Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844
Vol. IV (2009), No. 1, pp. 82-91

Development Journey of QADPZ - A Desktop Grid Computing Platform

Monica Vlădoiu, Zoran Constantinescu

Monica Vlădoiu
Petroleum-Gas University of Ploieşti, Departament of Informatics
Bd. Bucureşti, Nr. 39, Ploieşti, Romania
E-mail: mmvladoiu@acm.org

Zoran Constantinescu
Zealsoft Ltd.
Str. Tg. Neamţ, Nr. 60, Bucureşti, Romania
E-mail: zoran@unde.ro

Abstract: In this paper we present QADPZ, an open source system for desktop grid computing, which enables users of a local network or Internet to share resources. QADPZ allows a centralized management and use of the computational resources of idle computers from a network of desktop computers. QADPZ users can submit compute-intensive applications to the system, which are then automatically scheduled for execution. The scheduling is performed according to the hardware and software requirements of the application. Users can later monitor and control the execution of the applications. Each application consists of one or more tasks. Applications can be independent, when the composing tasks do not require any interaction, or parallel, when the tasks communicate with each other during the computation. The paper describes both QADPZ functionality and the process of design and implementation, with focus on requirements, architecture, user interface and security. Some future work ideas are also presented.
Keywords: desktop grid computing, distributed and parallel computing.

1 Introduction

Grid computing and Peer-to-Peer (P2P) are both concerned with the pooling and coordinated use of resources within distributed communities, and are constructed as overlay structures that operate largely independently of institutional relationships [1]. The Grid is foreseen as a system that coordinates distributed resources using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service [1, 2]. Grid computing systems can be classified into two broad types: heavy-weight, feature-rich systems that provide access to large-scale, intra- and inter-institutional resources, such as clusters or multiprocessors, and Desktop Grids, in which cycles are scavenged from idle desktop computers. P2P networks are typically used for connecting nodes via largely ad-hoc connections. A pure P2P network does not have the notion of clients or servers but only equal peer nodes that simultaneously function as both "clients" and "servers" to the other nodes on the network [3].

This paper deals with QADPZ ['kwod 'pi: 'si:], an open source system for desktop grid computing, which enables users from a local network or Internet to share their resources [4, 5]. QADPZ (Quite Advanced Distributed Parallel Zystem) is a system for heterogeneous desktop grid computing that allows a centralized management and use of the computational resources of idle computers from a network of desktop computers. QADPZ users can submit compute-intensive applications to the system, which are then automatically scheduled for execution. Applications can be independent, when the composing tasks do not require any interaction, or they can be parallel, when the tasks communicate with each other during the computation. Thus, the system provides support for both task- and data-parallelism. Here are some important features of QADPZ [4]:

- native support for multiple operating systems: Linux, Windows, MacOS and Unix;
- support for legacy applications, which for different reasons could not be rewritten;
- object-oriented development framework that supports either low-level programming languages as C and C++, or high-level language applications (such as Lisp, Python, or Java), and that provides for using such applications in a computation;

- master worker-model that is improved with some refined capabilities: pushing of work units, pipelining, sending more work-units at a time, adaptive number of workers, adaptive timeout interval for work units, and the use of multithreading [6];
- a master can act as a client to another master. That makes it possible to create a distributed master, which consists of independent master nodes which communicate with each other, thus creating a virtual master;
- extended C/C++ API, which supports creation of lightweight tasks and parallel computing, using the message passing paradigm (MPI) [7];
- low-level optimizations: on-the-fly compression and encryption for communication. To increase performance, an experimental, adaptive compression algorithm, which can transparently choose from different algorithms, is also provided;
- efficient communication by using two different protocols (UDP and TCP/IP);
- autonomic computing characteristics: self-knowledge, self-configuration, self-optimization and self-healing [8].

2 Justification for a new desktop grid system

The idea of using the idle computational resources from existing desktop computers is not new, though the use of such distributed systems, especially in a research environment, has been limited. This is due to the lack of supporting applications, and challenges regarding security, management, and standardization. The need to develop QADPZ has arisen from the following main reasons:

o many existing systems were highly specialized in a very limited set of computationally challenging problems, and hence did not allow the execution of a general application. For example, SETI@home was programmed to solve one specific task: the analysis of data from telescopes [9, 10]. Similarly, distributed.net aimed to test the vulnerability of some particular encryption schemes [11];

o at the time of the development, the source code was generally not available, hence making difficult the extension or analysis of any new, non-standard application. Commercial systems such as Entropia, Office Grid and United Devices offered numerous features, but they were not free [4, 12]. On the other hand, some open source systems were available, e.g. XtremWeb [13], BOINC [14, 15], Condor [16], but they were limited in functionality;

o very few existing systems allowed specific considerations to be made wrt. challenges of computationally intensive applications, especially those of scientific computing and visualization [4]. Systems like BOINC and Bayanihan [12] allowed only task parallelism, where there was no communication between the running tasks during a computation. Most computationally intensive applications need such communication;

o most of the existing systems usually had a complicated deployment procedure, requiring high-level, privileged access to the desktop computers, which made very hard to use such systems on a larger scale, and also made further maintenance of the computers complicated - e.g. Condor and Globus Toolkit [12, 17, 18];

o many of today's networks are heterogeneous, thus requiring a distributed computing system with support for various architectures and different type of operating systems. The Java language provides the incentives for such requirements, and many Java based systems emerged: JXTA, Bayanihan, XtremWeb, Javelin [12]. There were very few systems supporting different architectures and operating systems in native mode, some of them being Condor and BOINC. There were also systems, which run only on one type operating system, either Windows or Unix, thus limiting their usability in heterogeneous environments - for instance, Entropia [12].

3 QADPZ Requirements

Given the reasons mentioned in the previous section, we have set up a set of requirements that a successful desktop grid computing system should satisfy to support computationally intensive applications. The overall goal of the system was to be friendly, flexible and suitable to a variety of needs. The main prerequisite has therefore been an open architecture that could evolve in pace with the needs and challenges of the real world.

Two sets of requirements for QADPZ have been specified: one for the system as a whole, mostly from a functional point of view, and another for the system interface. Additionally, a set of non-functional requirements that concern the development of the platform itself has been established. System requirements are concerned mainly with sharing and management of both resources and application jobs, in a heterogeneous environment. They also involve performance and usability of the system, as required by our conceptual model (extended master-worker). The system interface covers both user interfaces and programming interfaces [4, 6].

The system requirements are listed further on:

- *resource sharing*: idle computational cycles, storage space, specific data, etc. of the desktop machines which contribute to the system;
- *resource management*: efficient management of the available shared resources, which remain under the control of their owners via use policies and specific mechanisms;
- *job management*: users should be able to submit, monitor and control the execution of computational jobs on the system;
- *heterogeneity*: ability to work on a network of heterogeneous desktop computers, with different architectures (Intel, RISC, etc.) and different operating systems (UNIX, Windows, Mac OS, Linux);
- *simple installation and minimal maintenance;*
- *parallel programming support*: support for different parallel programming paradigms, for example both task- and data-parallelism, by using well known standards;
- *network support*: ability to work both in a LAN environment and in Internet;
- *communications*: the higher level communication protocol should rely on both TCP/IP and UDP/IP, this dual support increasing the efficiency;
- *autonomous features*: support for different autonomicity aspects: self-management, self-optimization, self-healing, self-configuration, and self-knowledge;
- *provide performance measurements*, which could be exploited for better usage of the available resources;
- *on-line/off-line support* for both batch (the user submits jobs which will be executed at a later time) and interactive applications (the user can inspect the partial result and interact with the execution of the application).

The interface requirements can be split up into two parts: first, the *user interfaces* that is the graphical interface, which the human users use to access the system. Using this interface, the users can either monitor or control the behavior of the system. The other interface is the *programming interface* (API), which allows different user applications to interact with the system. The interface requirements are enlisted beneath:

- *personalization*: different levels of access for various users, according to their personal skills and preferences;
- *job management interface*: a simple, platform independent, graphical user interface, to allow submission, monitoring and control of the different computational jobs;
- *resource sharing interface*: a simple, intuitive graphical user interface, which allows the control of shared resources.

The main non-functional requirements concern *object oriented programming*, for its well-known advantages, and *open source development*, which is a natural choice for modern research, as it encourages integration, cooperation and boosting of new ideas [19].

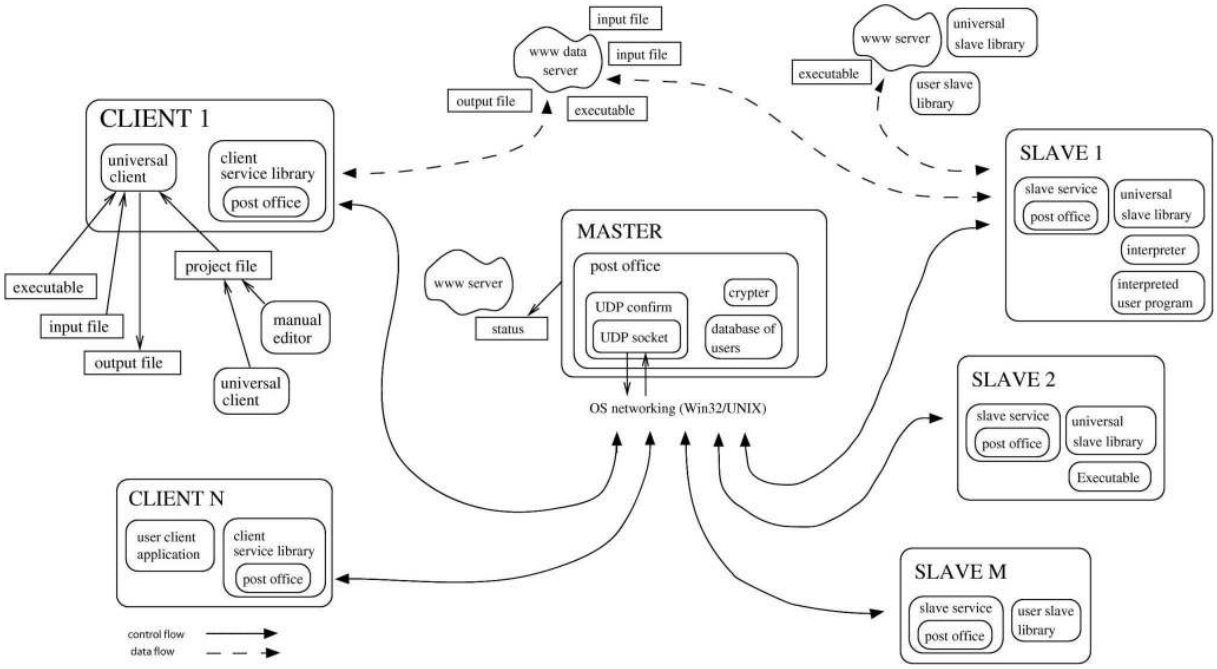

Figure 1: The QADPZ close-up architecture

4 QADPZ Architecture

The QADPZ system has a centralized architecture, based on the client-server model, which is the most common paradigm used in distributed computing. In our case, the server manages the available computational resources of the desktop computers. The client is a process that needs computational services in order to accomplish a certain work. It sends a request to the server, in which it asks for the execution of a concrete task that is covered by the services. Usually, the server carries out the task and sends back the result to the client. In our situation, the server has two parts: a single master, which accepts new requests from the clients, and multiple slaves, which handle those requests. The system consists of three types of entities: master, client, and slave (Figure 1).

The control and data flow in the system are separated. The data files (represented by binary, input, and output files) that are necessary to run the applications are not sent to the master. They are stored on one or more data servers. The smallest independent execution unit of the QADPZ is called a *task*. To facilitate the management, multiple tasks can be grouped into *jobs*. Different types of jobs can be submitted to the system: programs written in scripting languages (e.g. LISP, Java, Python), legacy applications and parallel programs (MPI). A job can consist of independent tasks, which do not require any kind of communication between them. This is usually called *task parallelism*. Jobs can also consist of parallel tasks, where different tasks running on different computers can communicate with each other. Inter-slave communication is accomplished using a MPI subset.

The current implementation of QADPZ considers only one central master node. This can be an inconvenience in certain situations, when computers located in different networks are used together. However, our high-level communication protocol between the entities allows a master to act as a client to another master, thus making possible to create a *virtual master*, consisting of independent master nodes, which communicate with each other.

4.1 Master

The main role of the master is to start and control the tasks, and to keep track of the availability, capabilities and configuration of the slaves. The *master* is responsible for managing the available resources and it has always an up-to-date overview of the system resources. It knows which slaves can accept jobs for execution and how to contact them. It schedules also the computational tasks submitted by any authorized user. Jobs are sent to the appropriate slave based on the hardware and software requirements from the job description. Tasks can be started, stopped, or re-scheduled by the master. Users create tasks that can be submitted to the master by using a *client*, which acts as an interface to the system. To make

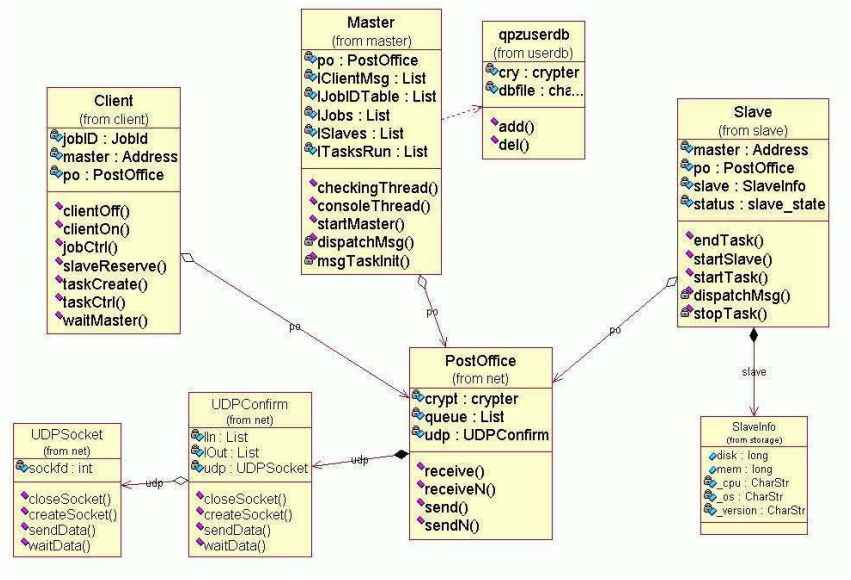

Figure 2: Simplified UML Diagram of QADPZ's architecture

this possible, the master keeps a database of authorized users (Figure 2).

4.2 Slave

Each computer contributing with computing resources to the system is called *a slave* and has two roles: first, it has to report the shared resources to the master. These are mainly computational resources (CPU cycles), but can also be storage space, input or output devices etc. The slave periodically sends to the master information about the system, which describes the hardware architecture of the slave (CPU type, CPU speed, physical memory, etc.), the software environment available on that architecture (operating system, available application or libraries), and the resources available on that slave (Figure 5). Secondly, the slave can accept computational jobs from the master. After accepting a computational request from the master, the slave downloads the corresponding binaries and data files for the task, executes the task, and uploads the result files after finishing. This can be done only when the slave is free, and any interactive, local user is not using the resources. The presence of a user logging into a slave computer is automatically detected and the task is killed or moved to another slave to minimize the disturbance to the regular computer users. The slave decides for itself whether or not to accept a computational job to be run (by setting some configuration parameters). The user can configure different times of day when the slave may accept computational jobs. It can also disable the slave at any time. The slave component runs as a small background process on the user's desktop. It starts automatically when the system starts. The program does not need any special privileges to run, which makes it very easy to install and control by an ordinary user. Below we present a simple example on how to create a computational application to be executed on a slave.

```
// SlaveDumb - simple example of how to create a computational job
#include "SlaveServ.h"
// callback functions for notification from the slave service

void taskStop ()
{ isTaskStop = 1;  DBUG_PRINT("taskStop");  }

void taskCtrl (const char *arg)
{ isTaskCtrl = 1;  DBUG_PRINT("taskCtrl arg=%s", arg);  }
// this is the exec loop on each task-thread
int taskExec (char *data, char *datares, char *userData)
{
  int isFinished = 0; DBUG_PRINT("task  started");
  // set callback functions
  q2adpz_slv_setcb_task_stop (taskStop);
  q2adpz_slv_setcb_task_ctrl (taskCtrl);
```

```
  DBUG_PRINT("input data '%s'", data);
  // start main task loop
  while (! isFinished) {
    //do some crunching of the data
    { ... if (ok) isFinished = 1; }
    //task needs to be stopped
    if (isTaskStop) { ... DBUG_PRINT("task stop executed"); break; }
    if (isTaskCtrl) {
        ... q2adpz_slv_task_status (task_ok, "task ctrl");
        DBUG_PRINT("info", ("task ctrl executed"); }
    //if crunching finished
    if (isFinished) {
        DBUG_PRINT("task finished res='%s'", datares); break; }
    } // while
}
```

4.3 Client

The client represents the interface for submitting jobs in the system. There are two execution modes for the client: a *batch mode* and an *interactive mode*. In the *batch mode*, a project file describes a job by specifying the required resources and how to start the tasks. This information is sent to the master, which is responsible for scheduling the tasks. The client can detach from the master and connect later for the results. Each project is described by using the XML language. In the *interactive mode*, the client remains connected to the master for the entire execution of the job. Also, the client can get direct connection to each of the slaves involved in the computation. The client has a lot of freedom over the creation and controlling of new tasks: it can dynamically create new tasks, send messages to the tasks already in execution, and receive feedback from the running tasks, either through the master node, or by means of direct communication with the slaves running the respective tasks. An example of a job description in XML is listed beneath.

```
<Job Name="executable_example">
  <Task ID="1" Type="Executable">
    <RunCount>3</RunCount>
    <DataPathPrefix>./datafiles/</DataPathPrefix>
    <FilesURL>http://www-data/qadpz/cgi-bin</FilesURL>
    <InputFile Constant="Yes">simple/source.txt</InputFile>
    <OutputFile>simple/dest.txt</OutputFile>
    <TaskInfo>
      <Memory Unit="MB">1</Memory>
      <Disk Unit="MB">1</Disk>
      <TimeOut>3600</TimeOut>
      <OS>Linux</OS>
      <CPU>i386</CPU>
      <URL>
        http://www-data/qadpz/app/lib/Linux/i386/libslv-app.so
      </URL>
      <Executable Type="File">simple</Executable>
    </TaskInfo>
    <TaskInfo>
      <Memory Unit="MB">1</Memory>
      <Disk Unit="MB">1</Disk>
      <TimeOut>3600</TimeOut>
      <OS>Win32</OS>
      <CPU>i386</CPU>
      <URL>
        http://www-data/qadpz/app/lib/Win32/i386/slv_app.dll
      </URL>
      <Executable Type="File">simple.exe</Executable>
    </TaskInfo>
    <TaskInfo>
      <Memory Unit="MB">1</Memory>
      <Disk Unit="MB">1</Disk>
      <TimeOut>3600</TimeOut>
      <OS>SunOS</OS>
      <CPU>sun4u</CPU>
      <URL>
        http://www-data/qadpz/app/lib/SunOS/sun4u/libslv-app.so
```

```
      </URL>
      <Executable Type="File">simple</Executable>
    </TaskInfo>
  </Task>
</Job>
```

4.4 Jobs, tasks and subtasks

The QADPZ users can submit, monitor, and control computing applications to be executed on the computers that share resources. Tasks can be binary programs, which can run on any of the sharing computers. A task comes in the form of an executable program, compiled for a specific architecture and operating system. For better performance, a task can be also in the form of a shared (dynamic) library, which can be more efficiently loaded by the slave program. As an alternative to native binary programs for a specific platform, a task can also be an interpreted or precompiled program. For example, it can be a compiled Java application or an interpreted program (e.g. Perl, Python), which further needs, respectively, a Java Virtual Machine or a specific interpreter, on the host computer.

Multiple tasks, which are related to each other, can be grouped into a job that is actually what a user submits to the system (see job life in Figure 3). A job can be composed of one or more tasks. Using jobs provides for easier structuring and management of the computational applications for both the user and the system. Each job is assigned uniquely to one user, however, a user can have multiple jobs submitted at the same time to the system. The tasks that correspond to a job can be independent or not at execution time. Tasks can further be divided into subtasks, consisting of finer work units that are executed within a task. Subtasks are used for interactive applications, which require permanent connection between a client and the slaves. They are usually generated at run-time at the client, and sent for execution to an already running task, which can solve them. The main reason for having subtasks is to improve the efficiency of smaller execution units without the overhead of starting a new task each time.

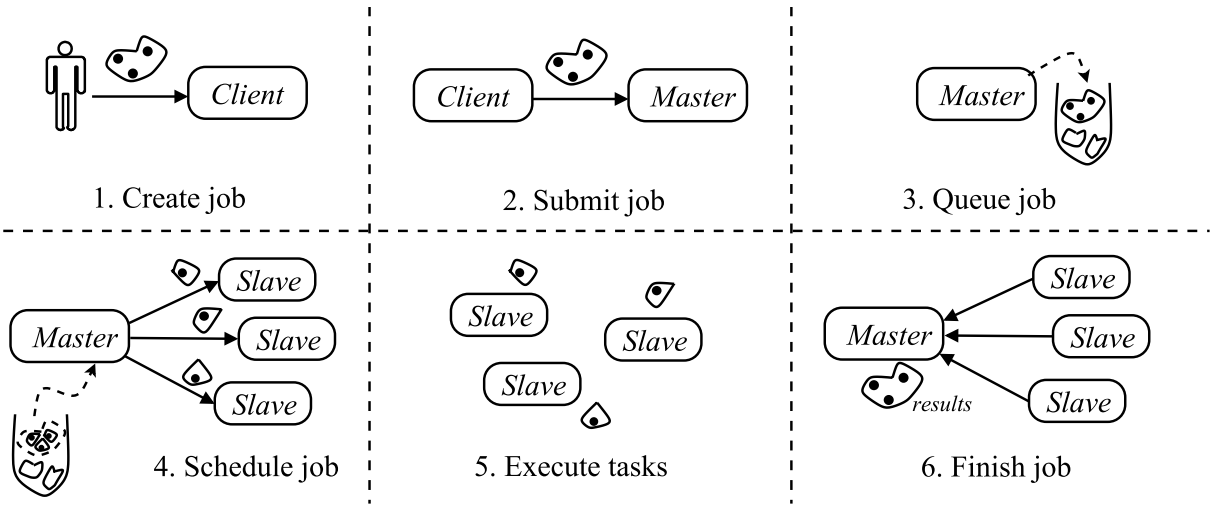

Figure 3: QADPZ job life

4.5 User interface

The QADPZ user interface provides for a user-friendly environment, in which the user can interact with the system. This interaction involves mainly the submission, the monitoring, and the management of the submitted computational applications, along with the resource monitoring and control. The first interface is the job-monitoring interface that is a web-based interface that provides detailed information about all existing jobs in the system. The user can browse the jobs, see their status, and view their component tasks. S/he can also easily create new jobs and tasks. Using this interface, each job can be stopped or deleted (Figure 4). The second interface is also web-based and provides information related to the resources in the system. Basically it gives a list of the slaves registered in the system and their current status (Figure 5). The owner of a desktop computer running a slave is given an interactive application,

which permits easy configuration of the slave. The user has complete control over the slave running on her computer.

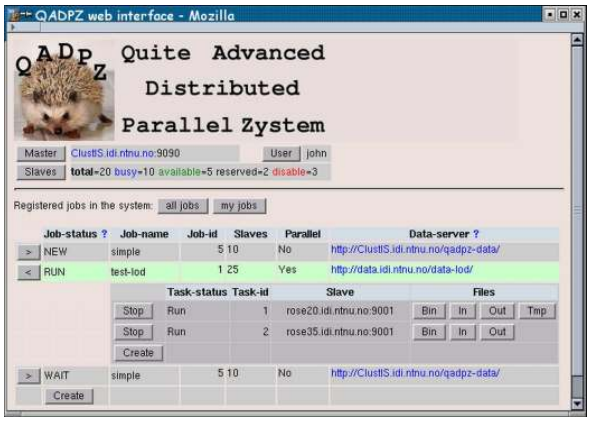

Figure 4: Job-monitoring web interface

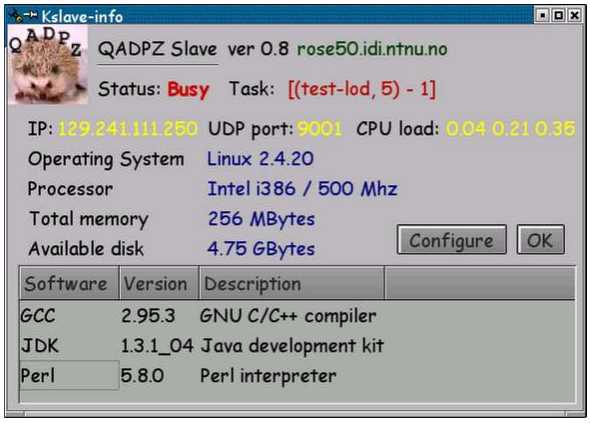

Figure 5: Slave information/configuration interface

5 Security

Because of the unreliability of the UDP protocol, which is our first option for the low-level communication protocol due to its benefits, it is not guaranteed that the execution tasks arriving to the slave computers are undoubtedly sent by the master. This is a serious security threat since it allows for a malicious hacker to submit any piece of code to the slave nodes (IPspoofing). For that reason, and on the cost of a decreased performance, all communication from clients to master and from master to slaves is encrypted and/or signed. Particularly, the data flow from client to master has to be authorized by the name and the password of a QADPZ user, and encrypted using a master public key. A master private key signs the data flow from master to slaves and the authenticity is verified using a master public key on slave nodes.

It is important to note that the data flow from slaves to master and from master to clients is neither encrypted nor signed, which means that a malicious hacker can monitor (packet sniffing) or alter (IP-spoofing) the data or control information arriving back to master or client nodes, and thus put the slave nodes and/or the master node out of operation, modify the resulted data that are submitted by the slaves, or do any other kind of harm to the computational process. In other words, the current QADPZ security scheme is designed to protect the security of the computers in the network, i.e. a malicious hacker cannot submit an alien piece of code to be executed instead of a user computational task. However, this scheme does not protect the QADPZ user data. We plan to provide optional data integrity in the future versions of the system.

Security of the system is handled in two ways. On the one hand, only registered users are allowed to submit applications for execution. This is done by using a user/password scheme, and allows a simple access control to the computational resources. The QADPZ system manages its own user database, completely independent of any of the underlying operating systems, thus simplifying users' access to the system. The QADPZ administrator can create new users by using some supporting tools. On the other hand, security involves the encryption of messages exchanged between various components of QADPZ. This is done by using public key encryption, and provides an additional level of protection against malicious attacks.

6 Conclusions and future work

The present paper reveals the development experience of QADPZ, a desktop grid computing environment. We summarized the main features of the system that make it a powerful platform for running computationally intensive applications. The reasons that have justified the endeavor of developing a new desktop grid platform are also presented. The QADPZ requirements have included all the core capabilities that a successful desktop grid system should provide [12]. We presented the detailed architecture of the system, along with some of the design details. When we started this work, our main goal was to build an easy to use, open source system that provides the complex functionality that users expect from

such a platform [4, 12]. It is worth mentioning that QADPZ has over a thousand users who have already downloaded it [20]. Many of them use it for their daily tasks and we have got valuable feedback from them [4].

Further on we present some future work ideas that aim to improve the QADPZ system:

- many areas of the QADPZ system are incomplete. For example, many large scale parallel problems require checkpointing: running a parallel application for hours or even days and loosing all results due to one failing node is unacceptable;
- data integrity is an important issue, especially in an open environment (Internet);
- improved support for user data security: computation results data can be encrypted and/or signed so that the user of the system can be sure the received data is correct;
- users could be provided with different scheduling algorithms to choose from, according to the type of their problem;
- more complete implementation of the MPI layer and development of a complete library of the collective communication routines;
- adding a set of transparent profiling tools for evaluating the performance of the different components, which is crucially important when running parallel applications;
- decentralizing the system by employing P2P services, which would permit to a group of users to form an ad-hoc set of shared resources; moving towards a P2P architecture;
- interconnection with a grid computing environment that must be decentralized, robust, highly available, and scalable [21], while efficiently mapping application instances to available resources in the system.

These future developments of QADPZ subscribe to the belief that the vision that motivates both Grid and P2P, i.e. that of "a worldwide computer within which access to resources and services can be negotiated as and when needed, will only become real if we are successful in developing a technology that combines important elements of P2P and Grid computing" [1].

Bibliography

[1] I. Foster, and A. Iamnitchi, *On death, taxes, and the convergence of peer-to-peer and grid computing*, in 2nd Int. Workshop on P2P Systems IPTPS 2003, pp. 118-128, 2003.

[2] I. Foster, C. Kesselman, *The grid: blueprint for a new computing infrastructure*, Boston: Morgan Kaufmann, 2004.

[3] J. I. Khan and A. Wierzbicki, Eds., *Foundation of Peer-to-Peer Computing*, Special Issue, Elsevier Journal of Computer Communication, Volume 31, Issue 2, Feb. 2008.

[4] Z. Constantinescu, *A Desktop Grid Computing Approach for Scientific Computing and Visualization*, PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2008.

[5] QADPZ, [online] Available: http://qadpz.sourceforge.net. [Accessed August 1, 2008].

[6] M. Vladoiu, Z. Constantinescu, *An Extended Master-Worker Model for a Desktop Grid Computing Platform (QADPZ)*, in 3rd Int. Conference on Software and Data Technologies -ICSOFT 2008, pp. 169-174, 2008.

[7] Z. Constantinescu, J. Holmen, P. Petrovic, *Using Distributed Computing in Computational Fluid Dynamics*, in 15th Int. Conf. Parallel Computational Fluid Dynamics ParCFD-2003, pp. 123-129, 2003.

[8] Z. Constantinescu, *Towards an autonomic distributed computing environment*, in 14th Int. Workshop on Autonomic Computing Systems, 14th Int. Conf. on Database and Expert Systems Applications DEXA 2003, pp. 694-698, 2003.

[9] D. P. Anderson, J. Cobb , E. Korpela , M. Lebofsky, D. Werthimer, *SETIhome: an experiment in public-resource computing*, Communications of the ACM, vol. 45, no. 11, pp. 56-61, 2002.

[10] SETI@home [online] Available: setiathome.ssl.berkeley.edu [Accessed May 5, 2003].

[11] Distributed.net [online] Available: http://distributed.net. [Accessed May 5, 2008]

[12] M. Vladoiu, Z. Constantinescu, *A Taxonomy for Desktop Grids from Users Perspective*, in Int. Conference on Parallel and Distributed Computing - ICPDC 2008, World Congress on Engineering (WCE 2008), pp. 599-605, 2008.

[13] C. Germain, V. Neri, G. Fedak and F. Cappello, *XtremWeb: Building an Experimental Platform for Global Computing*, in 1st IEEE/ACM Workshop on Grid Computing Grid2000, pp. 91-101, 2000.

[14] D. P. Anderson, *BOINC: A System for Public-Resource Computing and Storage*, in 5th IEEE/ACM International Workshop on Grid Computing, pp. 365-372, 2004.

[15] BOINC - Open Source Software for Volunteer Computing and Grid Computing [online] Available: http://boinc.berkeley.edu. [Accessed November 25, 2007].

[16] J. Basney, M. Livny, *Managing network resources in Condor*, in Proc. of the 9th IEEE Symposium on High Performance Distributed Computing (HPDC9), pp. 298-299, 2000.

[17] Globus [online] Available: http://www.globus.org [Accessed May 15, 2008].

[18] I. Foster, and C. Kesselman., *Globus: A Metacomputing Infrastructure Toolkit*, Intl J. Supercomputer Applications, vol. 11, no. 2, pp. 115-128, 1997.

[19] J. Cassens, Z. Constantinescu, *Free Software: An Adequate Form of Software for Research and Education in Informatics?*, in LinuxTag 2003 Conference, pp. 5-10, 2003.

[20] Sourceforge, [online] Available: http://sourceforge.net [Accessed April 1, 2008].

[21] F. Berman, G. Fox, A.J.G. Hey, *Grid computing: making the global infrastructure a reality*, New York: J. Wiley, 2003.

Monica Vlădoiu got her MSc (1991) and PhD (2002) in the Department of Computer Science of The Polytechnic University of Bucharest, Romania. Since then, she has been with the Dept. of Informatics, Petroleum-Gas University of Ploieşti (UPG), Romania. Her main research interests include digital libraries, learning objects, multimedia databases, reflective and blended learning, desktop grid computing and e-society. She has published over 30 research papers concerning these topics and she has (co-) authored 3 books.

Zoran Constantinescu got his MSc (1997) in the Dept. of Computer Science of The Polytechnic University of Bucharest, Romania. Since then, he has been working both in the software engineering industry and in Higher Education. He got his doctoral degree in Computer Science (2008), from The Norwegian University of Science and Technology, Trondheim, Norway. His research interests include parallel and distributed computing, desktop grid computing, GPS systems and embedded systems. He has published over 20 research papers dealing with the above mentioned topics.