Towards an Autonomic Distributed Computing System

Zoran Constantinescu
Norwegian University of Science and Technology
Department of Information and Computer Science
N-7491 Trondheim, Norway, zoran@idi.ntnu.no

Abstract

Increasing hardware performance of desktop computers accounts for a low-cost computing potential that is waiting to be efficiently used. However, the complexity of installation and maintenance of a large number of distributed heterogeneous computers is limiting the use of such systems on a large scale. Systems which are autonomic, capable of managing themselves are required. The architecture of QADPZ (Quite Advanced Distributed Parallel Zystem), an open source system for heterogeneous distributed computing is presented. The system allows the management and use of the computational power of idle computers from a network of PCs. Different autonomic computing features of the system are described, together with possible extensions of the system towards an autonomic distributed computing system.

1 Introduction

Increasing desktop CPU power and communication bandwidth are helping to make distributed computing a more practical idea. We consider *distributed computing* as an environment where it is possible to harness idle CPU cycles and storage space of tens or hundreds of networked computers to work together on a particularly computational intensive application. We are interested more in computers from a local area network, or a campus environment, and not computers connected over low-speed network connections over the Internet. The growth of such distributed systems has been limited, however, due to a lack of supporting applications, and because of security, management, and standardization challenges. The number of real supporting applications is still somewhat limited, and the above mentioned challenges are still significant.

We present QADPZ ['kwod "pi: 'si:] (Quite Advanced Distributed Parallel Zystem), an open source system for heterogeneous distributed computing. The system allows a centralized management and use of the computational power of idle computers from a network of PCs. It is possible to run either independent applications, or parallel applications, which can communicate between each other using a subset of the Message Passing Interface (MPI) standard. We describe its current capabilities as an autonomic computing system, together with our future extensions to improve its autonomic capabilities. Different parts of the system support different aspects of a self-managed computing system, mainly knowledge about its resources, self-configuration, self-optimization, and self-healing.

2 Autonomic Computing

IBM's manifesto on autonomic computing [5] points out that the difficulty of managing today's computing systems is not only because of the administration of individual software environments, but also because of the need to integrate multiple heterogeneous environments, and to extend beyond company boundaries into the Internet. All these factors contribute to increased levels of complexity in computing systems. Installing, configuring, and maintaining such large systems is becoming an increased challenge even for experts. A possible solution to this problem is to embed the complexity in the system infrastructure itself (both hardware and software), then automating its management. This is in a way similar to the human system, with its autonomic nervous system, which provides automatic, involuntary regulation of the major physiological functions.

The essence of autonomic computing systems is *self-management*, the intent of which is to free system administrators from the details of system operation and maintenance [6]. In a similar way to the biological systems, autonomic systems will maintain and adjust their operation in the face of changing components, demands, workloads, and external conditions, and also will be able to handle hardware or software failures. Such systems will be able to monitor their use and interact with other systems.

The following is a list of defining characteristics for an autonomic computing system [5]:

- *know itself*: the system should have detailed knowledge of its components, status, capacity, and connections with other systems; it will need to know the extent of its owned resources, those it can lend, and those that can be shared or should be isolated.
- *configure itself*: the system configuration should be done automatically, as must dynamic adjustments to that configuration to handle changing environments.
- *optimize itself*: the system should monitor its components and look for ways to optimize its working, like resource allocations, load balancing, different network traffic optimizations.
- *heal itself*: the system should be able to recover from faults that might cause some parts of it to malfunction.
- *protect itself*: the system should be capable of detecting and protecting resources from both internal and external attacks, thus maintaining overall system integrity.
- *adapt itself*: the system should be aware of its environment and the context surrounding its activity, and act accordingly, by finding rules for how best to interact with neighboring systems.
- *open standards*: the system should work in a heterogeneous environment and implement open standards; it cannot be a proprietary solution.
- *anticipatory*: an autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden; both the users and applications in the system should be unaware of the presence of the technology used to perform their functions.

3 The QADPZ System

3.1 Description

QADPZ is an open source, multi-platform system for distributed computing in a TCP/IP network. 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, scientific computing, and simulation of complex neural network models. We had available computers running many different operating systems: Linux, FreeBSD, MacOS X, Windows and Solaris, and we needed an easy way to use those computers without interfering with the normal administration of them. The design goals of the QADPZ system are ease of use at different user skill levels, inter-platform operability, a client-master-slave architecture using fast, message-based communication, modularity and extensibility, security of the computers participating in the system, and very easy and automatic install and upgrade on the computing nodes.

The system allows the exploitation of the computational power of idle computers in a network. The users can submit, monitor, and control computing tasks to be executed on computers participating in the QADPZ system. We use the notion of *task* to represent a basic computation. A task can take the form of a dynamic shared library, a directly executable program, a program executable by means of a virtual machine (e.g. Java application), or any other interpreted program type (e.g. Perl, Lisp, Python). Multiple tasks can be grouped into jobs, for an easier management by the system. One job is uniquely associated with one user of the system, who can then later monitor and control the execution of the tasks from that job.

Different types of jobs can be submitted to the system. A job can consist of independent tasks, which don't require any kind of communication between each other. This is usually referred to as 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 by using a subset of the most common functions from the Message Passing Interface (MPI) standard.

The system delivers both the input and output data required by the tasks, and provides a shared disk space in the form of a local data server. A web server, or any other service (e.g. ftp) which supports file downloads and uploads can be used as a data server.

3.2 Architecture

The QADPZ system can operate both in conditions of an open Internet environment and of a closed local area network which supports the family of TCP/IP protocols. It was designed as an object-oriented system and implemented using C++. Strong requirements of the system are open source, multi-platform support for both Unix/Linux and Windows systems, simple installation and later maintenance, support for multiple users, security, and the possibility to easily extend the system by adding new features.

The system consists of three types of entities: master, client, and slave. The architecture of the system is shown in figure 1. Each computer contributing with computing power to the system is called a *slave*, and is running a small background process in the form of a UNIX daemon or a Windows system service. The process can be run with the privileges of an ordinary user, it doesn't need to be run with

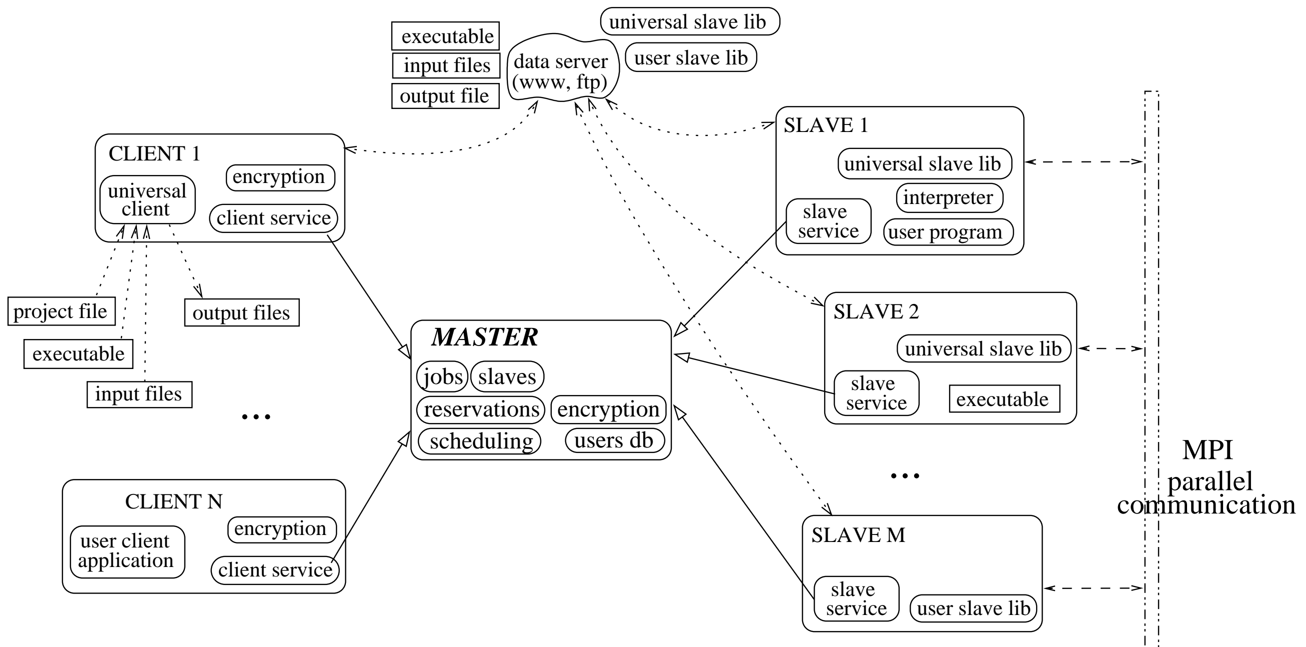

Figure 1. QADPZ architecture.

administrative rights. This process is responsible for reporting the computer's resources and status to a central server, the *master*. It also accepts computational requests from the master, downloads the corresponding binaries and data files for the tasks, executes the task, and then uploads the result files when finished.

The *master* is responsible for managing the available resources, keeping track of the available slaves, their capabilities and configuration. It also schedules the computational tasks submitted by any authorized user of the system, according to the required resources. Tasks can be started, stopped, or rescheduled by the master. Tasks are created by users, who can submit them to the master by means of a *client* as an interface to the QADPZ system.

The *client* is the interface by which a user interacts with the system. It allows the user to create jobs, monitor them, and control their execution. There are two execution modes for the client: a batch mode and an interactive mode. In the batch mode, which can be done using the universal client, tasks are described by a project file, specifying the required resources and how to start the tasks. This information is then sent to the master, which is responsible for scheduling the tasks. The client can detach from the master and connect later for the results. In the interactive mode, the client has much more freedom over the creation and controlling of new tasks: the user can dynamically create new tasks, send messages to already executing tasks, and can receive feedback from the running tasks, either through the master node, or communicating directly with the slaves running the respective tasks. This is more suited for applications where interactivity with the running computation is required.

Communication between different components is message based, using UDP as the underlying communication protocol. This is an unreliable protocol, in which packets are not guaranteed to arrive and if they do, they may arrive out of order. The advantage of UDP over TCP/IP is that UDP is faster, reducing the connection setup and teardown overhead, and is connectionless, making the scalability of the system much easier. Our higher-level communication abstraction implements a reliable, confirmation based message exchange protocol. Messages are represented in an XML format for easier extensibility and interconnection with other potential systems.

4 Autonomic Features

In this section we will describe how the different component types of the QADPZ system manifest autonomic characteristics.

4.1 Self-knowledge

First, the system must have detailed knowledge about itself. In QADPZ this is accomplished by detecting all available computing resources and their current status. Each slave knows about its own local resources, while the mas-

ter knows about all the available resources provided by the slaves contributing to the system.

When the slave background application is started on one of the computers in the network, it automatically detects the hardware and software resources available on that computer. Hardware resources are, for example, system architecture, CPU type and speed, available physical memory, available disk space. These characteristics of the computer can be obtained in different ways: by inquiring the operating system (e.g. the available memory and disk space), or by running some benchmark tests (e.g. CPU speed). Each operating system has its own way of providing such information, so that this auto-detection feature of the slave is dependent on the operating system. However, it is a small part of the code and can be easily adapted for a new system.

Software resources can be, for example, the operating system type and version, different shared system libraries and software applications available on the system. The slave is pre-configured to detect if certain software applications (e.g. compilers, interpreters, etc.) are available, and determines the installed version on that computer.

Using this information, the slave service is creating a description of the computer and registers it to the master. In this way, the master will collect detailed information about each of the slaves participating in the QADPZ system, keeping an overall knowledge about the whole system's resources, thus creating a knowledge about itself.

4.2 Self-configuration

The software running on each slave computer is capable of upgrading itself whenever there is a new version of the software. This is done automatically on the slave side, without any user intervention, or system restart. The user only needs to specify to the master the new version of the slave program and its location for the different operating systems. The master will notify the slaves about the availability of a new version. Each slave will upgrade itself if it has an older version. However, the upgrade can be delayed if a specific slave is running a task, until the computation is finished. Any additional new slave which connects to master will also be notified about a possible upgrade.

4.3 Self-optimization

The slave is also responsible for detecting if the computer is in use by any interactive user, or if the CPU resource is used by other applications. The first situation is detected by monitoring if there is an interactive session started on the computer: in Windows this is done by checking if the *explorer* application is running, while in Unix by checking for an X-Windows session. The second situation is detected by measuring the CPU load over a longer period of time (seconds, a few minutes). In any of these situations, the slave is considered unavailable, and will not be scheduled for executing computational tasks. Once the computer becomes available, its new status is reported to the master and scheduling of tasks becomes possible. This monitoring feature of the slaves is the first step in gathering information about resource utilization for the purpose of self-optimization of the system. The information is used by the master for scheduling the distribution of tasks to the slaves.

4.4 Self-healing

When a task is scheduled on one of the slaves, that slave receives a description of the task, which contains all the information needed to start it: the download addresses for the task to be executed and all the input files needed. All the files are downloaded locally on the slave and the computation is started. When the task is finished, the results are uploaded, every temporary files are removed and the master is notified about the end of the computation.

There are however certain situations when the execution of the task is interrupted, and which requires some kind self-healing mechanisms. One such situation is when the task started by the slave is crashing, due to a software problem in the executed program. The slave will detect such failure, the it will clean up any local temporary files, and notify the master about this. The master can either notify the user about the situation, or try to execute the task on a different platform slave, if possible.

Another situation is when a task is running and a user is starting an interactive session on that slave computer. Since interactive users have priority over any executing tasks, the running task will be interrupted. The task can be migrated to a different slave, or restarted, if migration is not possible, on a different slave. Migration can be done if the task program can provide the means to save the current state of the program and continue the execution from this point on a different computer. This however has to be done inside each task. A future extension we are investigating now is to use checkpointing techniques. When the task needs to be interrupted, it is first checkpointed, the the resulting memory footprint is transfered on the new slave computer, where the computation is resumed.

Another self-healing situation is necessary when a task is running for too long. In this case, the local slave will stop the execution and notify this to the master.

The current implementation of the system is made considering only one central master node. This can be an inconvenience in certain situations, where computers located in different networks are used together. The master node can also be subject to failures, software or hardware. A more decentralized approach is needed in this case. Currently, our high level communication protocol between the

entities, especially between the client and master, allows a master to act as a client to another master, thus making possible to create a *distributed master*, consisting of independent master nodes which communicate between each other. Ideas from peer-to-peer (P2P) computing will be used for implementing such a decentralized approach.

5 Conclusion and Future Work

In this paper we presented an overview of the architecture of the QADPZ distributed computing system, and its autonomic features that simplify the management of the system.

The use of already existing hardware resources, like for example desktop PCs, in a distributed computing environment has a tremendous potential of providing a computing platform on which different computationally demanding applications can be executed in a time comparable with supercomputers and dedicated clusters. This approach is also a very affordable alternative, as investment costs and later maintenance are minimal compared to other systems.

The QADPZ system is currently used for different kind of computational projects and is installed in our department on a cluster of 40 PCs running Linux, and on 80 desktop computers in a lab running Windows and FreeBSD. Problems from the areas of large numerical simulations (parallel computational fluid dynamics), evolutionary algorithms, image processing and scientific visualizations are successfully solved on our installations.

Our future work will focus on studying the self optimization of the system, by monitoring and evaluating the performance of the different components. This is an important issue, especially when running parallel applications. Dynamic balancing of the workload can be used.

Other possible extensions of the system are currently considered, for example interconnection with a grid computing environment and a more decentralized approach for the master entity. Security is also an important issue, further work being needed to improve the existing security features.

References

[1] QADPZ web site. `http://qadpz.sourceforge.net/`.

[2] R. Buyya. *High Performance Cluster Computing: Programming and Applications*. Prentice Hall, 1999.

[3] Z. Constantinescu, P. Petrovic, and A. Pedersen. Q^2ADPZ * An Open System for Distributed Computing. In *NordU2002 Conference, Helsinki, Finland*, 2002.

[4] V. K. Garg. *Elements of Distributed Computing*. John Wiley & Sons, 2002.

[5] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM Corporation. `http://www.research.ibm.com/autonomic`, Oct. 2001.

[6] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, January 2003.