Q^2ADPZ: An Open System for Distributed Computing

Zoran Constantinescu
Pavel Petrovic
Atle Pedersen
Norwegian University of Science and Technology
zoran@idi.ntnu.no, petrovic@idi.ntnu.no, atlep@idi.ntnu.no

Abstract

The recent growth of computational power of desktop computers calls for their efficient use in larger organizations, especially those, which need to run computationally intensive tasks, such as universities and research centers. We describe Q^2ADPZ ['kwod "pi: 'si:], a modular, open source implementation in C++ of a multi-user and multi-platform system for idle distributed computing in a TCP/IP network. The computing power of large number of idle desktop computers is utilized by automatically scheduled tasks that are submitted, monitored, and controlled by users. Flexibility of the system is implied by several user application modes. Task software and hardware requirements and input and output files are handled automatically by the system. The tasks can be mobile. Internal communication protocol is based on optionally encrypted XML messages using public/private keys, user names and passwords. We are currently using the system for research tasks in the areas of large-scale scientific visualization, evolutionary computation, and simulation of complex neural network models.

1 Introduction

In many universities and research organizations, the following recent trends can be identified:

1. larger amounts of data are being accumulated and manipulated;
2. hardware performance of desktop computers increases dramatically;
3. new technological advancements stimulate use of computing applications with extreme requirements for computational power;
4. use of computing, simulations, visualizations, and optimization in various research fields and practical applications is accelerating, and leads to very high demands on computing power;
5. the pace of development of high-performance servers hardly equals these trends, but for very high financial costs.

Increasing hardware performance of desktop computers accounts for a low-cost high-performance computing potential that is waiting to be efficiently put in use.

Distributed Computing harnesses the idle processing cycles of the available workstations on the network and makes them available for working on computationally intensive problems that would otherwise require a supercomputer or a dedicated cluster of computers to solve.

A distributed computing application is divided to smaller computing tasks, which are then distributed to the workstations to process in parallel. Results are sent back 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 results are returned. 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.

Q^2ADPZ is a research project developed by graduate students in our department [1]. The main purpose is to build a system that will be used for obtaining results for the students' and researchers' projects by utilizing idle computers in the student laboratories. As such, it is developed open-source. Free availability of the source code allows its flexible installations and modifications

[1] Department of Computer Science, Norwegian University of Science and Technology (NTNU), Trondheim

based on the individual needs of research projects and institutions. 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 others.

In the remaining sections of this article, we will present a short overview of similar distributed computing systems as compared to Q^2ADPZ, describe the features of our system in detail, explain the architecture and implementation, show an example application and finally conclude with ideas for future development.

2 Related Work

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

Distributed.net [webb] is a very large network of users all over the world, using their computers' idle processing cycles to run computational challenge projects, which require a lot of computing power. Examples of projects are RC5-64 secret-key, DES or CS-Cipher challenges. Another similar project is Seti@home, [webe], 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. The focus on very specific computational problems, and the closed code of the client makes the above mentioned systems difficult to use for our research projects.

Entropia [webc] is a similar, but commercial version of distributed computing over the Internet. They offer 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 such a high cost.

Condor [Mic88] is a high throughput computing environment that can manage very large collections of distributively owned 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 very appealing for our projects, however the very restrictive license of the software makes it almost impossible to adapt to our requirements.

A Beowulf cluster [Don95] 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.

3 Features

The design goals of the Q^2ADPZ 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 of computers participating in Q^2ADPZ, and easy and automatic install and upgrade.

In Q^2ADPZ, a small software program (*slave service*) runs on each desktop workstation. As long as the workstation is not being utilized, the slave service accepts tasks sent by the server (*master*). The available computational power is used for executing a task. Human system administration required for the whole system is minimal. We will now describe the features in detail.

3.1 User modes

Each installation of the system requires a local administrator, who is responsible for configuring the system and installing the *slave service* on desktop computers, and the *universal client* on user computers. Individual users, however, do not need to have any knowledge about the system internals. On the contrary, they are able to simply submit their executable or interpreted (such as Lisp or Java) program from a menu-driven command-line application, where they can specify

- number of runs of the application,
- file path to the executable and command line arguments,
- input and output files (their names are automatically generated from the run number) – either for all runs or for specified subset of runs,

```
<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>
```

Figure 1: Simple library-type project file.

- 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,
- in what order ought the task groups be executed,
- hardware (disk, memory, CPU type and speed) and software (operating system, and installed programs) requirements of the application.

These project configuration parameters are saved into 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 www-server. The progress of execution can be viewed in any www-browser. (see the figure 6).

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. 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 figure 1 and 2 for examples.

More advanced users can write their own client application that communicates directly with the master using 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 high degree of parallelism.

```
<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>64</Memory>
   <Disk>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>
```

Figure 2: Simple executable-type project file.

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

3.2 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^2ADPZ 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, and Win32/iX86. Most of the code is ANSI C++ and POSIX.1 compliant and therefore porting to a new platform does not require too much efforts. We use the POSIX threads API.

3.3 Installation and maintenance features

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 user name and password and a special administration utility for their maintenance is provided. Manual configuration of the data www-server and master automatic startup is currently required, however automatic installation of the slave service on multiple PC workstations is solved for Win32/iX86 platform and is easy to setup for UNIX platforms.

Upgrade of the slave service is automatic, it is started by administration utility program – a new version 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 offline while their 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 computer, including those not involved in the Q^2ADPZ 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 www-page.

4 Security

There are two conceptually different parts about security: system integrity and data integrity. In Q^2ADPZ, we have primarily focused on system integrity, meaning it should not be possible to use the system to gain access to any of the machines involved. Based on this we have the following requirements:

- only registered users should be able to upload code to the slave machines
- slave code has limited access to the host environment

In order to reach the first requirement, the master is fitted with a private/public key pair using the OpenSSL library [webd]. 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 key-pair is defined at install-time. Slave code access to the system is defined by the owner of the system hosting the slave, and is thus outside Q^2ADPZ control.

The slave can be requested to download codeblocks from other locations. These locations are also outside the control of Q^2ADPZ. 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.

5 Architecture

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

Slave component is run as a daemon or Windows service. Its first role is to notify the central master about its status and the available resources. These include:

- operating system type
- processor information: CPU type, CPU speed

```
<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>
```

Figure 3: Slave status message is sent from all computational nodes in regular intervals.

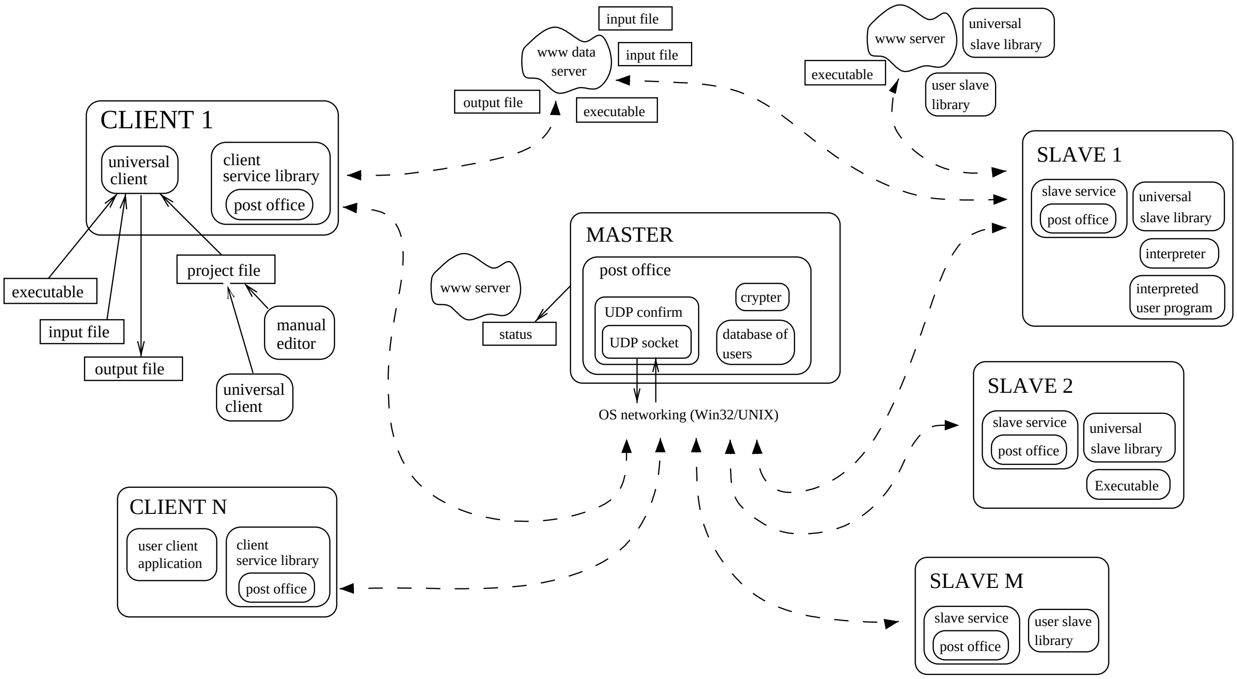


Figure 4: Q^2ADPZ architecture.

- physical memory available
- local disk available
- existing software on the local system

An example of slave status message is shown at the figure 3.

Another role of the slave is to launch an application (task) as a consequence of master's request. The application, in 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 case of executable and interpreted tasks, *universal slave library* is used. After it is launched by the slave service library, it first downloads the executable or interpreted program, either from an automatic data store (now implemented on top of www-server in Perl), 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 platform, the user (or universal) slave libraries come in form of DLL module, while on UNIX platform they are dynamic libraries (this makes it difficult to port the application for example to Darwin/Mac OS, which doesn't support dynamic 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, master accepts reservations for serial or parallel groups of computational nodes: clients are notified after resources become available. Master generates a report on 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 the 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 sub-

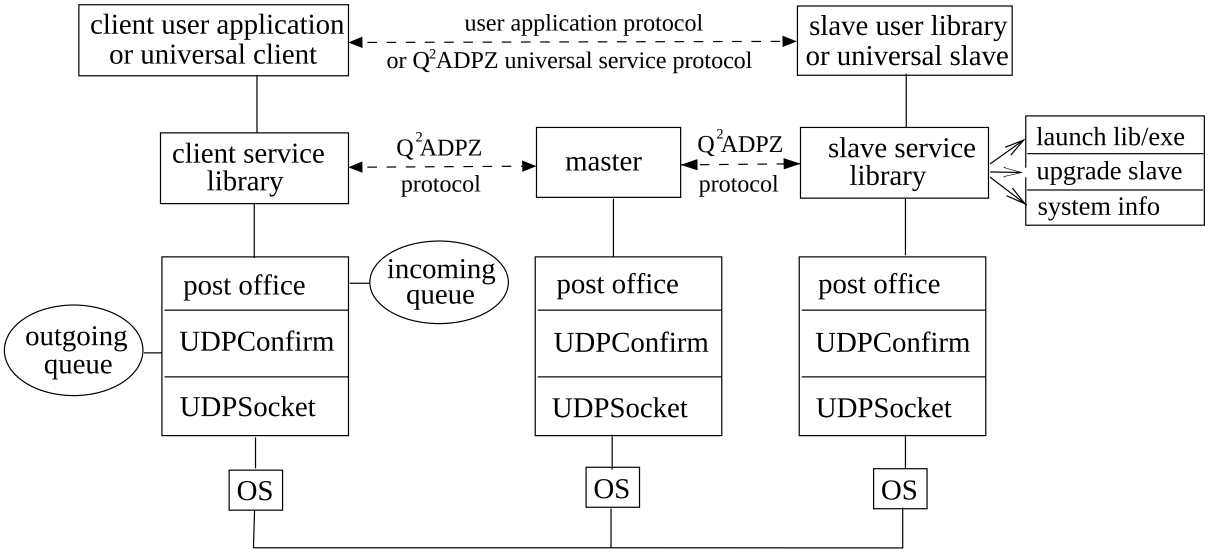


Figure 5: Q^2ADPZ communication layers.

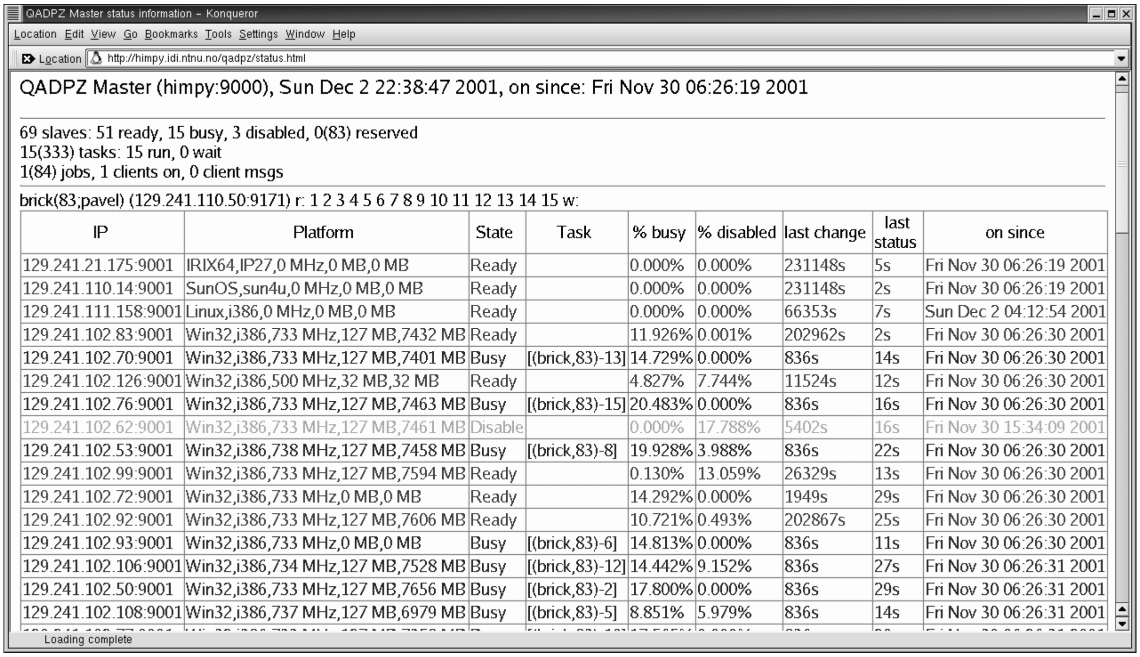
mits and controls the tasks based on an XML-formatted project file. In version 0.6 of the system, each job needs a different client process, although we are working on extending the client functionality to allow single instance of client to optionally connect to multiple masters and handle multiple jobs.

Communication in Q^2ADPZ 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 easier. The higher-level protocol is message based, and the size of the messages exchanged between the components of the system is small. 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.

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 acknowledgement from the receiver. Each sent or received message is accounted, together with the corresponding acknowledge, and in case of not receiving an acknowledgement, 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 the figure 3. XML elements are internally stored as objects of class XMLData, which in turn contain their subelements – 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 class in order to achieve flex-


QADPZ Master status information – Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Location http://himpy.idi.ntnu.no/qadpz/status.html
Loading complete

QADPZ Master (himpy:9000), Sun Dec 2 22:38:47 2001, on since: Fri Nov 30 06:26:19 2001

69 slaves: 51 ready, 15 busy, 3 disabled, 0(83) reserved
15(333) tasks: 15 run, 0 wait
1(84) jobs, 1 clients on, 0 client msgs

brick(83;pavel) (129.241.110.50:9171) r: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 w:

IP	Platform	State	Task	% busy	% disabled	last change	last status	on since
129.241.21.175:9001	IRIX64,IP27,0 MHz,0 MB,0 MB	Ready		0.000%	0.000%	231148s	5s	Fri Nov 30 06:26:19 2001
129.241.110.14:9001	SunOS,sun4u,0 MHz,0 MB,0 MB	Ready		0.000%	0.000%	231148s	2s	Fri Nov 30 06:26:19 2001
129.241.111.158:9001	Linux,i386,0 MHz,0 MB,0 MB	Ready		0.000%	0.000%	66353s	7s	Sun Dec 2 04:12:54 2001
129.241.102.83:9001	Win32,i386,733 MHz,127 MB,7432 MB	Ready		11.926%	0.001%	202962s	2s	Fri Nov 30 06:26:30 2001
129.241.102.70:9001	Win32,i386,733 MHz,127 MB,7401 MB	Busy	[(brick,83)-13]	14.729%	0.000%	836s	14s	Fri Nov 30 06:26:30 2001
129.241.102.126:9001	Win32,i386,500 MHz,32 MB,32 MB	Ready		4.827%	7.744%	11524s	12s	Fri Nov 30 06:26:30 2001
129.241.102.76:9001	Win32,i386,733 MHz,127 MB,7463 MB	Busy	[(brick,83)-15]	20.483%	0.000%	836s	16s	Fri Nov 30 06:26:30 2001
129.241.102.62:9001	Win32,i386,733 MHz,127 MB,7461 MB	Disable		0.000%	17.788%	5402s	16s	Fri Nov 30 15:34:09 2001
129.241.102.53:9001	Win32,i386,738 MHz,127 MB,7458 MB	Busy	[(brick,83)-8]	19.928%	3.988%	836s	22s	Fri Nov 30 06:26:30 2001
129.241.102.99:9001	Win32,i386,733 MHz,127 MB,7594 MB	Ready		0.130%	13.059%	26329s	13s	Fri Nov 30 06:26:30 2001
129.241.102.72:9001	Win32,i386,733 MHz,0 MB,0 MB	Ready		14.292%	0.000%	1949s	29s	Fri Nov 30 06:26:30 2001
129.241.102.92:9001	Win32,i386,733 MHz,127 MB,7606 MB	Ready		10.721%	0.493%	202867s	25s	Fri Nov 30 06:26:30 2001
129.241.102.93:9001	Win32,i386,733 MHz,0 MB,0 MB	Busy	[(brick,83)-6]	14.813%	0.000%	836s	11s	Fri Nov 30 06:26:30 2001
129.241.102.106:9001	Win32,i386,734 MHz,127 MB,7528 MB	Busy	[(brick,83)-12]	14.442%	9.152%	836s	27s	Fri Nov 30 06:26:31 2001
129.241.102.50:9001	Win32,i386,733 MHz,127 MB,7656 MB	Busy	[(brick,83)-2]	17.800%	0.000%	836s	29s	Fri Nov 30 06:26:31 2001
129.241.102.108:9001	Win32,i386,737 MHz,127 MB,6979 MB	Busy	[(brick,83)-5]	8.851%	5.979%	836s	14s	Fri Nov 30 06:26:31 2001

Figure 6: List of slaves – status information provided by master.

ibility 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, like for example http, ftp, ldap, etc. For this, we are using the open source library “cURL” [weba]. 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.

6 Evaluation

To evaluate the system, we employed the version 0.6 of the system in artificial evolution of layers of 3D LEGO models [Pav01]. 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 shown at the the figure 2). To obtain statistically significant data, tens of independent runs were required. Q^2ADPZ installation included 70 high-performance PentiumIII/733MHz workstations located in a student laboratory. Their status is on and idle during the night, and except of the exercise deadline season approximately 30-50% idle also during the day.

The figure 6 shows an example status of computational progress. We received results worth many weeks of single computational time within approximately 3 days time with no configuration overhead, by simply submitting our executable to Q^2ADPZ.

The development of the system was done simultaneously in UNIX 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 designed 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. We keep the development sources stored in a CVS system.

7 Conclusions and Future Work

Q^2ADPZ is a free, open-source, multi-platform system with limited security for distributed computing in an IP network. It allows users to submit tasks to be computed on idle computers in the network.

Q^2ADPZ 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 Q^2ADPZ.

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 can 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 would be different interfaces for other languages, e.g. 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 and we appreciate any kind of feedback. The current implementation is available from the project's home page `http://www.idi.ntnu.no/qadpz`.

8 Acknowledgments

This research was supported by the Research Council of Norway and by the Norwegian University of Science and Technology. We are grateful to all members of the Q^2ADPZ team for their work on the project. We also thank our thesis advisors, Richard Blake and Keith Downing, for their support.

References

[Bri98] Brian Hayes. Collective wisdom. In *American Scientist*, 1998.

[Don95] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pages 11–14, 1995.

[Mic88] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

[Pav01] Pavel Petrovic. Solving LEGO brick layout problem using Evolutionary Algorithms. In *Norsk Informatikkonferanse NIK'2001*, pages 87–97, 2001.

[Pea] Kirk Pearson. Internet Based Distributed Computing Projects. `http://www.aspenleaf.com/distributed`.

[Str97] Bjarne Stroupstrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[weba] cURL. `http://curl.haxx.se/`.

[webb] Distributed.net. `http://www.distributed.net/`.

[webc] Entropia. `http://setiathome.ssl.berkeley.edu/`.

[webd] OpenSSL. `http://www.openssl.org/`.

[webe] SETI@home. `http://setiathome.ssl.berkeley.edu/`.