ENGINEERING JOURNAL

Article

PicoGrid: A Web-Based Distributed Computing Framework for Heterogeneous Networks Using Java

Nipun Wittayasooporn, Apikrit Panichevaluk, and Yan Zhao*

International School of Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand
*E-mail: yan.z@chula.ac.th

Abstract. We propose a framework for distributed computing applications in heterogeneous networks. The system is simple to deploy and can run on any operating systems that support the Java Virtual Machine. Using our developed system, idle computing power in an organization can be harvested for performing computing tasks. Agent computers can enter and leave the computation at any time which makes our system very flexible and easily scalable. Our system also does not affect the normal use of client machines to guarantee satisfactory user experience. System tests show that the system has comparable performance to the theoretical case and the computation time is significantly reduced by utilizing multiple computers on the network.

Keywords: Distributed computing, heterogeneous networks, Java.

ENGINEERING JOURNAL Volume 19 Issue 1
Received 24 April 2014
Accepted 7 October 2014
Published 30 January 2015
Online at http://www.engj.org/
DOI:10.4186/ej.2015.19.1.153

DOI:10.4186/ej.2015.19.1.153

1. Introduction

Organizations around the world invest millions of dollars on supercomputers and mainframes to process financial and scientific calculations that require an enormous amount of computing power. Nonetheless, there is already a huge amount of computing resources available from idling computers in offices and computer labs, which are ready to be cultivated. This work aims to utilize the abundant computing power that lies idle in an organization's local area network (LAN) and put it into better use.

This framework allows authorized users to easily write and deploy distributed computing tasks on a network of computers that are registered as agents on the system's central server. Users can later retrieve results once the task is completed. The computing service can be accessed by users through a web interface via any standard and compliant web browsers. The implementation of the framework is based on Java platform utilizing its portability, its security feature via Java security manager, and benefits from its globally large installation base.

There are many distributed computing frameworks available in the market. However, most of them are very complex or do not provide a way for users to easily deploy and execute distributed programs by themselves. Moreover, only a few of them have the capability of utilizing idle computing resources. This work aims to bring together all the features offered by existing systems to achieve our objectives.

The rest of the paper is organized as follows: Section 2 discusses existing platforms or frameworks for distributed computing; Section 3 introduces objectives of the framework; detailed framework design is explained in Section 4; a practical implementation of the framework is introduced in Section 5; a benchmarking and performance test of the implemented framework is included in Section 6; finally, Section 7 draws a conclusion and discusses the future work.

2. Related Work

This section discusses related platforms or frameworks for distributed computing, and explains major differences between our proposed framework and existing systems.

JavaParty is an extension of Java designed to support transparent remote objects [1]. It introduces a new keyword, 'remote', which defines that the class should be executed remotely. JavaParty uses a preprocessor, which converts remote classes into pure Java code utilizing the *Remote Method Invocation* (RMI). This greatly simplifies the programming task; however, modifying the Java language comes at an expense of Java compatibility. Developers must compile classes using the provided JavaParty compiler. Overall, the focus of JavaParty is on making distributed RMI Java programming simpler, at the cost of source code portability. Our developed program does not require any Java modification, and the use of standard Java makes our program very simple to deploy.

Java/DSM uses a modified *Java Virtual Machine* (JVM) to implement a *Distributed Shared Memory* (DSM) system across networks of workstations [2, 3]. In order for Java/DSM applications to work correctly, it is implemented on top of the TreadMarks DSM implementation to assist the underlying system in keeping memory consistent across machines. TreadMarks is a DSM system which is heavily dependent on the operating system and architectural support. Thus, the modified JVM used for Java/DSM is also limited to the architectures supported by the TreadMarks system. Our program uses standard Java with no modification required, making it very portable.

Javelin brings together three types of entities: clients, brokers and hosts [4]. Clients, who seek computing resources, register with a broker and submit their work in the form of an applet. Hosts, who are willing to donate resources, contact the broker and run the applets. This work emphasizes methods for bringing together clients and hosts, and focuses on ways of bartering for CPU time. Support is concentrated on programs which can be divided into and run in Java applet form. Due to the reliance upon applets, Javelin does not support object interactivity in the form of remote invocations, or advanced features such as migration. Our developed program does not require applet, which enables us to monitor the system state.

Charlotte attempts to harness the power of idle computers on the Internet using Java's applet technology [5]. The goal of Charlotte is to support distributed and shared memory on top of the JVM. Charlotte provides classes that support *Distributed Shared Memory* (DSM) semantics on top of the standard JVM. Charlotte uses an interesting distributed programming approach that makes use of existing Internet infrastructure such as HTTP servers and web browsers running applets to accomplish its goals. However, Charlotte programs must conform to the parallel routine program structure and must be implemented as a Java applet. This approach is not very transparent or flexible because programmers must adhere to both the applet API and Charlotte's parallel routine structure. Charlotte requires keeping shared memory consistent across nodes. Our program avoids these complexities and does not require applet which makes it very simple. Shared memory is used only on the server, but not the clients.

Ninflet is a Java framework that harnesses unused computational power of idle computers on the Internet [6]. It was built to support object-based programming and takes advantage of many of the existing features of the Java language such as security, remote method invocation and object serialization. Ninflet servers host objects for clients and must register themselves with a scheduler. The scheduler manages servers and provides them to clients for hosting their objects. Our system is simpler than Ninflet, and it does not have scheduler but only contains server and clients.

ObjectSpace's Voyager is a comprehensive commercial Java based distributed computing environment [7]. Voyager's distributed systems toolkit supports remote object creation, asynchronous remote method invocation and migration as well as substantial features beyond that Babylon supports such as multi-casting, name spaces, servlets, timers and CORBA support. Voyager is based on the *Object Request Broker* (ORB) architecture but does not limit itself to a specific remote messaging standard. It supports not only CORBA but also RMI, the *Distributed Component Object Model* (DCOM) and the *Simple Object Access Protocol* (SOAP). As a result, Voyager can act as the 'glue' between varieties of distributed applications using any appropriate underlying remote messaging standard. However, Voyager does not provide how their `distributed glue' is constructed in detail, thus not allowing any implementation comparison with our framework.

ProActive is a Java library that provides tools and services for parallel and distributed application development [8]. ProActive does not require JVM changes or external tools. ProActive generates proxies for remote objects at runtime using bytecode engineering libraries as it predates dynamic proxies in Java. This capability is used to create new remote objects and export local ones. ProActive can execute remote methods synchronously or asynchronously. Our program does not generate bytecode or allow objects to be created remotely, but simply uses distributed JAR instead.

BOINC, short for Berkeley Open Infrastructure for Network Computing, is an open source software for volunteer and grid computing [9]. It is used by science research projects such as SETI@Home, Rosetta@home, World Community Grid, Climateprediction.net and many other Universities and Organizations around the world. Volunteers can download the BOINC software, attach to one or more of these science projects and participate in science research. While BOINC uses voluntary models, our framework tends to be invisible to users. Our system works in a single organization network with no external connection from the Internet.

3. Objective of the Framework

The objective of this work is to design and implement a distributed computing framework that allows users to execute distributed computing tasks and retrieve results via a web interface. The user friendliness is the topmost priority. The framework should be easy to be familiarized by anyone with little programming experience in distributed computing. Its architecture should allow many problems to be solved using parallel algorithms with a shared-memory model. Calculation performance can be a tradeoff in favor of user friendliness and flexibility when a design decision needs to be made. Detailed objectives of the framework are listed as follows:

- The framework should allow users to manage their distributed computing tasks via a web interface. Main functionalities include uploading new tasks, canceling running tasks, and downloading results for finished tasks.
- The framework should satisfy appropriate security requirements. It should allow the work to be done transparently without compromising both the work itself and the client data.
- The system should be fault-tolerant to failing agent nodes. It should be designed for high-peer churning environment in mind.
- The agent software should be transparent to the active user of the agent machine and limit the impact to the user experience as much as possible.

The main purpose of this work is to minimize the administrative overhead on all users including administrators. The platform deployment should be very simple, and the task can be automatically distributed. The scope of the work presented in this paper includes:

- A very simple user management system is provided for the purpose of system test only. No administrative software is provided, which is expected to be integrated into an existing user management system of an organization by administrator/programmer.
- The security measure taken in the design and implementation is not tested. The security is proven in the design document only in theory.
- The security of the distributed program itself is not guaranteed, as the attempt to peek and modify memory is very easy on agent computers.

The system setup of the proposed framework is shown in Fig. 1.

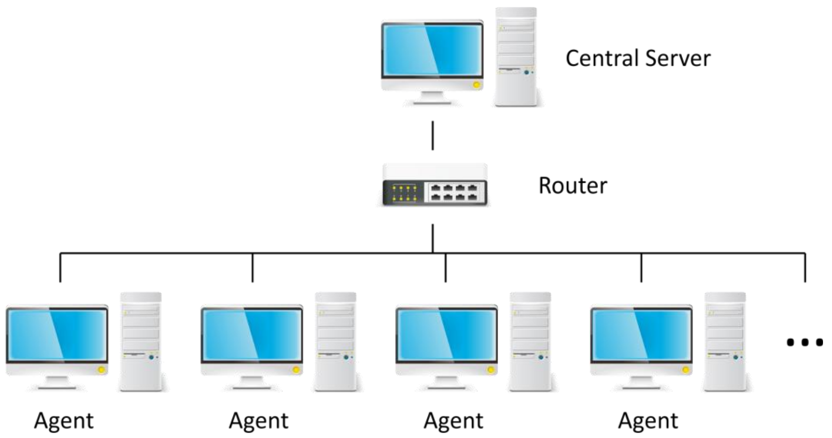


Fig. 1. The network configuration of the proposed framework.

4. Framework Architecture

In this section, the design of the framework system is described, including Terminology, Framework Overview, Scenario, Central Application, Agent Application, Communication, and Security.

4.1. Terminology

- *Framework* is the base system that makes distributed computing possible. The framework includes both deployment and execution processes. All distributed applications for use in the system are written using the API provided by the framework. The control flows and scheduling are all handled by the framework.
- *Framework Services* are the applications that manage all the processes in the framework, from scheduling tasks to managing all the data within the system. Services in the framework system include Central for the server and Agent for the clients. The administrators are responsible for installing and managing the services.
- *Distributed Application* is an application written for accomplishing jobs by dividing them into smaller tasks and work simultaneously in a distributed manner. All distributed applications for use in the system are written using the API provided by the framework, and any other methods of communication that are not allowed by the security manager may not be used.
- *Job* is referring to a set of tasks that work together in order to solve a problem. To accomplish a job in this framework, a distributed application must be written to create tasks and solve the problem using parallel or distributed algorithms.
- *Task* is referring to the smallest unit of work to be distributed and executed on the framework.
- *Administrator* is the person with the privilege and responsibility to manage the framework services.
- *User* is the person who may run jobs using the system. Distributed applications are written by the users and then submitted to the system. The administrators are responsible for managing the user access.

4.2. Framework Overview

The architecture of the proposed framework is shown in Fig. 2.

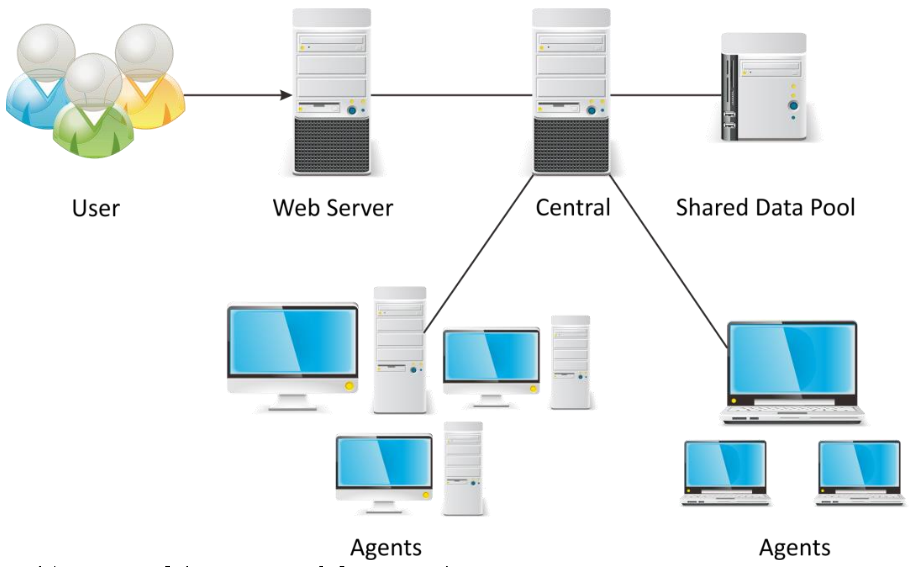


Fig. 2. The architecture of the proposed framework.

The operation of the system is explained as follows:

- Users interact with the distributed computing services through a website.
- The web server sends a job to the Central and the Central sends back results once the job is accomplished.
- The Central distributes tasks to the agents.
- All communications between the agents are achieved through the Central using a shared data pool (which is exposed through the framework API).
- No synchronization is available, though it is possible to do a simple joint synchronization through task queuing.

4.3. Scenario

A research organization would like to test the performance of their newest algorithm for breaking *Advanced Encryption Standard* (AES) encryption but does not want to invest on more machines. The organization can utilize this framework using existing machines. The scenario is illustrated in Fig. 3.

The complete process of calculation is detailed as follows:

1. The User writes distributed applications using framework API that will break AES encryption using certain arguments as input.
2. The Administrator starts the Central on the machine that serves as a server.
3. The Administrator configures the Agent to point to the Central's address.
4. The Administrator starts the Agent on any PCs within the network (Agents can enter and leave the network at any time from this point).
5. The User uploads the distributed application via the Central hosted website.
6. The Central loads the job and insert an initialization task into the queue.
7. An Agent receives the initialization task and runs the initialization (setting up shared variables, queuing more tasks etc.).
8. Each Agent receives a task from the Central.
9. The Agent loads a distributed application associated with the task from the Central (by either downloading a new one or using a cached one on the client machine).
10. The Agent executes the distributed application with received task as an argument.
11. The Agent uses the framework provided API for retrieving and submitting data. The shared data pool will not be updated until the Agent finishes the task.

12. The Agent sends 'done' message to the Central so that it can commit the changes made during the previous step to the shared data pool.
13. Once the Agents have done all the tasks, one of them will send a pack message to the Central which will pack the result into a text file.
14. The User then downloads the result from the website.

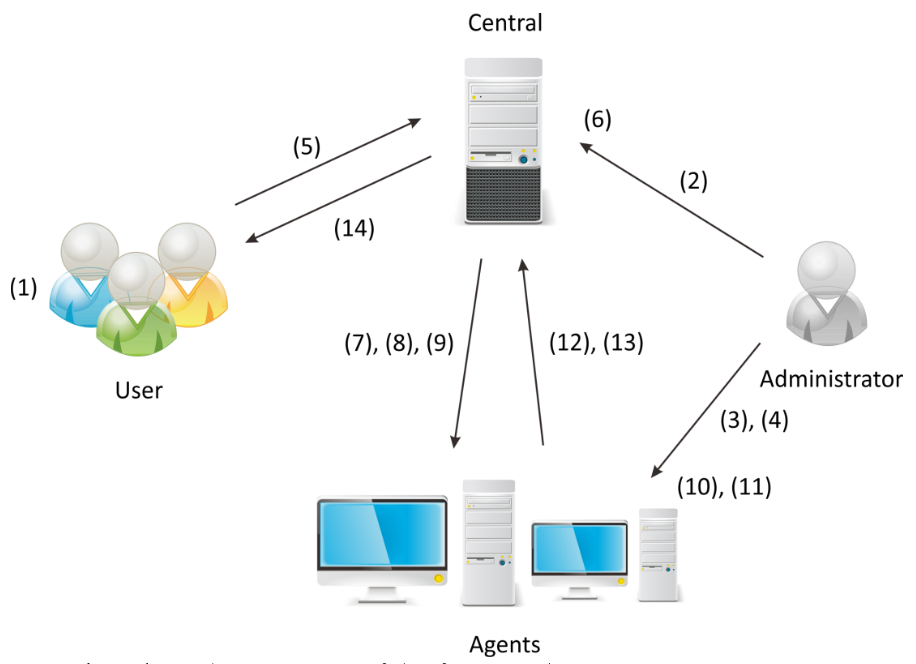


Fig. 3. A scenario describing the operation of the framework system.

4.4. Central Application

Central is the core of the framework. Its main responsibility is to manage task distribution and the shared data pool. We propose a simple task distribution model with the intention of making it less error-prone, more tolerant to failures and also simpler to implement:

- One task queue for each job. The task queue will maintain phase variable, waiting task lists and working task lists. As illustrated in Fig. 4, when Init 1 is being executed, all tasks inside it must be completed before Init 2 is initiated.
- The task queue starts off empty once a job is added to the Central.
- The Agent queries the Central for a task when it is available to work. The Central iterates through task queues for each job in a first-in-first-out manner.
  - If the waiting task list in the task queue is not empty, then it will remove a task in a first-in-first-out manner and send it back to the Agent.
  - If the waiting task list in the task queue is empty but the working task list is not, then the Central waits until a new task is available in any queue.
  - If both the waiting task list and the working task list in the task queue are empty, then the Central must add an initialization task to the queue with the value of phase variable as an argument. After that, the value of phase variable is increased by one.

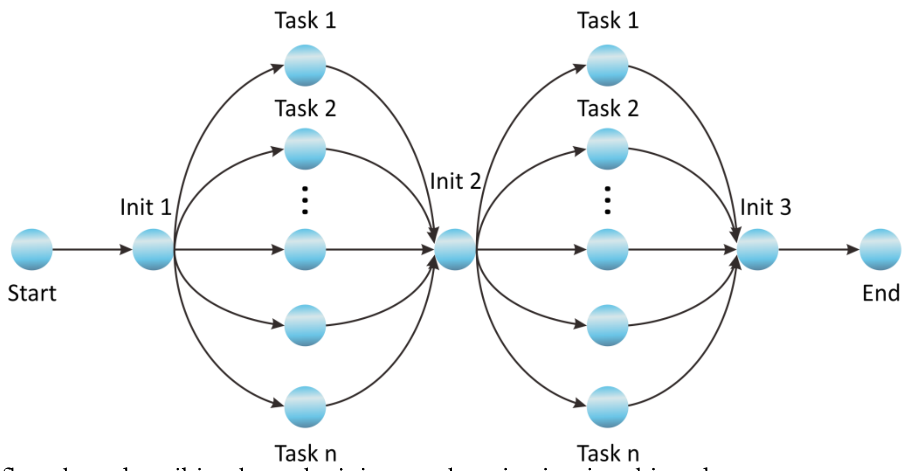


Fig. 4. The flowchart describing how the joint synchronization is achieved.

The reason behind the phase variable is to allow a simple synchronization mechanism. Each initialization task serves as a joint synchronization point where all tasks must be done before a new initialization task is started. The name is chosen to be initialization task because its purpose is only to setup concurrent tasks for the next phase. The last initialization task should create a task to send a pack message to the Central to indicate the end of the execution process of the job.

The working task list is required to keep track of the tasks that have been given out to the Agents but have not yet completed. In case if an Agent goes offline or it decides to hand back the task (machine being used heavily), the Central can recover the task from the list and put back into the waiting task list.

4.5. Agent Application

Agent is the workforce of the framework. Its main responsibility is to execute tasks in a secure and discreet manner.

Since each task should be designed to work in a single thread environment, like one would write for green/lightweight thread or fiber, the Agent will fork a worker thread for each computation unit (logical core) on the machine. Each worker thread queries the server for tasks and executes them according to the task received. The task indicates the name of the job which in turn indicates the name and location of the executable.

The Agent is required to be discreet, that is not to disturb the user of the machine. The user should be able to use non-heavy computation applications such as the Word processor, web browser, or media player without noticing that the Agent is running. A few good strategies can be implemented as follows:

- Running tasks during the time that the screensaver is active. BOINC framework uses this method. Although it is a good idea, idle CPU cycles during the screensaver being inactive are wasted, which is against our objective. Moreover, it is also operating system-specific unlike the Java AWT/Swing GUI library.
- Stopping tasks when the user is using keyboard or mouse. Though it sounds like a possible solution, in fact it is not as practical as one may think. Many user tasks do not require keyboard/mouse inputs, such as watching movies. This strategy could interfere with video rendering and make the system less responsive.
- Monitoring other processes' CPU usage and dynamically adapting to it. This is one of the good strategies but hard to implement correctly and efficiently. The solution is not cross-platform but a lot of operating systems support, as monitoring CPU usage is a quite common operation. The distributed application must make a call to check CPU usage every once in a while, especially in high CPU usage tight loops. The method of lowering the CPU usage is similar to limiting the frame rate of video rendering applications.
- Setting the worker threads to low/background priority. This is the easiest method and provides good enough results. By setting them to a lower priority, the operating system will automatically lower the threads' CPU usage by giving more CPU time to higher priority processes.

4.6. Communication

In this framework design, all communications are accomplished through the shared data pool handled by the Central. By using centralized memory, the data will not be lost in the event of down agent nodes or disconnections. The communication speed may be lower than that of the direct message passing between two agent nodes, but the latter cannot be implemented because we cannot guarantee the availability of any nodes but the Central. The distributed memory model is impractical if we consider the Agent nodes as volatile. Thus this configuration also requires the Central machine to be able to store a large amount of memory, either in the main or secondary memory of the machine.

All the access to the shared data pool is in the form of transactions. This is to prevent downed Agents from damaging the data when it is unable to completely transfer the result data back to the Central. Furthermore, it allows redundancy checks as described in the Security section.

4.7. Security

This framework addresses several security issues that come with running potentially dangerous code on other people's machines as well as the security of the result data coming from unknown contributors.

The distributed applications are user-created applications which can contain any code including harmful ones. However, Java has built-in Security Manager which controls the calls to all potentially harmful system calls. It disallows all calls to file operations, socket operations, clipboard accesses, creating processes, and initiating printing jobs, etc. The Agent handles the Security Manager and puts the worker threads to run under it before executing distributed application codes. Though it is possible for one to digitally sign the distributed application and ask JVM for higher privileges, the machine user must explicitly allow them. It should be noted that Java Security Manager cannot prevent unproductive CPU usage (tight loops of doing nothing).

The results coming from anonymous contributors are also potentially dangerous since they can make all the works in vain. Several security measures must be implemented to prevent them. Firstly, all communications should be encrypted using some kind of encryption algorithms such as public-private key encryption through SSL/TLS. The key exchange can be done prior to the actual execution; it could be exchanged during the installation/deployment phase. This can prevent the middle-man interference. Secondly, the Central can create duplicate tasks and send them to more than one Agent as a redundancy check. After the tasks are distributed and done, the central will check for the result transactions coming back from the Agents. The Central should pick the result with highest redundancy. It could also put a lower limit on the percentage of the required amount of results that should be the same. However, the redundancy check will slow down the entire process by the amount of multiples the task is duplicated.

The best security that the framework could achieve is not in the software. The administrators should make sure that all distributed applications are safe to run. They should allow only trusted and identifiable people to be able to upload tasks onto the system. The Agents should only be installed on the trusted machines, such as the employees' machines, lab machines, or old machines that are not being used.

5. Framework Implementation

5.1. Scope of the Implementation

The implementation of the framework is a subset of the design given in the previous section. The objective of the implementation is to show that the full implementation of framework is possible, to run tests on the agent CPU throttling strategies, and run benchmarks to test the scalability. The core architecture stays the same, as the steps in the scenario case still remain valid. The security measures described in the Security section is not implemented and the communication is not in transaction. But if the agent only gives back a single result per task, the framework will behave as if it were a transaction, putting this restriction onto the distributed application programmer instead. Also, the Central only has a single task queue, so it could only execute one job at a time, although the design already explains the algorithm for running multiple jobs concurrently in a clear and precise manner. For the sake of easily referring to the implementation, we have conceived a name PicoGrid for our implementation.

5.2. Protocols

In this implementation, all inter-process communications are character-based via TCP/IP sockets. The protocol is described as follows:

- QUERY -- This command is initiated by the Agent to ask the Central for a new task. The Central will send back the task to the Agent using this character-based format: <TASK> <TASK ARGUMENTS>. The Central always gives back a task, or wait indefinitely until new tasks come into the queue.
- PUSH <TASK> <TASK ARGUMENTS> -- This command is initiated by the Agent to push new tasks into the task queue. This command will always succeed. Note that the actual design should implement this as a transaction as well.
- DONE -- This command is initiated by the Agent to signal the Central that it has completed the task and the Central should commit the changes into the shared data pool. This command will always succeed.
- GET <KEY> -- This command is initiated by the Agent to ask the Central for data associated with the KEY in the shared data pool. The Central will respond OK <VALUE> if the data is found within the shared data pool. Otherwise, the Central should respond NA (not available).
- PUT <KEY> <VALUE> -- This command is initiated by the Agent to put new data or modify existing ones in the shared data pool. This command will always succeed.
- PACK -- This command is initiated by the Agent to signal the end of the job. The Central will write the data in PICOGRID_RESULT key-value pair to the shared data pool.

The delimiter between keyword/value is a tab character. The delimiter for each command is a newline character.

5.3. Agent CPU Throttling Strategies

We have investigated the following three strategies: setting normal priority processes, CPU monitoring then adaptively throttle CPU usage, and setting lower priority processes. The screensaver and input monitoring strategies are ruled out because they do not fit to our objectives. The tests have been conducted by collecting comments from six people using the computer while the Agents are constantly running tasks. The comments are split into 'slowdown noticed' and 'slowdown unnoticed', as listed in Table 1.

Table 1. Performance evaluation results of the implemented framework.

	Slowdown noticed	**Slowdown unnoticed**
Normal process priority	2	4
Adaptive CPU throttling	3	3
Lower process priority	1	5

From the result data, we believe that the implementation of adaptive CPU throttling is unacceptable. But it is also possible that CPU monitoring costs extra CPU cycles which may make the strategy worse than doing nothing. But we also found that most people do not notice any slowdown even though we did not do anything to throttle down the CPU. This may be because the operating system is able to identify high CPU usage processes and put them into background as no feedback is output to the user. On the other hand, setting lower process priority produces satisfactory results.

6. Benchmark

6.1. Calculation of Euler's Number

We choose the calculation of Euler's number (*e*) as it is simple to create a concurrent algorithm and we can focus more on the benchmark result rather than fixing bugs. The definition of Euler's number in the form of infinite series is given by:

$$e = 1 + \frac{1}{1} + \frac{1}{1\times 2} + \frac{1}{1\times 2\times 3} + \frac{1}{1\times 2\times 3\times 4} + \dots \tag{1}$$

In our test, we calculate the Euler's number up to 10,000 terms at the precision of 10,000 digits. An initialization task is responsible for queuing these terms to the task queue, each Agent calculates a given term and sends the result back to the Central. After all terms are computed, an Agent will take responsibility to compute the sum and send the result back to the server. While we can split the summing operations into smaller tasks, we found that one thread could calculate it faster due to the large amount of data required to be sent back and forth between each Agent and the Central.

Agents can accept concurrent operations up to the machine's logical cores. For example, an agent machine with Intel quad-core processor with hyper-threading enabled can run up to eight concurrent tasks.

We divide the time measurement into two parts, the scalable time which is the time spent on computing the terms, and the total time when the complete result is sent back to the server. We use these two types of data to compute the summation time which is the time spent on summing the terms that are conducted on a single thread. Thus

$$\text{Summation Time} = \text{Total Time} - \text{Scalable Time} \tag{2}$$

The same test is conducted for five times and the average computation time is calculated. Then we calculate the number of tasks per second for evaluation.

6.2. System Setup

The tests are conducted in a heterogeneous environment involving: Intel Core 2 Duo laptops, Intel Core i7 Laptop and AMD Athlon PCs, having dual-core and quad-core processors, running Apple Mac OSX and Microsoft Windows connected by a gigabit switch, as we prefer faster connections to minimize the I/O overhead. The configuration of the network is shown in Fig. 5.

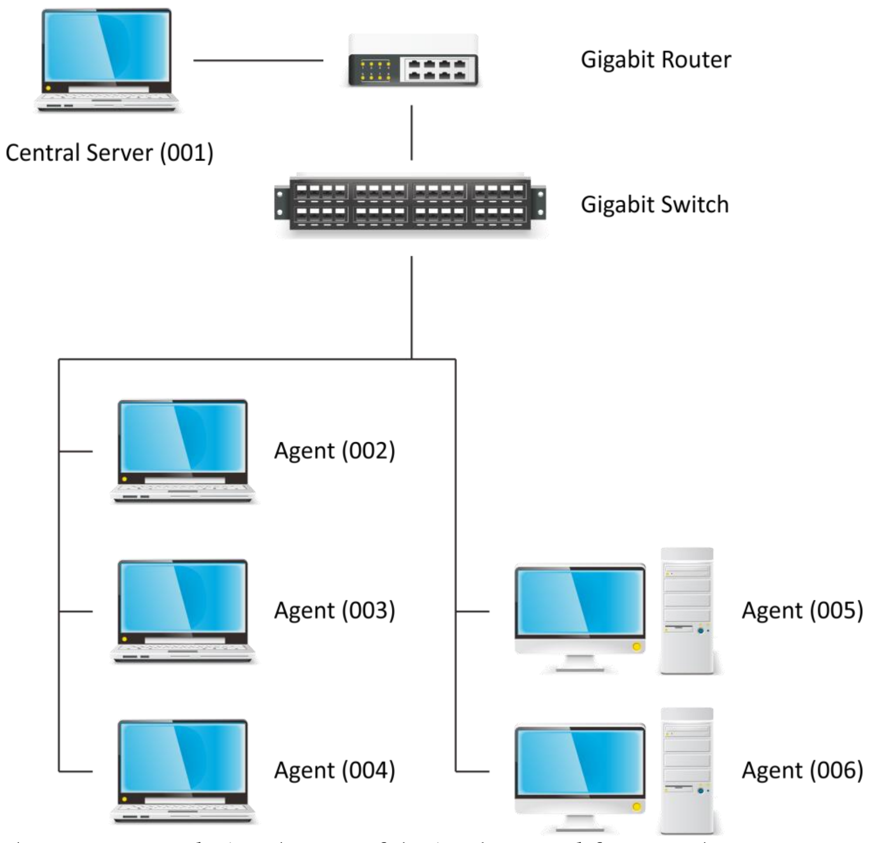


Fig. 5. Network system setup during the test of the implemented framework.

All machines are connected to the gigabit switch which is routed by a gigabit router. The purpose of the switch is to extend the number of Ethernet ports in the network.

The Central Server (001) is an Intel Core 2 Duo laptop running Windows 7, with the Central application installed. Machines running Agent applications include Agent (002), an Intel Core 2 Duo laptop running Mac OS X Lion, Agent (003), an Intel i7 quad-core laptop running Mac OS X Lion, Agent (002), an Intel Core 2 Duo laptop on Mac OS X Snow Leopard, Agent (005) and Agent (006), both AMD Athlon desktops running Windows XP.

The web interface of the implemented framework is shown in Fig. 6, where users can upload distributed application files to run, as well as retrieve results once the job is completed.

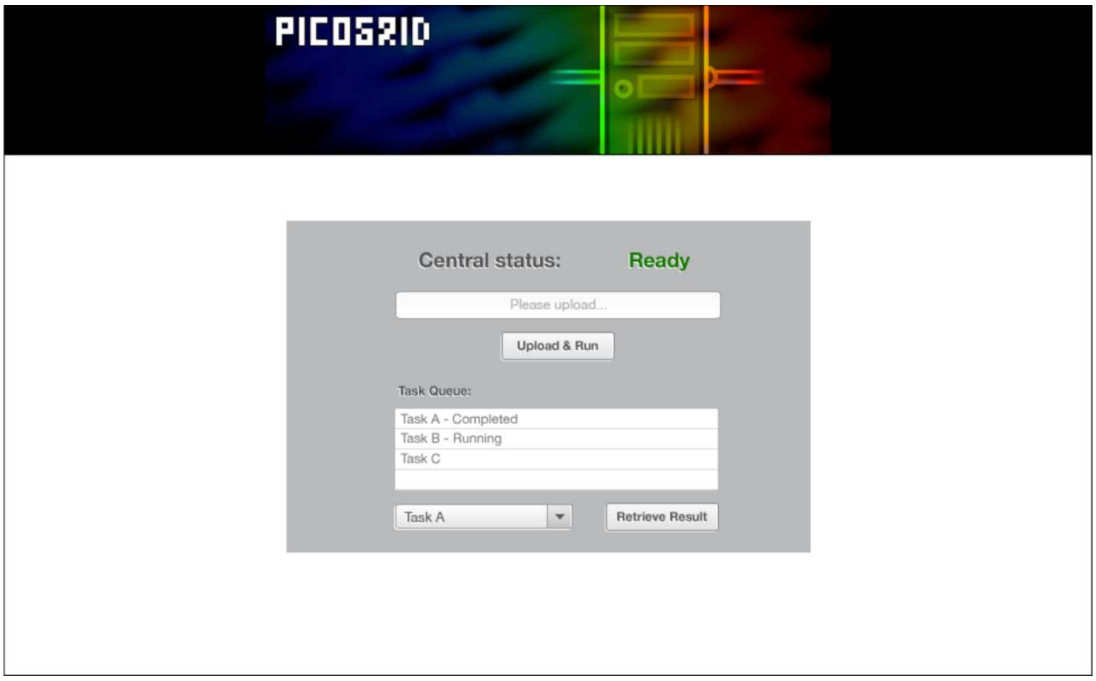
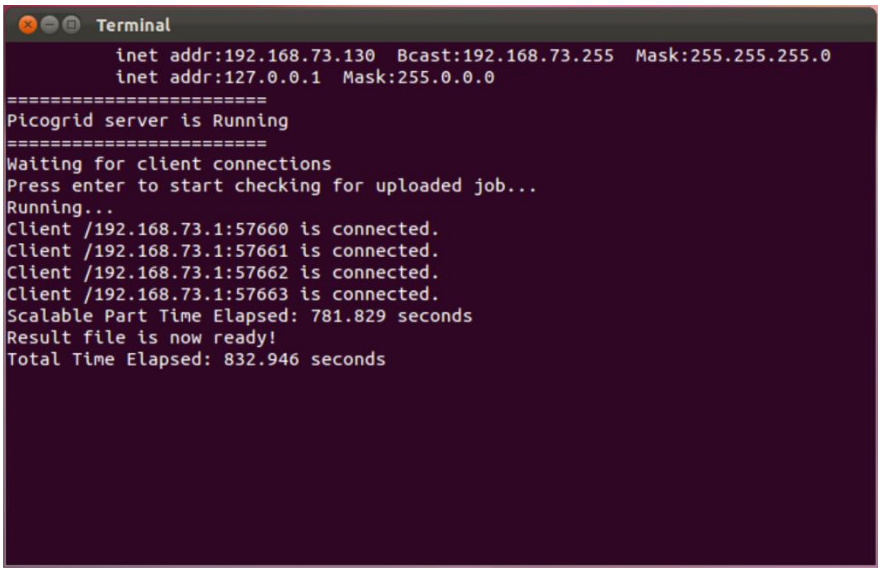


Fig. 6. Screenshot of the web interface of the implemented framework.

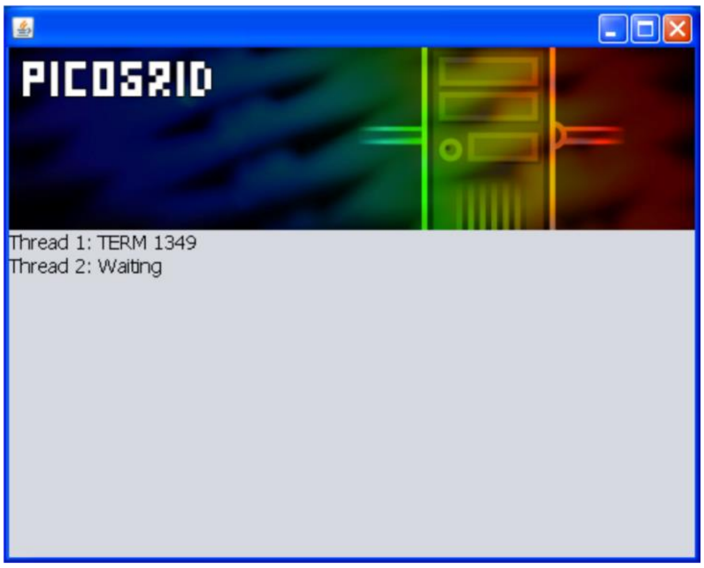
Figure 7 shows the complete process when the implemented program is running on the server.



```
          inet addr:192.168.73.130  Bcast:192.168.73.255  Mask:255.255.255.0
          inet addr:127.0.0.1  Mask:255.0.0.0
========================
Picogrid server is Running
========================
Waiting for client connections
Press enter to start checking for uploaded job...
Running...
Client /192.168.73.1:57660 is connected.
Client /192.168.73.1:57661 is connected.
Client /192.168.73.1:57662 is connected.
Client /192.168.73.1:57663 is connected.
Scalable Part Time Elapsed: 781.829 seconds
Result file is now ready!
Total Time Elapsed: 832.946 seconds
```

Fig. 7. Screenshot of the server application running on the Central during a system test.

In this particular case, four clients are connected to the server to perform calculations. The total computation time is also returned at the end of the running program. Figure 8 shows the message window when one of the threads is dedicated to running calculations on an Agent machine during the system test.



Thread 1: TERM 1349
Thread 2: Waiting

Fig. 8. Screenshot of the client application running on one Agent machine during a system test. Note that this window will not be displayed during the real implementation of the program.

However, in the real implementation of the program, the calculation is performed at background and this window will not be displayed to the machine user.

For the system test, the above computation is repeated on various numbers of machines/threads and the calculated speed-up factor is shown in Fig. 9, where the speed-up factor is defined as the computation time for a job on a single thread, divided by the computation time for the same job on multiple threads.

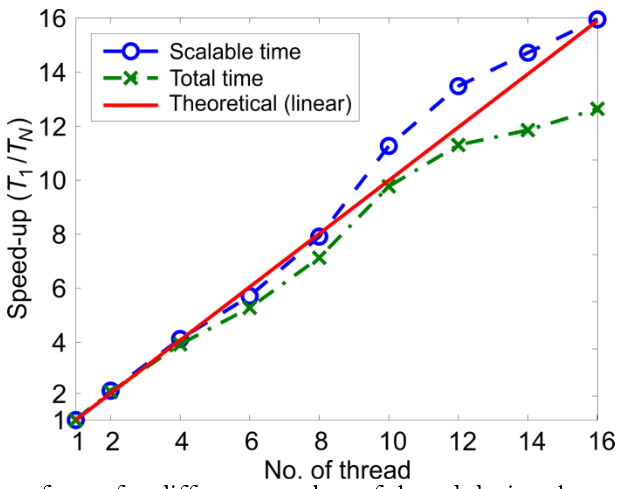


Fig. 9. Calculated speed-up factor for different number of thread during the system test.

The theoretical linear speed-up factor is also shown in Fig. 9 by the red straight line. From the results it can be seen that the scalable time can exceed the theoretical case in some system configurations. This is because the theoretical speed-up factor is calculated based on the assumption that all threads have the same computational speed. While in our case, some Agent machines with higher specifications join the computation later than those with lower specifications during the test. Nonetheless, the total time is always below the theoretical case.

7. Conclusion

In this paper, we propose a framework for distributed computing applications in heterogeneous networks. The system is simple to deploy and can run on any operating systems that support the Java Virtual Machine. Using our developed system, idle computing power in an organization can be harvested for performing computing tasks. Agent computers can enter and leave the computation at any time which makes our system very flexible and easily scalable. Our system also does not affect the normal use of client machines to guarantee satisfactory user experience. System tests show that the system has comparable performance to the theoretical case and the computation time is significantly reduced by utilizing multiple computers on the network.

As the system has been developed and tested in a small network, further investigations may attempt to scale up the system to the size of the entire institution and investigate whether a higher performance could be achieved in a larger environment and whether the system can be implemented for practical use.

References

[1] M. Phillippsen and M. Zenger, "JavaParty - Transparent Remote Objects in Java," ACM 1997 Workshop on Java for Science and Engineering Computation, 1997.

[2] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, and C. Amza, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18-28, 1996.

[3] Alan Cox, and Weimin Yu, "Java/DSM: A platform for heterogeneous computing," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1213-1224, 1997.

[4] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu, "Javelin: Internet-based parallel computing using Java," *Future Generation Computer Systems - Special issue on metacomputing*, vol. 15, pp. 5-6, 1999.

[5] M. Karaul, Z. Kedem, and P. Wyckoff A. Baratloo, "Charlotte: Metacomputing on the web," Proceeding of the 9th Conference on Parallel and Distributed Computing Systems, 1996.

[6] S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, U. Nagashima, and H. Takagi, "Ninflet: a migratable parallel objects framework using Java," ACM 1998 Workshop on Java for High-Performance Network Computing, pp. 151-159, 1998.

[7] Inc. Recursion Software, "Voyager ORB developer's guide," version. 4.6, 2003. Online at: http://www.recursionsw.com/products/voyager/voyager.asp.

[8] D. Caromel, R. Guider, and I. Attali, "A step toward automatic distribution of Java programs," Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 141-161, 2000.

[9] D. P. Anderson, "BOINC: A system for public-resource computing and storage," 5th IEEE/ACM International Workshop on Grid Computing, 2004.