Crossroads
The ACM Student Magazine

Q²ADPZ: An Open Source, Multi-platform System for Distributed Computing

by ***Zoran Constantinescu*** and ***Pavel Petrovic***

Introduction

The recent increases of computational power in desktop computer's has engendered their widespread use in organizations with large computational needs, such as universities and research centers. Several years ago, such centers would invest in special-purpose high-performance servers. Today several hundred linked pentium workstations provide CPU power comparable to the most powerful parallel computer in Norway. Another option, **Distributed Computing**, harnesses the idle processing cycles of networked workstations to attack computationally intensive problems that would otherwise require a supercomputer or a dedicated cluster of computers.

A distributed computing problem is divided into small, independent computing tasks, which are then distributed to the workstations to be processed in parallel. Results are returned to the server, where they are collected. The more PCs in a network, the more processors available to process applications in parallel, and the faster the application can be executed. A network of a few thousand PCs can process applications that otherwise can be run only on fast and expensive supercomputers. This kind of computing can transform a local network of workstations into a virtual supercomputer.

Several public systems for distributed computing have appeared [4]. Most of these systems, however, are either focused on some specific problems, require dedicated hardware, or are proprietary, offering very little flexibility to developers.

Distributed.net [8] is a very large network of users all over the world, using their computers' idle processing cycles to run computational challenge projects. Examples of projects include the RC5-64 secret-key, and DES/CS-Cipher challenges. Another similar project is SETI@home [11], a scientific experiment that uses Internet-connected computers in the Search for Extraterrestrial Intelligence (SETI), by analyzing radio telescope data. The user installs a small, problem specific client program, which is either in the form of an executable or a screen saver. This program connects to the project's server and downloads a set of data for analysis. The result is later uploaded to the server. Both systems focus on very specific problems, and the unavailability of source code makes them difficult, if not impossible to use for our research projects.

Entropia [9] is a similar, but commercial version of distributed computing over the Internet. It offers a robust technology and assistance with expertise in a seamless integration into existing network environments and in a deployment of custom applications. However, many of our academic research projects cannot afford extra costs.

Condor [3] is a high throughput computing environment that can manage very large collections of distributed workstations. The environment is based on a layered architecture that enables it to provide a powerful and flexible suite of resource management services to sequential and parallel applications. The maturity of Condor makes it appealing, however the restrictive license, complicated installation procedure, and limited platform availability made it impossible to adapt to our requirements.

A Beowulf cluster [1] is built out of commodity hardware components, running a free-software operating system like Linux or FreeBSD, interconnected by a private high-speed network. It is a dedicated cluster for running high-performance computing tasks. The nodes in the cluster don't sit on people's desks; they are dedicated to running cluster jobs. Usually, in a dedicated cluster environment, all computers are running the same operating system.

Q^2ADPZ

Our goal was to use computers from our labs for our CPU-intensive research projects from the areas of large-scale scientific visualization, evolutionary computation, and simulation of complex neural

network models. We had available computers running many different operating systems: Linux, FreeBSD, MacOS X, Windows and Solaris. Since we couldn't find a distributed computing system to fit our needs, we decided to build our own, open-source system for connecting the available computers.

We will describe our system, Q^{2}ADPZ ['kwod pi: 'si:] (Quite Advanced Distributed Parallel Zystem), a modular, open-source implementation in C++ of a multi-user and multi-platform system for distributed computing on non-dedicated desktop computers. This is a research project developed by graduate students in our department (Department of Computer Science, Norwegian University of Science and Technology (NTNU), Trondheim).

As such, it is developed as an open-source project. Free availability of the source code allows its flexible installations and modifications based on the individual needs of research projects. In addition to being a very powerful tool for computationally-intensive research, the open-source availability makes it a flexible educational platform for numerous small-size student projects in the areas of operating systems, distributed systems, mobile agents, parallel algorithms, and possibly others.

The design goals of the Q^{2}ADPZ system are ease of use at different user skill levels, inter-platform operability, client-master-slave architecture using fast, message-based communication, modularity and modifiability, security, and simplicity of installation and upgrade on the computing nodes.

In Q^{2}ADPZ, a small software program (called *slave service*) runs on each desktop workstation. As long as the workstation is not in use, the slave service accepts tasks sent by the server (called *master*). The available computational power is used for executing a task. Only minimal human system administration is required.

Each installation of the system requires a Q^{2}ADPZ administrator, who is responsible for configuring the system and installing the slave service on desktop computers, and the universal client on user computers. In principle, the administrator doesn't need administrator rights to install the slaves (except for the Windows platform), since the UNIX slave daemon can run with simple user rights. Individual users, however, do not need to have any knowledge about the system internals. On the contrary, they are able to simply submit their tasks. A task can be either a binary executable, a dynamic library, an intermediate binary program (for example Java byte-code), or an interpreted program (for example Perl, Python).

The description of a task can contain:

- number of runs of the application,
- file path to the executable and command line arguments,
- input and output files (their full names are automatically generated from the run number)
- directories where the files reside,
- utilities to be run after individual tasks (typically to process the output files before another task is started),
- maximum time allowed for a task to execute,
- hardware (disk, memory, CPU type and speed) and software (operating system, and installed programs) requirements of the application.

Here is a very simple example of a job description:

```
<Job Name="example">
  <Task ID="1" Type="Library">
    <RunCount>1</RunCount>
    <TaskInfo>
      <Memory Unit="MB">64</Memory>
      <Disk Unit="MB">5</Disk>
      <TimeOut>3600</TimeOut>
      <OS>Linux</OS>
      <CPU>i386</CPU>
      <URL>http://server/lib-example.so</URL>
    </TaskInfo>
  </Task>
</Job>
```

These project configuration parameters are saved into an XML-structured file. The executable can be taken from a local disk or downloaded from any URL-specified address. The input and output data files are automatically transferred to slaves using a dedicated data web server. The progress of the execution can be viewed in any web browser.

Each run corresponds to a task - the smallest computational unit in Q^2ADPZ. Tasks are grouped into jobs - identified by a group name and a job number. The system allows control operations on the level of tasks, jobs, job groups, or users. If preferred by advanced users, the project file may be edited manually or generated automatically (see the two job description examples).

More advanced users can write their own client application that communicates directly with the master

using the API of the *client service library*. This allows submitting tasks with appropriate data dynamically.

Finally, advanced users can write their own slave libraries that are relatively faster than executable programs and very suitable for applications with many short-term small-size tasks, i.e., with a higher degree of parallelism.

A more advanced example of a job description is given below:

```
<Job Name="brick">
  <Task ID="1" Type="Executable">
    <RunCount>15</RunCount>
    <FilesURL>http://server/cgi-bin/</FilesURL>
    <TaskInfo>
      <TimeOut>7200</TimeOut>
      <OS>Win32</OS>
      <CPU Speed="500">i386</CPU>
      <Memory Unit="MB">64</Memory>
      <Disk Unit="MB">5</Disk>
      <URL>http://server/slave_app.dll</URL>
      <Executable Type="File">../bin/evolve_layer.exe</Executable>
      <CmdLine>sphere.prj 2 50</CmdLine>
    </TaskInfo>
    <InputFile Constant="Yes">sphere.prj</InputFile>
    <OutputFile>sph/layout/layout.2</OutputFile>
    <InputFile Constant="Yes">sph/sphere.1</InputFile>
    <InputFile Constant="Yes">sph/sphere.2</InputFile>
    <InputFile Constant="Yes">sph/sphere.3</InputFile>
    <OutputFile>sph/logs/evolve_layer.log.2</OutputFile>
    <InputFile>sph/layout/layout.1</InputFile>
  </Task>
</Job>
```

The communication between the system components is human readable XML and can optionally be saved into log-files so all activity. Extensive debug logs can be produced as well. The system provides basic statistics information on usage accounting.

Interplatform operability

Inter-platform operability is achieved by the pool of computers in a network that can run different operating systems and have different hardware architectures. Q^{2}ADPZ handles task submissions with platform specifications, and the appropriate library or executable is automatically used. Currently, we use a daemon process on Unix environments, or a system service on Windows. At the time of writing, we have successfully tested the system on the following hardware platforms: Linux (iX86,sparc, sparc64), FreeBSD (iX86), SunOS (sun4m,sun4u), IRIX64 (IP27,IP35), Win32 (iX86), and MacOS X (Darwin). Most of the code is ANSI C++ and POSIX.1 compliant and therefore porting to a new platform does not require much effort. We use the POSIX threads API for the UNIX environments, and the native threads in Windows.

All three main components of the system - client, master, and slave have their configuration files, which are well-documented and pre-configured for normal operation (only the IP address of the master needs to be modified). Each user of the system is authorized by means of a user name and password. A special administration utility for their maintenance is provided. Manual configuration of the data web server and master automatic startup is currently required, however automatic installation of the slave service on multiple workstations is possible both for UNIX and Windows platforms.

Upgrade of the slave service is automatic, and can be started by an administration utility program - a new version of the program is downloaded and started by each slave service. This allows large number of network computers to be easily integrated.

The computers submitting jobs (the clients) can be off-line while the tasks are running on slave machines. The master keeps track of the jobs and caches computation results when needed. In addition to a flexible storage place for the pre- or post-computational data, computational nodes can use common Internet protocols for data transfer to or from any other computers, including those not involved in the Q^{2}ADPZ system.

Tasks are automatically stopped or moved to another slave when a user logs on to one of the slave workstations. The system does not support job checkpointing yet and does not handle restart of master computer. Adding these features has high priority. However, tasks can be moved from one slave to another at the request of the running task. This is equivalent to resubmitting a task with the addition that initial input data can be different from the original task.

The system installation, administration and use, as well as system internals are documented in the manual that is available from the project's web page.

Security

There are two conceptually different parts to security: system integrity and data integrity. In Q^{2}ADPZ, we have primarily focused on system integrity, meaning it is not possible to use the system to gain access to any of the machines involved. Only registered users are able to upload code to the slave machines.

In order to reach this requirement, the master contains a private/public key pair. All commands from clients to the master are signed with a username/password pair, so that only registered users can submit work. The passwords are saved in an encrypted form on the master host system. The transmission of the username with password is always encrypted. Likewise, all commands from the master to the slaves are signed using the master's private key. The slave verifies the messages arriving from the master. As a consequence, the computers that run the slave service are guaranteed to be save against the misuse, provided that the security policy is enforced correctly. The key-pair is defined at install-time. This security is guaranteed even in the environment where malicious hackers subvert TCP/UDP packets. Slave user code access to the system is defined by the owner of the system hosting the slave, and is thus outside Q^{2}ADPZ control.

The slave can be requested to download code-blocks from other locations. These locations are also outside the control of Q^{2}ADPZ. This means that if the system administrators of slave hosts give the software unnecessary system access, these computers will be vulnerable to unlawful users and to users ignorant of security issues. We pay this price for flexibility. In our setup, the slave is started under separate network user that has the disk read and write access only in a special temporary directory.

Architecture

The system consists of a central process - the *master*, a variable (high) number of computing processes on different computers in the network - the *slaves*, and a number of *client* processes, or user applications, which generate tasks grouped in jobs.

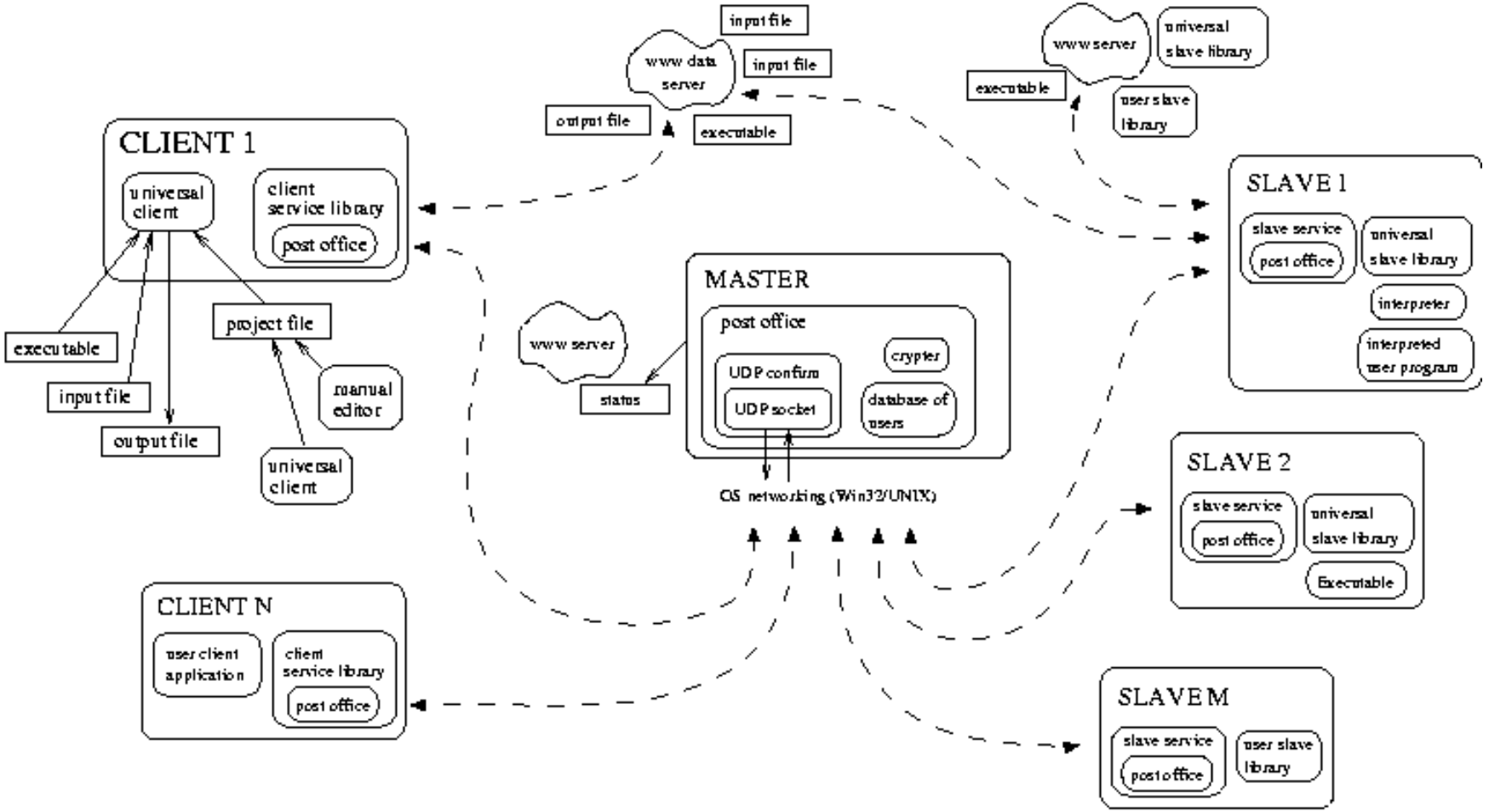

Figure 1: Architecture of Q^{2}ADPZ system.

The slave component is run as a daemon (Unix) or system service (Win32). Its main role is to notify the central master about its status and the available resources on the slave, including:

- operating system type
- processor information: CPU type, CPU speed
- physical memory available
- local disk available
- existing software on the local system

An example of slave status message is shown here:

```
<Message Type="M_SLAVE_STATUS">
```

```
  <Status>Ready</Status>
  <SlaveInfo>
    <Version>0.5</Version>
    <OS>Win32</OS>
    <CPU Speed="500">i386</CPU>
    <Memory Unit="MB">32</Memory>
    <Disk Unit="MB">32</Disk>
    <Software Version="1.3.0">JDK</Software>
    <Software Version="2.95.2">GCC</Software>
    <Address>129.241.102.126:9001</Address>
  </SlaveInfo>
</Message>
```

Another role of the slave is to launch an application (task) at the master's request. The application, which takes the form of a library, executable, or interpreted program, is transferred from a server according to the description of the task, then it is launched with the arguments from the same task description.

In the case of executable and interpreted tasks, the universal slave library is used. After it is launched by the slave service library, it first downloads the executable or the interpreted program, either from an automatic data storage system or from a specified URL location. The universal library proceeds with downloading and preparing all the required input files. After the executable or interpreted program terminates, the generated output files are uploaded to the data store to be picked up later by the universal client, which originated the task.

On Win32 based platforms, the user slave libraries come in form of a DLL module, while on UNIX platform they are dynamic shared libraries.

The master is listening to all the slaves. This way, it has an overview of all the resources available in the system, similar to a centralized information resource center. It accepts requests for tasks from clients and assigns the most suitable computational nodes (slaves) to them. The matching is based on task and slave specifications and the history of slave availability. In addition, the master accepts reservations for serial or parallel groups of computational nodes: clients are notified after resources become available. The master generates a report on the current status of the system either directly on a text console - possibly redirected to a (special) file, or in form of an HTML document.

The client consists of a client service library and a client user application or the universal client application. The client service library provides a convenient C++ API for a communication with the master, allowing controlling and starting jobs and tasks and retrieving the results. Users can either use this API directly from their application or utilize the universal client, which submits and controls the tasks based on an XML-formatted project file. In version 0.6 of the system, each job needs a different client process, although we are working on extending the client functionality to allow single instance of client to optionally connect to multiple masters and handle multiple jobs.

Communication in Q^{2}ADPZ is based on TCP/UDP, an unreliable communication protocol, in which packets are not guaranteed to arrive and if they do, they may arrive out of order. The advantage of UDP/IP over TCP/IP is that UDP is fast, reducing the connection setup and teardown overhead, and is connectionless, making the scalability of the system much easier. The higher-level protocol is message based, and the messages exchanged between the components of the system are of a small size. Also, messages are exchanged only for control purposes. This makes UDP a very good option for our low-level communication protocol. This layer, called *UDPSocket*, is also responsible for hiding operating system specific function calls, and making a more general interface for communication.

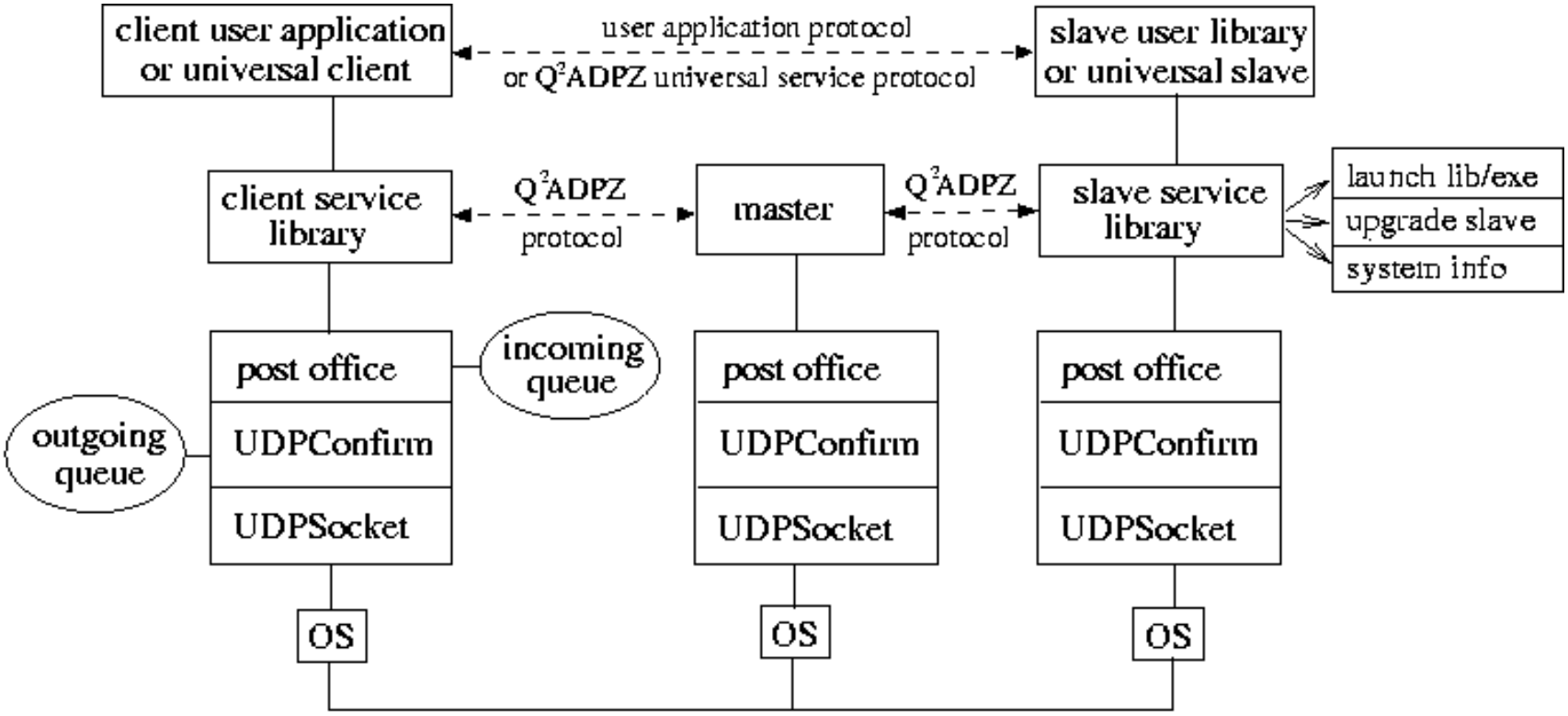

Figure 2: Communication in Q^{2}ADPZ system.

Because of the unreliable nature of the UDP protocol, an additional, more reliable level of communication is needed. This is based on message confirmation. Each message contains a sequence number, and each time it is sent, it is followed by an acknowledgment from the receiver. Each sent or received message is accounted, together with the corresponding acknowledge, and in case of not receiving an acknowledgment, the message is resent a few more times. An acknowledge and a normal message can be combined into one message to reduce the network traffic. This layer is called *UDPConfirm*, and permits both synchronous and asynchronous message sending.

The next communication layer, *PostOffice*, has a similar functionality as the real life post office service. It delivers and receives high level messages - XML elements represented as instances of XMLData class. Both blocking and non-blocking modes are supported. Messages have a source and a destination address. Received messages can be kept by the PostOffice as long as needed, the upper layers in the system having the possibility to retrieve only certain messages, based on the sender's address. The PostOffice is also responsible for the encryption and decryption of messages, if necessary.

Messages exchanged are in XML format, in accordance with a strictly defined communication protocol between client and master, and between master and slave. Each message is represented as an XML element <Message Type="message_type"> (see for example the SLAVE_STATUS XML-message above). XML elements are internally stored as objects of class XMLData, which in turn contain their sub-elements - other XMLData objects. Element attributes are instances of XMLAttrib class. These classes provide extensive functionality for manipulation with XML elements including input/output string and stream operations. When the data for slave user library are sent in the message, they are encapsulated inside of standard <![CDATA[]]> XML elements. We chose to implement our own lightweight XML module in order to achieve high efficiency, flexibility and easy extensibility of its functionality.

Message-based communication is used only for controlling the entities in the system. Shared libraries and executable files for task execution on the slaves, as well as data files for the computations, are transferred using standard Internet protocols, for example http, ftp, ldap, etc. For this, we are using the open source **cURL** library [7]. Currently, the slave is using http to download files from a server (which can be the master itself, or another, specialized data server), but this can easily be changed to a different protocol.

Evaluation

To evaluate the system, we employed the version 0.6 of the system in artificial evolution of layers of

3D LEGO models [5]. A 3D model was decomposed into individual layers. The layout of each layer, i.e., the placement of LEGO bricks was evolved by a separate task. The input and output files were automatically transferred by the universal client as specified by the project file. To obtain statistically significant data, tens of independent runs were required. Q^2ADPZ installation included 70 Pentium-III/733MHz workstations located in a student laboratory. Their status is on and idle during the night, and except for the exercise deadline season approximately 30-50% idle also during the day.

Figure 3 shows an example status of computational progress. The overall statistics shown below the title are followed by a list of active jobs (only one: brick(81;pavel) is shown) with the list of running and waiting tasks belonging to the job. Below that is a list of slave computers identified by their IP:port address together with basic statistical information. Those slaves, where the task column shows a task identifier, are assigned a computational task. This on-line available status is periodically updated and available for the users to view the progress of their computational jobs. We received results worth many weeks of single CPU time within approximately 3 days with no configuration overhead, simply by submitting our executable to Q^2ADPZ.

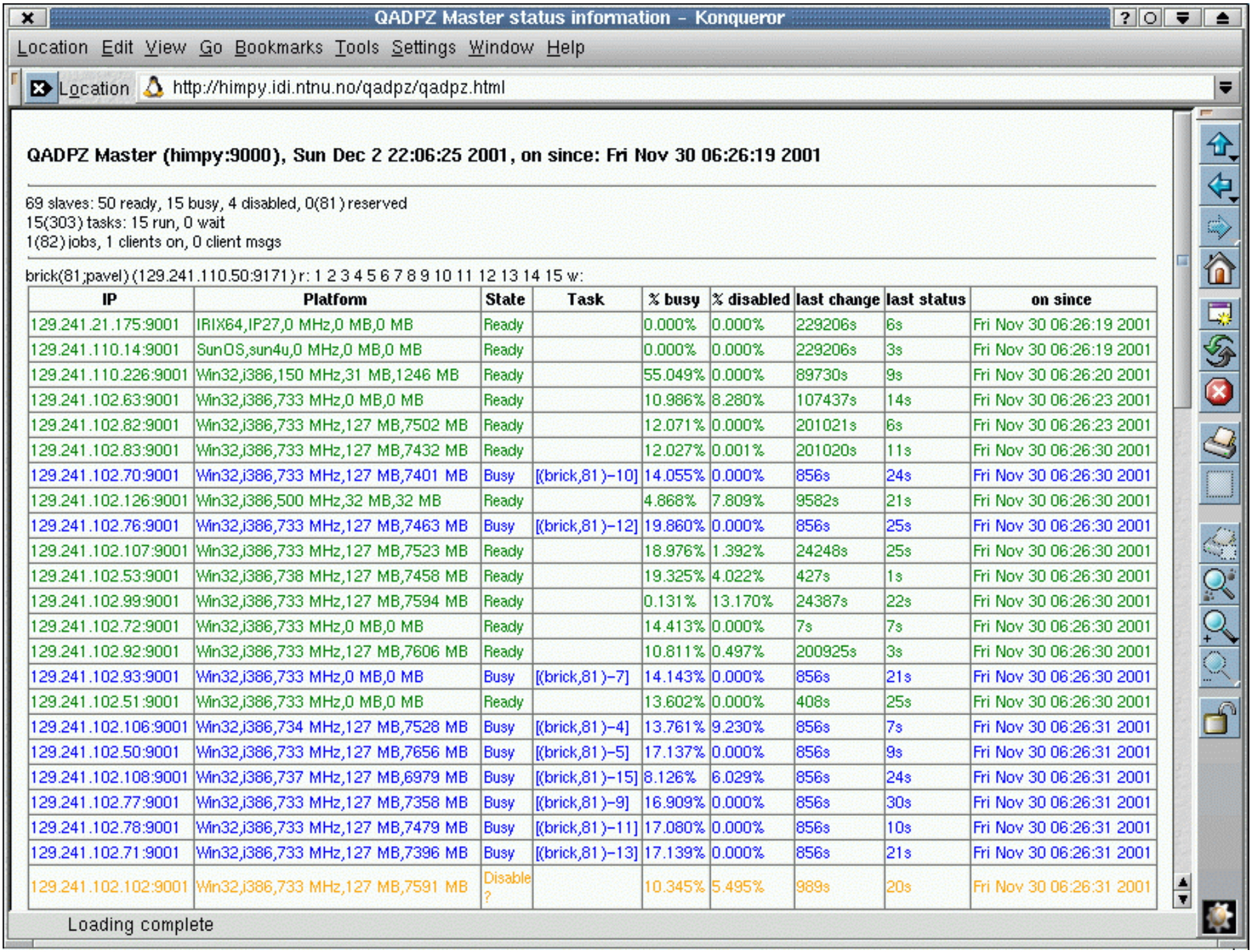

Figure 3: Example Status of Computational Progress in a Q^{2}ADPZ system.

The development of the system was done simultaneously in Linux and Windows environments. This made the integration of different platforms much easier and helped us to find the flaws in the source code faster. The design was made with help of UML diagrams, and the BSCW (Basic Support for Collaborative Work) system from FIT and OrbiTeam Software for keeping track of design documents and development discussion material.

Conclusions and future work

Q^2ADPZ is a free, open-source, multi-platform system for distributed computing in an IP network. It allows users to submit tasks to be computed on idle computers in the network. Q^2ADPZ's design goals include user-friendliness, inter-platform operability, client-master-slave architecture using XML message-based communication, modularity and modifiability, and security of the computers participating in the system.

The latest version is in beta testing using a set of student lab computers at the Department of Computer and Information Science at Norwegian University of Science and Technology with research projects in Visualization and Evolutionary Algorithms. The structure of the implementation of the system is modular and encourages reuse of useful system components in other projects.

Future development of the system will include improved support for user data security. Computation results data will be encrypted and/or signed so that the user of the system can be sure the received data is correct. This is especially useful if the system is used in an open environment, for example over the Internet.

For faster performance, slave libraries will be cached at slave computers - in the current version, they are downloaded before each task is started. A flexible data storage available to other computers in Q^2ADPZ will be provided by slave computers. The scheduling algorithm of the master needs improvements. We plan to support more hardware platforms and operating systems.

The current user interface to the system is based on C++. Possible extensions of the system could be implemented in other languages, for example Java, Perl, Tcl or Python. This can easily be done, since the message exchanges between different components of the system are based on an open XML specification. We invite the interested developers in the open-source community to join our development team. We appreciate any kind of feedback. The current implementation is available from the project's home page [**6**].

References

1

Becker, D. J., Sterling, T., Savarese, D., Dorband, J. E., Ranawak, U. A., & Packer, C. V. *Beowulf: A Parallel Workstation for Scientific Computation.* Proceedings of the 1995 International Conference on Parallel Processing (ICPP), 1995.

2

Constantinescu, Z., Petrovic, P., & Pedersen, A. *Q²ADPZ * An Open System for Distributed Computing.* NordU2002 Conference, Helsinki, Finland, 2002.

3

Litzkow, M., Livny, M., & Mutka, M. *Condor - A Hunter of Idle Workstations.* Proceedings of the 8th International Conference of Distributed Computing Systems, 1988.

4

Pearson, K. *Internet Based Distributed Computing Projects.* http://www.aspenleaf.com/distributed.

5

Petrovic, P. *Solving LEGO brick layout problem using Evolutionary Algorithms.* Norsk Informatikkonferanse NIK'2001.

6

Q²ADPZ home page. http://qadpz.idi.ntnu.no, http://sourceforge.net/projects/qadpz.

7

cURL library. **http://curl.haxx.se**.

8

Distributed.net. **http://www.distributed.net**.

9

Entropia. **http://www.entropia.com**.

10

OpenSSL library. **http://www.openssl.org**.

11

SETI@home. **http://setiathome.ssl.berkeley.edu**.

Biographies

Zoran Constantinescu (**zoranc@acm.org**) is a doctoral student in Computer Science at the Norwegian University of Science and Technology, Trondheim, Norway. His research interests include large scale scientific visualization, parallel and distributed programming, clustering, software engineering methodologies. Home page: http://www.idi.ntnu.no/~zoran.

Pavel Petrovic (**petrovic@idi.ntnu.no**) is a doctoral student in Computer Science at the Norwegian University of Science and Technology, Trondheim, Norway. His research interests include Evolutionary Computation, Artificial Intelligence, and Robotics. Home page: http://www.idi.ntnu.no/~petrovic.