggedNodeId);

    /// <summary>
    /// Passes to broker information about current status of node.
    /// This will be saved and displayed on connected nodes list.
    /// </summary>
    /// <returns>True on success, false otherwise.</returns>
    bool setNodeStatus(int loggedNodeId, string status);

    /// <summary>
    /// Passes to broker result of finished task.
    /// </summary>
    /// <returns>True on success, false otherwise.</returns>
    bool putTaskResult(int loggedNodeId, long portionNumber, string result);
  }
}
```

Figure 5. TaskPortionReceiver interface

```
  using System;
  using System.Collections.Generic;
  using System.Text;

  namespace Nestor
  {
    public interface TaskPortionReceiver
    {
      /// <summary>
      /// Sends to a node a task portion
      /// </summary>
      /// <returns>true on success, false otherwise</returns>
      bool putTaskPortion(TaskPortion portions, long taskPortionNumber);

      /// <summary>
      /// Tests if a connection has been made
      /// </summary>
      /// <returns>Always true when connected, otherwise would return false
but without connection will throw an exception</returns>
      bool testConnection();

      /// <summary>
      /// Signals to a node that broker disconnects
      /// </summary>
      void brokerDisconnects();

    }
  }
```

results from all the task portions, the final result is created by the broker and returned. In this project, a task is defined by two interface classes: Task and TaskPortion, with the following methods:

- Task.doBrokerJob: This method runs just after receiving the task. All initial instructions, which are preparing variables and job should be set here;
- Task.getResult: This method runs after all jobs return the results. This method aggregates partial results and displays result of the task; and
- TaskPortion.doNodeJob: This method runs on a node, which contains all the operations needed to get the partial results. This method is passed a parameter which is a reference to a table of results.

C# interfaces do not contain any information about the constructor, but when we are defining a task we need to add constructor for TaskPortion class. This constructor needs a parameter that will differentiate various task portions.

The body of the doBrokerJob method needs to create a set of tasks which will be put into readyJobs table (passed by reference). This is done using the reflection mechanism. First, we use reflection to create an object containing the type of TaskPortion class. Second, we use reflection to dynamically create a number of instances of Task-Portion classes with distinct parameters, and then to store those instances as ready jobs.

STUDY OF THE FRAMEWORK PERFORMANCE

The main aim of creating this framework is to utilize the power of more than one computer and create a simple way for parallel processing of various tasks. To apply for the success, the following study was made. The two defined tasks described use extensive mathematical calculations.

- **Task 1.** Checks which numbers from range 1000000001 … 1000000100 are prime. This task is irregular, and there is no way to predict how long it might take to calculate one portion of the task.
- **Task 2.** Calculates 100 packs of 100 hexadecimal digits of the number π. This task is regular; all the portions take the same amount of processor time.

Normally, those tasks, depending on processor speed, require several minutes to complete. In our study, we have checked how the number of nodes influences the time of the task.

The results of the study are the following. Task 1, on a single node, was performed in 1 minute 11 seconds. After adding the second node, we have gained a decrease of 21 seconds. By adding the third node, we got an additional 26 seconds. Then, adding the next node did not result in speed gain. We can explain this with the nature of this task. When we send a task portion to a node, we don't know the size of this portion. If this number will be prime, it will be checked against all numbers smaller than itself (which is time consuming). If it will not be prime, the task will be stopped as soon as we find a number that is a divider of this number. From the result, we see that the complete loop of testing one number takes about 24-30 seconds. Also, from Figure 6 we see that in studied range we have only three prime numbers, because after adding the third node we do not gain any speed, which basically

Figure 6. Performance index for prime numbers checking on nodes

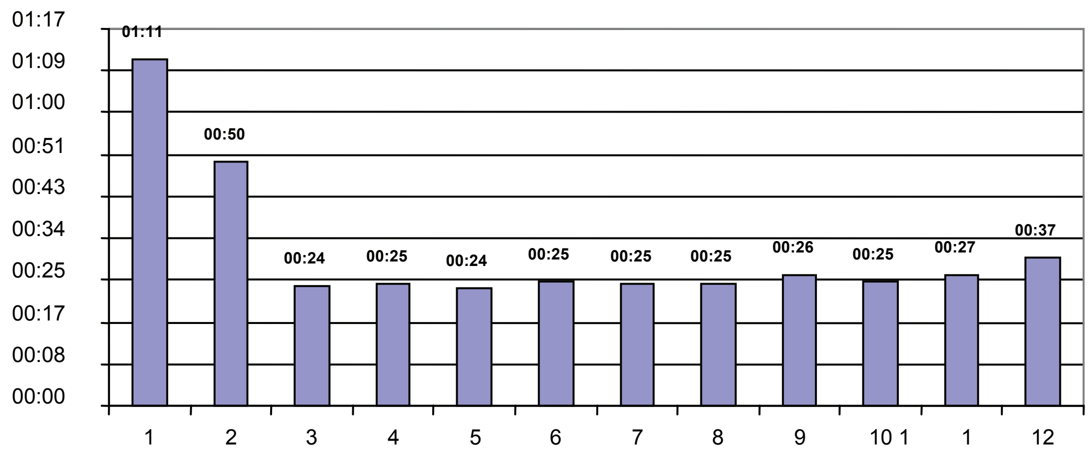
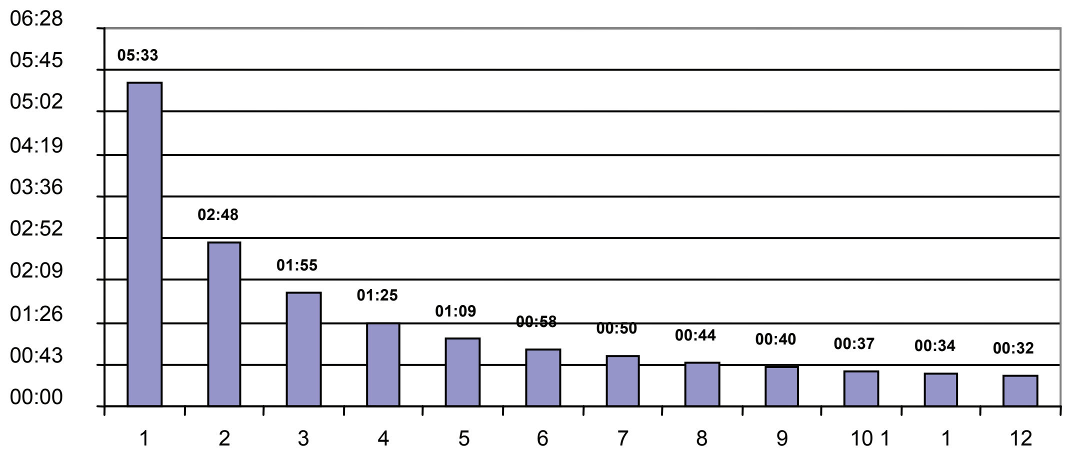


Figure 7. Performance index for digits of the number π calculation on nodes



means that we have three nodes processing, and the rest of the nodes are idle.

The second task proved the worth of building such networks. In this task, all the portions are even. From Figure 7, we can clearly see that with adding each new node, the total processing time was reduced. The relation is close to invert proportional (total time = total time for 1 node/ number of nodes).

CONCLUSION

This framework is prepared to only run a pair of predefined algorithms. To add more algorithms, the following actions should be made.

- Create two serializable task classes inheriting from Task and TaskPortion classes and implement all needed methods. Add those classes to application namespace, so that client application is able to create instances.
- The task class has to use the doBrokerJob method to fill the readyJobs table passed as a parameter. This table needs to be filled with instances of the TaskPortion class.
- The task class has to use the getResult method to aggregate results of the task portions (table jobResults) and return the single string containing the aggregated result.
- The taskPortion class can use the constructor parameters to pass parameters from the broker to a node.
- The taskPortion class has to use the doNodeJob method to perform task portion calculations and return the result of a task portion by reference to the table jobResult.

The only limitation of this framework is that algorithms used here must be easily divided into smaller tasks (which can require a lot of calculations). Task portions are distinguished by a parameter passed to a task portion. The type and number of those parameters are not limited, allowing the designer to make TaskPortion flexible to allow any kind of parameters.

In our approach, we can observe emergent simplicity while defining algorithms. In both algorithms, the first run of the doBrokerJob method creates all task portion instances that will be needed throughout the whole run of the task. While defining more parameters, we will need to add task portions conditioned on the results of previous task portions or other conditions. Modification of the doBrokerJob method will solve the issue noted here.

As computational framework evolves from network-oriented to task-oriented, our efforts are shifted to the semantic agent-based grid. To succeed into this trend, various research aspects should be investigated. Our architecture should not be restricted for grids. It can also benefit the advantages from multi-agent systems, P2P technique, and Web services. Autonomous intelligent agents can monitor, evaluate and repair the system. This demands many migrations from agent environments to grids. We can also use communicative intelligence, fuzzy logic, nature-inspired algorithms and game theory.

ACKNOWLEDGMENT

I wish to thank WUT (Wroclaw University of Technology), which gave me a grant. I am pleased to acknowledge the help of Marek Kowalczyk for his technical support and contribution. The author gratefully acknowledges helpful remarks and suggestions from Mariusz Nowostawski. Many thanks to anonymous reviewers who made some comments on the chapter. All errors and flaws are the sole responsibility of the author.

REFERENCES

Alchemi (2007). *.NET-based enterprise grid*. Retrieved April 3, 2008, from http://www.alchemi.net/

Anderson, D.P. (2004). BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, (pp. 4-10).

Deng, J., Han, R., & Mishra, S. (2005). *Secure code distribution in dynamically programmable wireless sensor networks* (Tech. Rep. No. CU-CS-1000-05). Boulder, CO: University of Colorado.

Gybels, K., Wuyts, R., Ducasse, S., & Hondt, M. (2006). Inter-language reflection: A conceptual model and its implementation. *Computer Languages, Systems and Structures, 32*, 109-124.

Haeuser, J., et al. (2000). A test suite for high-performance parallel Java. *Advances in Engineering Software, 31*, 687-696.

Król, D. (2005). A propagation strategy implemented in communicative environment. *Lecture notes in artificial intelligence* (Vol. 3682, pp. 527-533). Springer-Verlag.

Król, D., & Kukla, G. (2006). Distributed class code propagation with Java. *Lecture notes in artificial intelligence* (Vol. 4252, pp. 259-266). Springer-Verlag.

Lanunay, P., & Pazat, J.L. (2001). Easing parallel programming for clusters with Java. *Future Generation Computer Systems, 18*, 253-263.

Laure, E. (2001). OpusJava: A Java framework for distributed high performance computing. *Future Generation Computer Systems, 18,* 235-251.

Lauzac, S.W., & Chrysanthis, P.K. (2002). View propagation and inconsistency detection for cooperative mobile agents. *Lecture Notes in Computer Science, 2519,* 107-124.

Lin, J.W., & Kuo, S.Y. (2000). Resolving error propagation in distributed systems. *Information Processing Letters, 74*(5-6), 257-262.

Matsuoka, S., & Itou, S. (2001). Towards performance evaluation on high-performance computing on multiple Java platforms. *Future Generation Computer Systems, 18,* 281-291.

Mitschang, B. (2003). Data propagation as an enabling technology for collaboration and cooperative information systems. *Computers in Industry, 52*(1), 59-69.

OGSI.net. (2007). *Main page.* Retrieved April 3, 2008, from http://www.cs.virginia.edu/~gsw2c/ogsi.net.html

Poshtkohi A., Abutalebi, A.H., & Hessabi, S. (2007). DotGrid: A .NET-based cross-platform software for desktop grids. *International Journal of Web and Grid Services, 3*(3), 313-332.