國立交通大學

資訊科學系

碩 士 論 文

一個非同步低耦合度之動態格網運算系統

An Asynchronous Decoupled Dynamic Grid Computation System

研 究 生：沈上謙

指導教授：袁賢銘　教授

中 華 民 國 九 十 四 年 六 月

一個非同步低耦合度之動態格網運算系統

An Asynchronous Decoupled Dynamic Grid Computation System

研 究 生：沈上謙　　　　Student：Shang-chien Shen

指導教授：袁賢銘　　　　Advisor：Shyan-ming Yuan

國 立 交 通 大 學

資 訊 科 學 系

碩 士 論 文

A Thesis
Submitted to Department of Computer and Information Science
College of Electrical Engineering and Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer and Information Science

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

一個非同步低耦合度之動態格網運算系統

研究生：沈上謙　　　　　指導教授：袁賢銘

國立交通大學電機資訊學院

資訊科學研究所

摘要

諸如基因排序以及蛋白質解析等複雜的解碼工作需要大量的運算量以及繁複眾多的執行步驟，業界多半利用格網運算(Grid Computing)的方式將繁重的運算工作交予後端的分散式叢集電腦群來快速完成，這一類的格網系統通常仰賴著昂貴的硬體設備或是特殊、特定的軟體，使得格網運算這個名詞數年以來一直狹隘地隸屬於高速科學計算的領域。縱使在科學領域已被廣泛使用，昂貴的整體擁有成本(Total Cost of Ownership)讓一般使用者或甚至中小型企業對於採用格網概念望之卻步。龐大的潛在電腦運算資源依然未能有效被利用。使用者持續尋找著更多的運算資源來解決他們的問題。在這些挑戰之上，格網本身尚暴露著技術層面的瑕疵。業界的方案無法有效解決諸如此類單一切入點故障(Single-point of Failure)的架構性問題、動態擴充格網體積的延展性問題(Scalability)、或是支援跨平台特性的普及性問題。

本篇論文提出一個低成本的純軟體跨平台格網方案，有效解決單一切入點故障、動態擴充格網體積等問題，並提供簡化格網應用程式開發的工具組以及程式設計介面(API)，大幅縮短研發人員粹取企業內部運算資源的時間，將生產力最佳化。研發人員將能專注於設計與開發，而不再需要週而復始地在企業中辛苦獵取閒置但是隱形的運算資源。

An Asynchronous Decoupled Dynamic Grid Computation System

Student：Shang-chien Shen *Advisor：Dr. Shyan-ming Yuan*

Department of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

Abstract

Complex jobs such as bio-genetic sequencing and protein modeling requires massive quantity of calculation and execution procedures. Today, industry applies Grid Computing technologies to delegate the intensive computational work to a farm of cluster computers in order to accelerate computing speed. This category of grid computing rely on sumptuous hardware or distinctive, specific software, thus restraining grid computing to constricted domains such as high-speed scientific computation. Despite the widespread acceptance of grid concept, high TCO(Total Cost of Ownership) intimidated the general public or even SMEs(Small-Medium Enterprises) from adopting grid technologies. Vast amount of potential computing capacity still remains untapped. Users are continually searching for more computing resources to assist solve problems. On top of these challenges, Grid itself suffers certain technical imperfections. Commercial solutions are incapable of solving single-point-of-failure issues, incapable of dynamically expanding the volume of grid network and is certainly having a difficult time migrating grid infrastructure to a universe of different electronic devices existing today.

This research proposes a low-cost, pure software-based, cross-platform grid framework, eliminating the mishap of single-point of failure, allowing dynamic grid expansion. The framework also provides utility tools and Application Programming Interfaces(APIs) that simplifies the process of grid application development, thus optimizes overall productivity. Developers must focus on design and development rather than hunting for resources hidden within the enterprise.

Acknowledgement

首先我要感謝在這兩年的歷程之中給予我珍貴的建議，並引導我朝正確研究方向邁進的指導教授 袁賢銘 教授，在指導過程之中給予我龐大的空間讓我自由揮灑創意。另外要感謝 蕭存喻、高子漢、葉秉哲 與 吳瑞祥 四位學長不時給我豐富的意見以及適時的鼓勵讓我可以順利地完成這篇論文。同時也謝謝分散式系統實驗室裡的成員 柯憲昌、葉倫武、顏志明，在研究上彼此互助交換意見，相互學習，一起成長。特別謝謝 憲昌在計畫中幫了我許許多多的忙，祝福他進入產業界後能順利將自己心中的理想與憧憬實現。

最後，謹將此篇小小的學術成就獻給我的父母親 沈立勝 與 王麗文 ，感謝你們給予我良好的環境讓我求學生涯無後顧之憂，專心於學術研究，同時也謝謝你們多年以來對我無條件地支持與栽培。

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

1. INTRODUCTION

1.1. PREFACE

Complex jobs such as bio-genetic sequencing and protein modeling requires massive quantity of calculation and execution procedures. Today, industry applies Grid Computing technologies to delegate the intensive computational work to a farm of cluster computers in order to accelerate computing speed. This category of grid computing rely on sumptuous hardware or distinctive, specific software, thus restraining grid computing to constricted domains such as high-speed scientific computation.

1.2. MOTIVATION

Despite the widespread acceptance of grid concept, high TCO (Total Cost of Ownership) intimidated the general public or even SMEs(Small-Medium Enterprises) from adopting grid technologies. Vast amount of potential computing capacity still remains untapped. Table 1-1[1] designates a research adapted from Internet Infrastructure & Services by Bear, Stearns & Co., quantifying the total idle computational resources, into a more tangible measurement. The result is astonishing.

	$/processor	$/used	$/used processor	Cost of unused cycles
1 desktop	$1,200	$300	$150	$1,050
1000 desktops	$1,200,000	$300,000	$150,000	**$1,050,000**

Table 1-1: Cost of unused computational resources [1]

According to the research, an enterprise with one thousand computers wastes minimum of $1.05 million worth of computational resources daily.

On top of things, Grid itself still suffers certain technical imperfections. Commercial solutions such as the SUN Grid [2] are incapable of solving single-point-of-failure issues, incapable of dynamically expanding the volume of grid network and is certainly having a difficult time migrating grid infrastructure to a universe of different electronic devices existing today.

In addition, the steep learning curve substantially increases the cost and risk of developing a stable real-world grid application. We must keep in mind that grid users are seldom experts in distributed technology, the significance of their innovation in developing applications often exceeds their knowledge towards grids. Thus, middleware providers need to provide exception-free, thread-safe and simple-to-use tools and APIs for grid application developers.

This research proposes a low-cost, pure software-based, cross-platform grid framework, named mGrid. mGrid eliminates the mishap of single-point of failure and permits dynamic grid expansion. The framework also provides utility tools and Application Programming Interfaces(APIs) that simplifies the process of grid application development, thus optimizes overall productivity. Developers must focus on design and development rather than hunting for resources hidden within the enterprise.

1.3. RESEARCH OBJECTIVES

mGrid framework is intended to achieve six major objectives: low-cost, cross-platform, high performance resource sharing, high-scalability, high-flexibility and simple-to-use.

1.3.1 Low-Cost and Cross-Platform

Sumptuous hardware or distinctive, specific software is a key problem averting grid technologies from being embraced by the general public. In order for grid solutions to be extensively adopted, two issues needs to be taken into consideration.

First is the cost issue, the TCO of owning a grid must be sufficiently acceptable. The second is the ability to connect various electronic devices existing today. In other words, the cross-platform characteristic of a grid solution directly determines the potential volume of the grid in the future.

1.3.2 High Performance Resource Sharing

Performance is undeniably the key measurement in evaluating a grid middleware. People expect grids to be fast, reliable and stable, anything less would be intolerable. Performance in grid systems is effected by two sets of elements:

- The nature of the job submitted to the grid. Sequential computation is by nature the worst case in grid performance, compared to complete parallel computations.
- Grid architecture design. Bad task scheduling algorithms could lead to potential bottleneck of the whole system, while naive control-message routing strategy could quickly overload the grid environment.

From the middleware provider's point of view, the first element is beyond our scope.

Nevertheless, the second element is the responsibility of a good grid middleware.

1.3.3 High-Scalability

The performance growth of a grid is direct proportional to how fast it can scale, e.g. the more computers that is currently within/joining the grid, the faster it processes tasks. Scalability can be classified into two categories: static and dynamic. Majority of commercial grids supports only static scaling, the grid size is fixed to a predefined cluster of computers, new computers joining the grid will require manual modification of attributes in the central task dispatching server. Dynamic scaling does not require manual interference, new computers notifies the grid of its existence automatically. Grid systems should be able to scale dynamically and scale high.

1.3.4 High-Flexibility

A grid framework should preserve resilience for application developers. Grid computing is an approach that lets you organize widespread, diverse collections of resources into a more uniform, manageable, visual whole. The resources we are referring here does not narrowly limit to CPU or storage, it might refer to anything with digital representation, e.g. Multimedia files, libraries, data, applications…etc. A good grid middleware should be as flexible as possible, it should not confine the innovation of grid application programmers within the scope of a badly designed middleware API.

In other words, creativity should not be limited by the framework.

1.3.5 Simple-to-Use

Grid systems involves complicated low-level network communications and protocol design. A grid middleware has the responsibility to hide these underlying complexities.

A set of simple-to-use Application Programmer Interfaces(APIs) should be provided for application designers. Furthermore, cumbersome grid administrative jobs should be made simple by utility tools, provided with the middleware.

1.4. Research Contribution

To achieve the objectives listed in section 1.3, we encountered various perplexities while designing mGrid. This thesis discusses the issues that were encountered and our corresponding solutions. The major contributions of this research can thus be categorized into seven parts.

- We crafted a low-cost, pure software-based high performance grid solution that works on various devices with java support. Figure 1-1 below depicts this cross-platform characteristic:

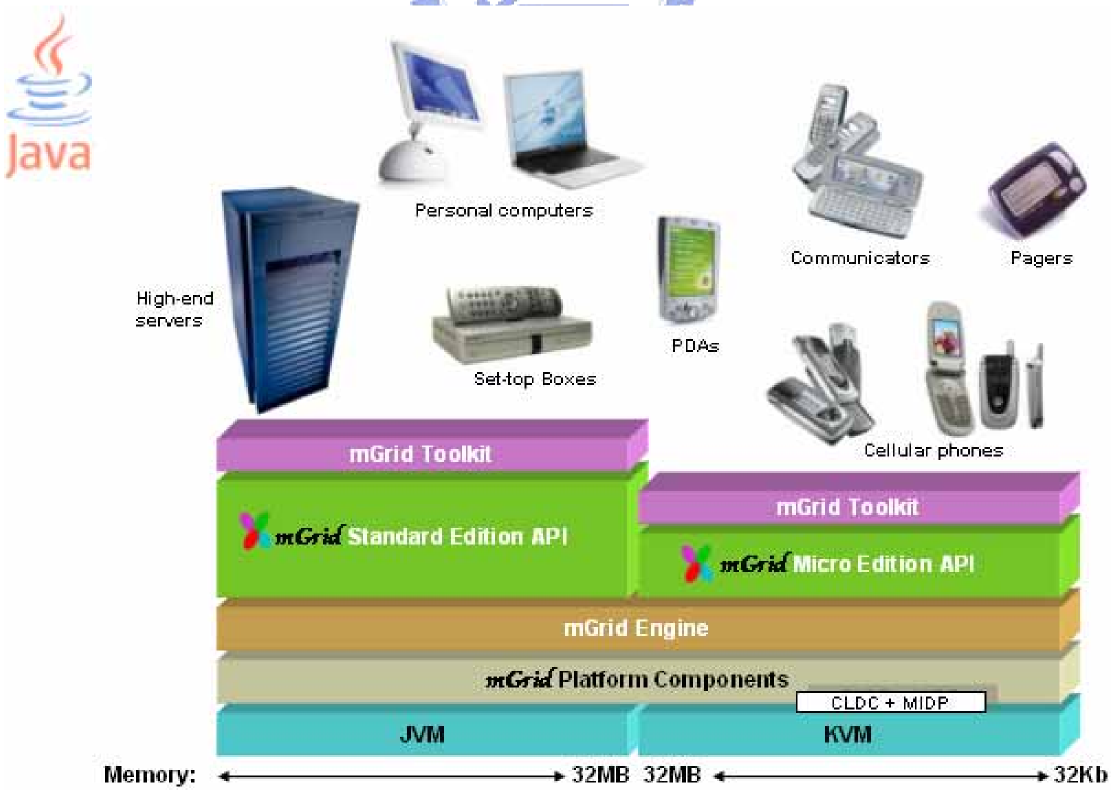


Figure 1-1: mGrid platform runs on various electronic devices with Java support

● We introduced a simple-to-use grid programming model.

● We introduced a set of utility tools to facilitate the administration of mGrid.

● We discussed the scalability issues in grid middleware, and implemented mGrid with a decentralized space-oriented architecture that supports dynamic grid expansion, and effectively solved the single-point-of-failure problem.

● We discussed the flexibility issues of grid middleware and proposed a considerably general platform for developers to write a variety of grid applications.

● We experimented mGrid performance with a modified version of the Linpack benchmark[3], a standard benchmarking program for commercial super computers.

● We analyzed the pros and cons of various grid design decisions.

1.5. Outline Of The Thesis

This dissertation is composed of six chapters. Chapter 1, this one, is introductory.

In Chapter 2, we introduce the background of grid computing methodologies and bring forth major commercial solutions for discussion.

In Chapter 3, we confer the detail implementations of mGrid framework. This section proposes our peculiar distributed algorithms and system architectures that render life to mGrid, pro and con of various design decisions is also debated here.

In Chapter 4, we switch to the developers' point of view by introducing the mGrid API. We can appreciate the ease in both writing grid applications and administrating grid environments using tools and APIs supplied with mGrid. This chapter also serves as a tutorial for the developers to operate the mGrid framework. Last but not least, some innovative example applications of mGrid framework is also presented here.

In Chapter 5, we put mGrid performance to the test using a modified version of Linpack benchmark[3]. Various experiments will be held and the performance of mGrid framework will be thoroughly quantified. Finally, in Chapter 6, we bring forth conclusion and a brief discussion on future works along with potential business opportunities.

1.6. SUMMARY

In this chapter we briefly described what grid computing is, what it can do, and the problems that exists with industrial grid solutions today. Grids are expensive thus intimidated the general public or SMEs(Small-Medium Enterprises) from adopting grid technologies. Furthermore, commercial grids still suffer from technical imperfections. Then we pointed out the objectives of this thesis and introduced our proposed grid system, named mGrid framework. In the end we categorized the major contributions of this research into multiple points.

2. BACKGROUND

2.1. Chapter Introduction

This chapter gives you the background of our research. To begin with, in section 2.2, we bring forth an interesting comparison of three popular models of super computing, namely super computers, physical clusters and virtual grids. A survey on their price, capability, size and such is revealed here. This survey serve as one of our basis in explaining why grid concept is gradually replacing traditional super computers or large mainframe clusters.

Next, we introduce the definition of grid computing according to TurboWrox Corporation[9] in section 2.3. What is a grid? How can it be applied? What applications are suitable to be submitted to a grid for processing? What are not?

It is known that commercial grid solutions are implemented using dissimilar architectures and communication models, which possesses different characteristics. Sections 2.4 and 2.5 discusses several approaches in implementing a grid, the pros and cons, and explain why we choose a particular model. Finally we introduce other related works in section 2.6.

Note that each survey we made has a certain level of impact on how we implement the mGrid Framework. We try to present to you concrete evidence referenced from other academic researches and industrial studies to justify our path of choice.

2.2. Super Computer, Physical Cluster and Virtual Grid

Grid computing is not the only option in efficiently solving complicated scientific calculations, there are other options existing. These options includes:

- Super Computers. Large and expensive singular computing hardware, normally used in areas such as weather condition modeling and nuclear simulation. Famous examples include NEC Earth simulator[10] and IBM ASCI White[11].
- Physical Clusters. Supercomputing devices consists of a large number of computers. Each of the computing node is interconnected using a LAN. Physical clusters usually resides within a single organization and is rarely open to the public. Jobs are dispatched to the back-end cluster for computation.
- Virtual Grids. PCs' computing capacity are donated freely to join virtual grids. Each PC is connected across the internet using software programs. Famous examples include SETI@Home[12], Folding@Home[13] and GIMPS[14]. A virtual grid usually span across several geographical locations.

These three options possesses different characteristics. Entry barriers in adopting each of the mentioned technology is also different. Table 2-1 presents a comparison of these three choices, using commercial products as example:

Computing Option	Name	Specification	Cost (million USD)
Super Computer	IBM ASCI White	8192 RS/6000 processors 6TB memory	**$110**
Super Computer	NEC Earth Sim	5104 vector processors 16GB memory	**$350**
Physical Cluster		100,000 Intel P4 1G processors 256MB memory	**$213**
Virtual Grid		100,000 Intel P4 1G 256MB memory	**Absorbed by PC owners**

Table 2-1: Comparison of super computers, clusters and grids [10][11][12][13]

According to a market research report by the Insight Research Corporation in 2004, cost is the primary decisive factor for the adoption of super computing devices[15]. This means, the higher the TCO(Total Cost of Ownership) for an enterprise to obtain a grid, the lower chance that they will actually adopt the technology. Table 2-1 shows that virtual grids proliferated an undeniable attraction due to its low-cost. Therefore, one of our major research objective is in creating a low-cost grid framework that provide all the necessary functions of a grid.

mGrid is a pure java-based grid framework. The java language provided cross-platform abilities so that an enterprise can interconnect all of their internal computers using mGrid Framework, regardless of their operating systems. This thoroughly utilizes all the idle resources in an enterprise without having to purchase any new hardware. TCO is thus lowered to an acceptable range with mGrid.

2.3. Grid Computing Definition

IDC and Insight Research Corporation predicts that worldwide grid spending will grow from $714.9 million in 2005 to approximately $19.2 billion in 2010[15]. With all the hype in the future of grid computing, it is surprising that there is still a lack of approval on what it is. TurboWrox Corp's definition is a pragmatic one[9]. It is a computing model that:

- Aggregate a set of diverse, widespread, distributed CPU resources into an organized virtual supercomputer.
- Aggregate a set of diverse, widespread, distributed Memory resources into an organized virtual system memory.
- Aggregate a set of diverse, widespread, distributed Data resources into an organized virtual data warehouse.

- Provide a unified visual view of the set of disperse resources mentioned above.
- Provide simple management, administration and utilization of distributed resources spanning across the network.

TurboWrox's grid definition left out one important item:

- Grid is flexible.

mGrid sees grid computing not only as a mean to aggregate computing resources, but also as a platform for innovative grid applications. It needs to be extremely flexible for developers to write various creative applications other than computation-based programs. See chapter 4 for more creative grid applications written with mGrid.

So what kind of application is suitable for a grid?

Our answer is that parallel programs are more applicable to grids. By parallel programs we refer to a set of procedures that do not interfere with one another. Each step in the program is independent. An example of such system is a distributed searching program that allows searching on individual grid nodes. Parallelism ignite the full potential of grids, see chapter 5 for the quantification of our statement here.

In summary, mGrid matches all five of TurboWrox's grid definitions, and we added an extra definition of our own, by allowing more flexibility in mGrid Framework.

Now that we understand the basic background of grid computing, what it is, what it can do, we now dive into more advanced discussions: underlying technical variations of grids. We begin with higher-level architectural options, then lower-level communication model options.

2.4. System Architectures

Three possible types of grid system architectures are Client-Server, Peer-to-Peer and Space-oriented. We discuss each individual approaches and analyze their advantages and disadvantages.

2.4.1 Client-Server Architecture

Client-Server model is simple and common in the world of network. Certainly, it can be applied to grid computing as well. In the Client-Server model, a grid user submits jobs to a centralized job dispatcher, this dispatcher then "dispatch" jobs to a appropriate node within the grid for processing, according to the current load of each computing node. Figure 2-1 depicts such model:

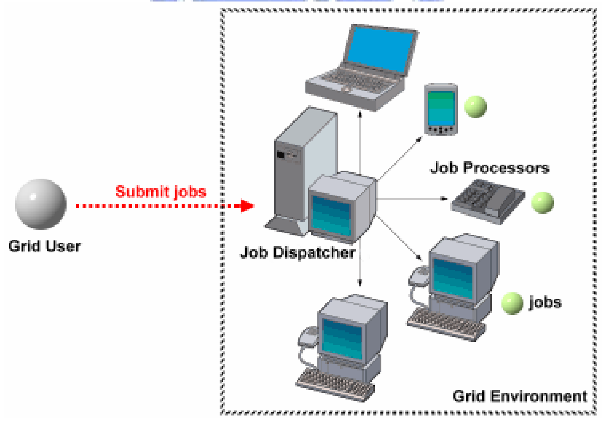


Figure 2-1: a Client-Server based grid architecture[16]

The major advantage of Client-Server model is in its easy management nature. The central dispatcher also serves as the management node, thus the administration of each computing node can be conducted by directly linking to the dispatcher server.

Its disadvantages includes the Single-point-of-Failure(SPF), and low-scalability. SPF refers to the situation when the central dispatcher server crashes, the entire back-end grid is immediately rendered useless. As for the second disadvantage, low-scalability is obvious when a new compute node joins the grid, configuration needs to be manually made in the dispatcher server. This is an extremely tedious task when you need to substantially expand your grid size.

2.4.2 Peer-to-Peer Architecture

The Peer-to-Peer grid model works in a simple manner: a grid user simply pass the jobs to its immediate neighbors for processing. Sometimes your task will be flooded across the whole p2p network depending on the grid algorithm the system applies. Figure 2-2 illustrates a p2p grid architecture:

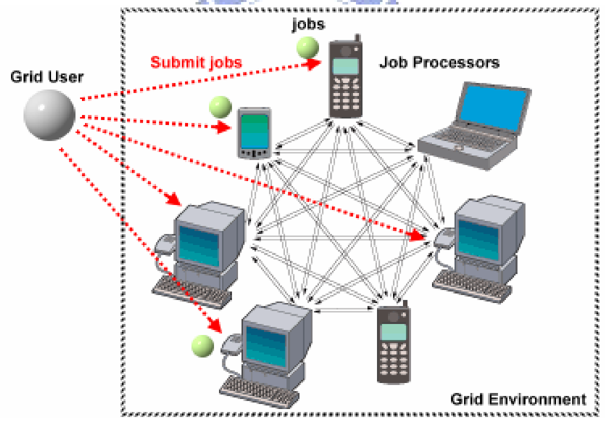


Figure 2-2: a Peer-to-Peer based grid architecture[16]

The advantage of a peer-to-peer architecture is that it does not have a centralized control, thus the SPF problem mentioned in section 2.4.1 is eliminated. Furthermore, new nodes can be added to an existing network without much effort.

The disadvantage is in its state consistency. What happens when a node with a job on hand suddenly crashes? How would we know which job is lost? Is it really lost or is it still under processing somewhere deep in your p2p network? The second downside is that the entire network state needs to be maintained by the grid user itself, the grid user needs to know the condition of each of its neighbor node in order to pass jobs to the right neighbors for processing. This enormously increases the overall workload of the grid user.

2.4.3 Space-oriented Architecture

Space-oriented concept originates from the Linda programming model from Yale University[17]. It basically works as follows: a grid user submits jobs to a storage space on the network, each computing node then fetch jobs randomly from the space to process. After the processing completes, the results were passed back into the space, the grid user then collects and combines the results from the space. Section 3.2.4 explains the space concept and our implementation in more detail. Figure 2-1 depicts a simple space-oriented architecture:

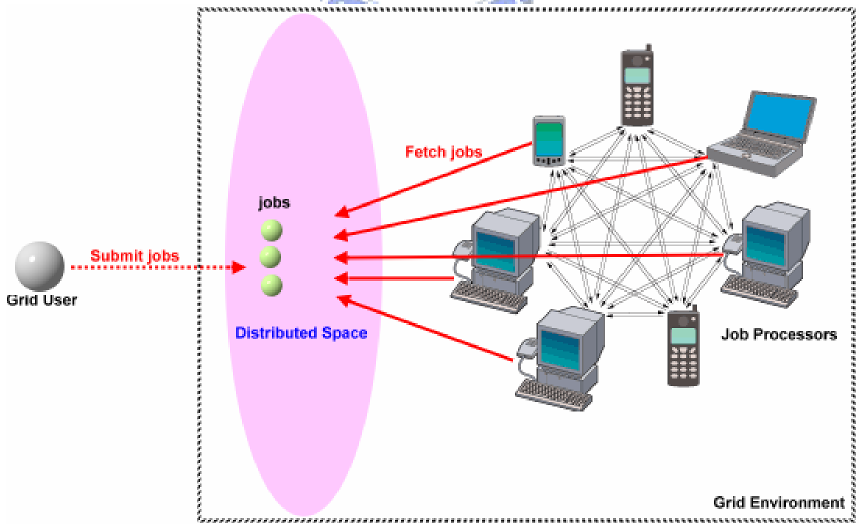


Figure 2-3: a Space-oriented grid architecture

The space-oriented architecture seemed to be a panacea for grid computing. It has the advantage of both Client-Server and Peer-to-Peer architectures, yet it solved most of the problems occurring in both models[17]. First, the space can be distributed across the network thus the SPF problem in 2.4.1 is settled. Secondly, all the compute node needs to register itself to the space upon start-up, thus the space manages and monitors the entire grid for the user.

One important characteristic of space-oriented grids is that each compute node spontaneously fetch jobs FROM the space for processing, this differs from both the Client-Server and Peer-to-Peer models in that these two models push the jobs TO the compute nodes for process. This implies one more advantage: each node in a space-oriented architecture is allowed to leave the network at will.

Due to the advantages the space-oriented architecture has over the other two models, mGrid chooses the space-oriented model as the underlying system architecture.

Next, we discuss the lower-level communication protocol options.

2.5. Communication Models

Grid nodes communicate with one another by means of communication protocol. Since different protocols bring about different effects on a grid system, we need to understand the features of each mechanism and decide which is most appropriate for grid computing systems.

Two models are introduced in this section, namely Synchronous Transmission and Asynchronous Transmission.

2.5.1 Synchronous Transmission Model

Synchronous transmission refers to the fact that when a client requests a remote service call, the execution process is temporarily suspended until a reply is received from the remote service. An implementation of such concept is the RPC(Remote Procedure Call) technology. Most commercial products such as SUN Grid[2] and IBM IntraGrid[18] are based on synchronous transmission mode.

2.5.2 Asynchronous Transmission Model

Asynchronous transmission, on the other hand, allows the service requestors to continue running after a request is sent, without blocking the entire program waiting for a reply. Examples of asynchronous transmission are MOMs(Message-Oriented Middleware) such as JMS(Java Messaging service)[19].

Synchronous and asynchronous transmission have advantages and disadvantages. The latter tends to be more robust to failures, while the former tends to be easier to develop with.

So which transmission model is suitable for grid computing? Reference [19] uses a simple M/M/1 queuing model to prove that for a piece of program that is consisted mainly of parallel codes, the overall performance of the asynchronous model is better than the synchronous model. Furthermore, grid computing applications by nature are supposed to be parallel, submitting sequential programs to a grid is essentially senseless. Therefore we expect the majority of mGrid users will utilize our framework in solving parallel problems. With these facts in mind, we decided to use the asynchronous transmission model for mGrid Framework.

Note that many commercial products are synchronous-based, thus we expect to have a better start than these grid solutions by choosing the correct transmission mode in advance.

2.6. Related Works

Many international research institutes and companies have collaborated in developing various projects associated with grid computing. These projects can be horizontally classified into *specific-grids*, *general-grids* and *grid middlewares*.

SETI@Home[12], Folding@Home[13] & GIMPS[14] are examples of *specific-grids*, meaning each of them solves only very specific problems. SETI@Home allows you to download a software that turns your computer into a node within SETI's grid, this software analyzes radio telescope data using the CPU resources of your PC and transmits results back to SETI central server. Folding@Home uses a similar architecture to studies protein folding, misfolding, aggregation, and related diseases. GIMPS works in the same fashion only it conducts a different job.

SUN Grid[2], IBM Intra Grid[18] and ALiCE[20] are examples of *general-grids*. These grid solutions do not restrict the logic of the applications running on-top of them. SUN Grid software typically bundles with SUN blade servers and allows jobs to be submitted to it. IBM Intra Grid provides an experimental worldwide-scale grid system accessible to all IBM employees. ALiCE is a java-based grid solution developed by National University of Singapore, applications such as protein modeling is written and tested with this framework.

Finally, Globus toolkit[21], Legion[22], JXTA[23] and GridSim[24] are considered *grid middlewares*. Globus is an open system that provides a set of basic services. Users can build higher-level services using lower-level services. Globus is largely platform dependent and requires UNIX to run. Furthermore, its complicated infrastructure setup, application development and deployment created a high learning curve both in mastering and using Globus. Legion is a toolkit that treats all software and hardware in the grid as objects, and provide remote method calls between these objects. JXTA is a set of protocols developed by SUN Microsystems to ease the development of p2p application, different grid systems can be built by using these protocols. Last but not least, GridSim offers a complete solution in the simulation of grid networks.

Most of the above researches are built on-top of the Client-Server architecture, and utilizes Synchronous transmission as underlying protocol. From sections 2.4 & 2.5 we pointed out that some of these technical decisions are probably not the best ones.

2.7. SUMMARY

This concludes our research background. We understand that there are multiple ways to super computing, and virtual grids offer a cost-effect and attractive option. Companies that cannot afford high TCO in purchasing grid solutions should consider about adopting grids that are purely software-based, such as mGrid.

Virtual grids can be implemented using various design options, such as determining the system architecture and communication models. In sections 2.4 & 2.5 we used concrete research results that proves the following:

- Space-oriented Architecture is a good choice for grids
- For parallel computing, asynchronous transmission model is more appropriate.

Finally we give an introduction on other related works of grids. The surveys and observations done in this chapter has influential impact in our design of mGrid Framework. In the next chapter we will walk you through our underlying implementation design in a thorough manner.

3. SYSTEM ARCHITECTURE

3.1. OVERVIEW

The proposed solution in this thesis, named mGrid framework, is a low-cost, pure java-based, high-performance grid solution. mGrid attempts to migrate grid computing concept onto mobile devices(e.g. Personal Digital Assistants, Cellular phones) and onto large computational equipments(e.g. PCs, mainframes), which has minimum network connectivity support. Figure 3-1 depicts the macroscopic view of the mGrid framework:

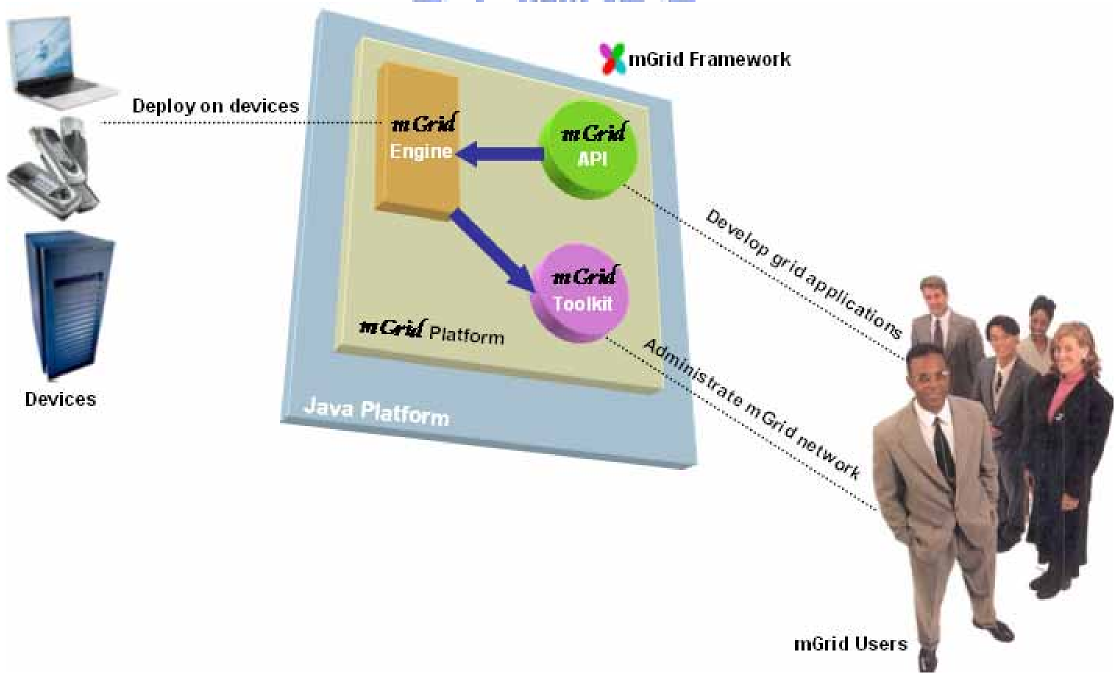


Figure 3-1: Macroscopic view of mGrid framework. mGrid is consisted of four major portions

mGrid framework is consisted of four major portions:

- mGrid Platform
- mGrid Engine
- mGrid Toolkit
- mGrid API

mGrid Engines can be installed on devices with java support. The J2SE[4] and J2EE[4] version is fully operational, while the J2ME[4] version currently has minimal functions. Engines deployed on devices automatically constructs a mGrid network environment that support transaction, natural load-balancing and security.

Application developers then use the mGrid API to compose various grid applications that utilizes the mGrid environment formed by the engines. Note that at least a single engine must be started for a mGrid network to be successfully built. We also provide a set of useful tools in the mGrid toolkit to allow easy monitoring and administrating of mGrid networks.

In this chapter we will focus on the implementation methodologies and distributed algorithms of three items: *mGrid Platform*, *mGrid Engine* and *mGrid Toolkit*. In chapter 4 we will discuss the mGrid API in depth.

We will emphasize on the J2SE and J2EE version of mGrid framework.

3.2. mGrid PLATFORM

Before we talk about mGrid Engine and mGrid Toolkit, we need to have some basic understanding of the platform itself. Note that both the engine and toolkit rely on the grid infrastructure constructed by the software components within mGrid platform. mGrid platform is an augmented version of SUN's JINI network technology[8], by augmented version we mean that it provides additional features such as including more specialized service components dedicated to grid computing.

Figure 3-2 shows the in-depth components that constitute the mGrid platform:

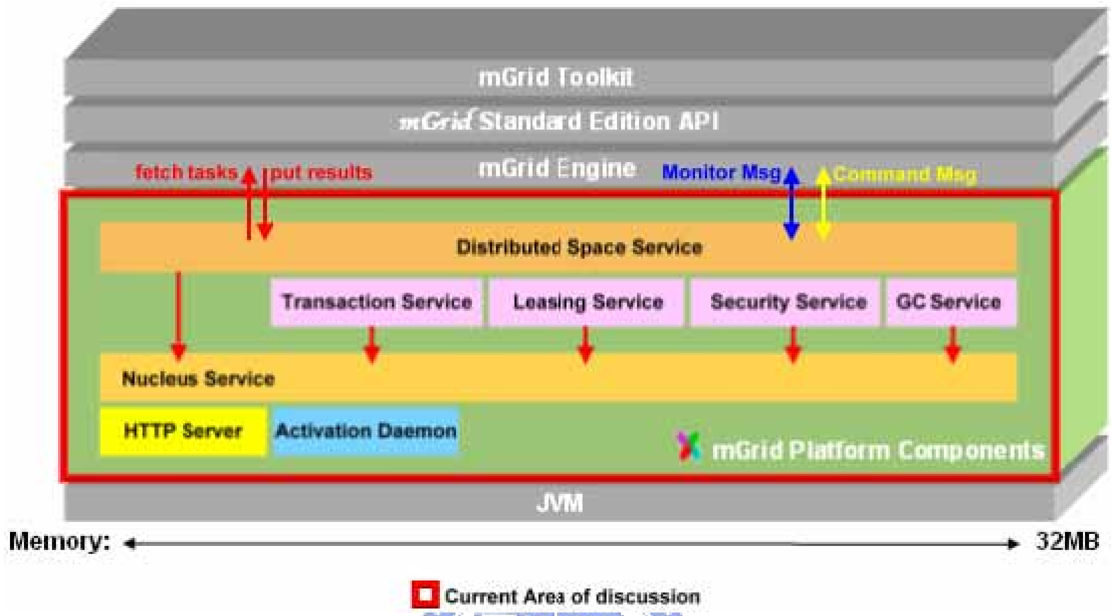


Figure 3-2: In-depth component view of the mGrid Platform, indicated in red

mGrid platform adheres to the SOA(Service-Oriented Architecture) concept[5], each component can exist as a remote service across the internet. For instance, we can start Nucleus service, Transaction service and Security service on computers A , B and C lying on the network. These services uses mGrid's underlying protocol to search, discover and communicate with one another, as if they were running on the same machine. Protocol details will be described in section 3.2.3. SOA allows the stress of executing services to be evenly-distributed across the network, so that no single computer will be overloaded[5].

Now we introduce the individual components within mGrid platform. We begin with two lower-level components first: HTTP Server and Activation Daemon.

3.2.1 HTTP Server

mGrid platform requires this facility because for many vital operations to realize, code needs to be dynamically downloaded from some remote service running somewhere on your network. The actual transmission of java code take place via the HTTP protocol. The implementation of our server is minimal, it only supports the GET operation, which is sufficient for code downloading.

In general, any code that may need to be downloaded across the network has to be accessible from a HTTP server instance.

3.2.2 Activation Daemon

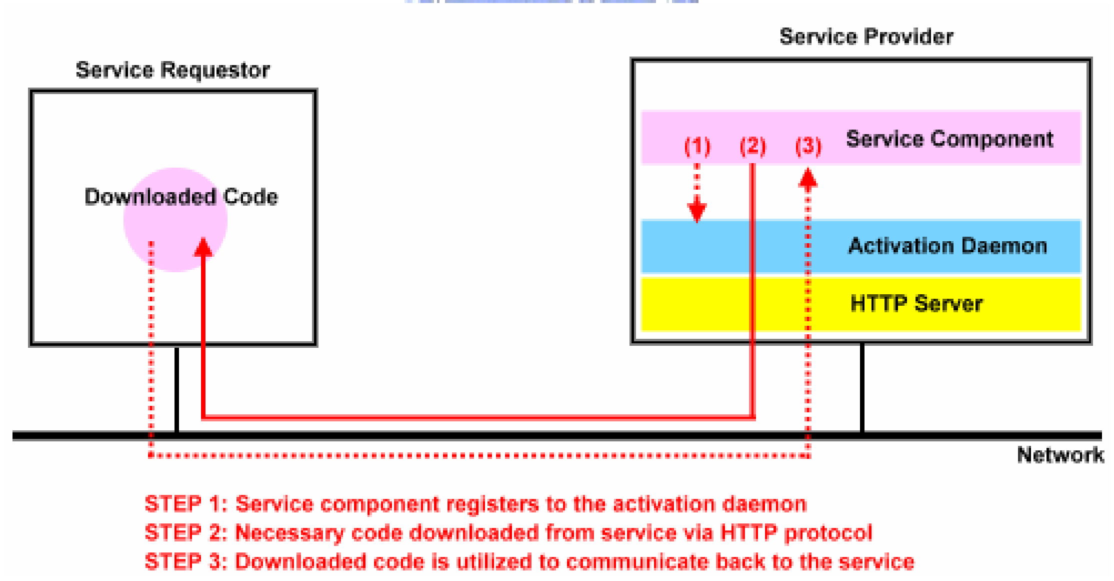
An activation daemon[6] is a piece of software which allows services that is invoked only rarely to essentially “hibernate”, and be automatically awakened when they are needed. Every service component will need to register itself with an activation daemon instance before running. Activation daemon has two major responsibilities:

- Service hibernation & de-hibernation: Manage the transition between active and inactive states for each service component.
- Service self-recovery: Restart a particular service after it crashes, restoring it to its previous state before the crash.

We make use of the activation daemon software that comes with J2SDK 5.0[4]. At minimal, you will need to run an instance of activation daemon on each host that runs services. The daemon creates log files that contains information of the activable service which has registered itself to the daemon. State transition and crash recovery relies on the information saved within those log files.

The reason we apply activation daemons not only is because it is able to recover services after a crash, but also economizes the use of system resources by sending currently unused service components into “hibernate” mode. The down-side is that it adds an extra layer below each service component, efficiency is therefore decreased during a service’s initial start-up time by approximately 7.5%, but proposed no further decreases in subsequent service calls. We decided that this is a minor trade-off compared to the valuable capabilities it adds to our platform.

In summary, the mGrid platform requires both HTTP server and activation daemon for services to pass necessary java codes across the network and to be self-recovery. This concept is exhibited in Figure 3-3:



Service Requestor
Downloaded Code
Service Provider
(1)
(2)
(3)
Service Component
Activation Daemon
HTTP Server
Network
STEP 1: Service component registers to the activation daemon
STEP 2: Necessary code downloaded from service via HTTP protocol
STEP 3: Downloaded code is utilized to communicate back to the service

Figure 3-3: Underlying code transmission steps for mGrid platform

Now that we understood how the lower-level code passing operates, we can start to probe into the upper-level service components provided by mGrid platform, namely Nucleus service, Transaction service, Leasing service, Security service, GC(Garbage

Collector) service and Distributed Space service. Note that these services relies heavily upon the schemes described in sections 3.2.1 and 3.2.2.

3.2.3 Nucleus Service

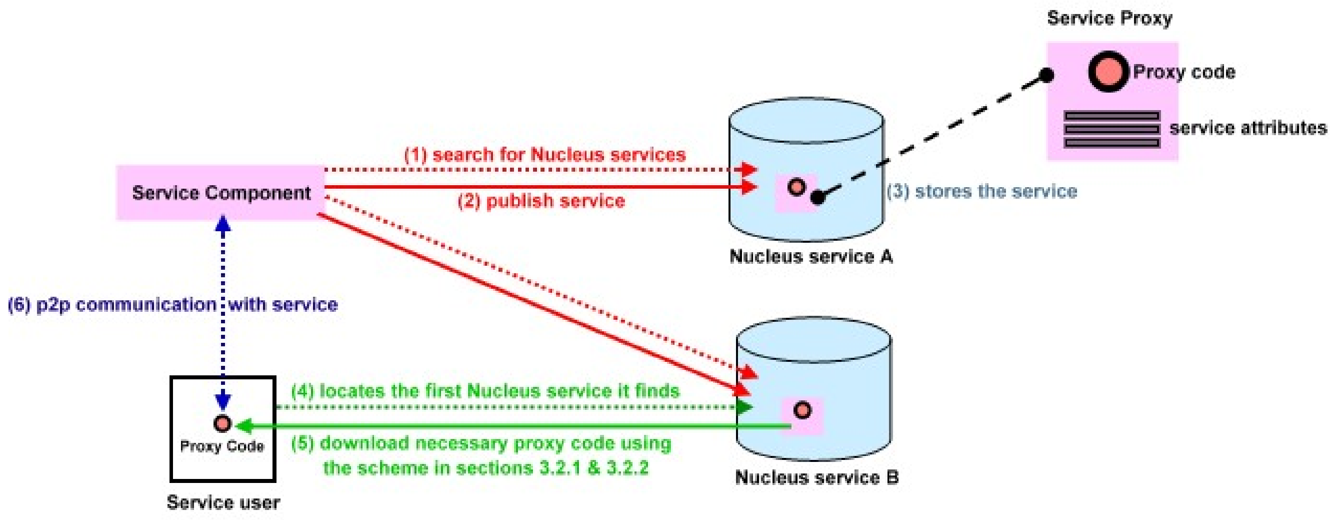
As the name implies, this service is the central core among the other services listed in the mGrid platform. A good analogy would be our solar system: The Nucleus service will be the sun, while the other services within mGrid platform are the planets constantly revolving around it, all using the functionalities the Nucleus service provides.

You can think of Nucleus service as a kind of naming/directory service[7], it keeps track of all other mGrid services currently running on the network. However, it differs from traditional naming/directory services, which only provides simple string-object mapping, the Nucleus service supports java type search, i.e. You can search for a particular service using the interface it implements or any of its super-interfaces.

The Nucleus service co-operates with other active services using the following steps:

1. A new mGrid platform service component searches for Nucleus services upon start-up, using IP multicast(in LAN) or unicast(beyond LAN).
2. The service component publishes the attributes and proxy code of the service it provides to the Nucleus service.
3. Nucleus service saves the attributes and proxy code.
4. A service user searches for a Nucleus service upon start-up, using IP multicast(in LAN) or unicast(beyond LAN).
5. The service user downloads the necessary proxy code it requires from the first Nucleus service it found.
6. The service user communicates with the service component in a p2p manner using the proxy code.

The above steps are illustrated in Figure 3-4:



Service Proxy
Proxy code
service attributes
(1) search for Nucleus services
Service Component
(2) publish service
(3) stores the service
Nucleus service A
(6) p2p communication with service
(4) locates the first Nucleus service it finds
Proxy Code
(5) download necessary proxy code using the scheme in sections 3.2.1 & 3.2.2
Nucleus service B
Service user

Figure 3-4: How Nucleus service works with the other services from mGrid platform

Note that steps 2 and 5 in Figure 3-4 utilizes the underlying dynamic code download scheme introduced in section 3.2.1.

Each Nucleus is also fault-tolerant, two mechanisms makes this possible:

- Each Nucleus service relies on the activation daemon described in section 3.2.2 to recover its state after a crash or restart. So you must run an activation daemon on each machine that runs a Nucleus.
- You have the option of running multiple instances of Nucleus service on your network. This redundancy allows unexpected failures of some Nucleus. It means as long as a single Nucleus lives, the mGrid platform can perform its duties as if no failure occurred, since each service user requires only a minimum of one Nucleus for proxy code downloading (see step 4 in Figure 3-4). This can be summarized into a simple mathematical formula:

(Max num of Nucleus failure tolerated) = (Total num of Nucleus started) – 1	**(1)**

Before discussing other service components in mGrid platform, we need to keep in mind that all the mGrid platform services needs to register itself to the Nucleus service upon start-up. Nucleus service keeps track of all mGrid service components currently running, and is capable of making them visible to service users, even if users have no previous knowledge of where the service components are on the network.

This interaction between service components, service users and Nucleus is illustrated once more using an UML sequence diagram in Figure 3-5:

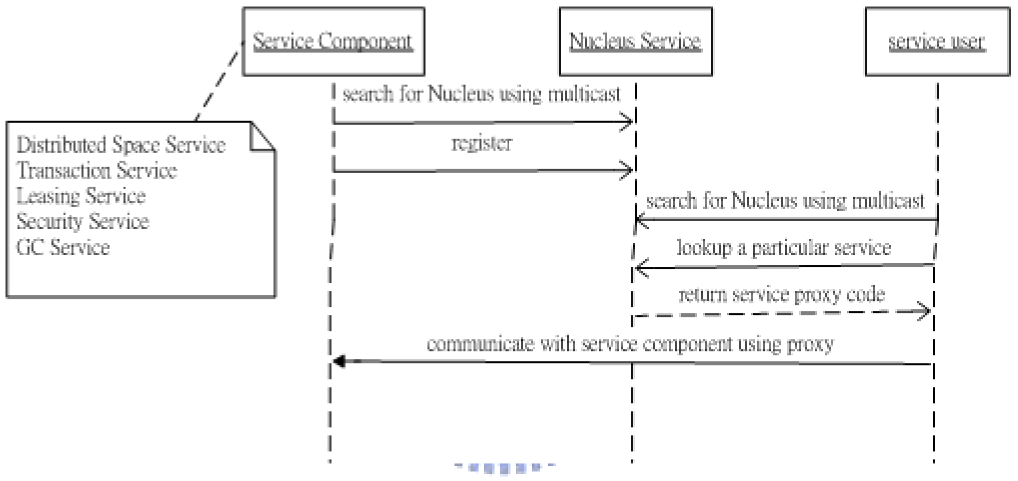

Service Component
Nucleus Service
service user
Distributed Space Service
Transaction Service
Leasing Service
Security Service
GC Service
search for Nucleus using multicast
register
search for Nucleus using multicast
lookup a particular service
return service proxy code
communicate with service component using proxy

Figure 3-5: All service components needs to register with the Nucleus service upon start

3.2.4 Distributed Space Service

The Distributed Space Service serves as the job exchanging location for our grid system, all jobs are transmitted to and taken from a space. From this point on we shall refer the distributed space service as simply “space” for convenience.

The concept of the space-oriented grid has been introduced in section 2.4.3, let’s add it with more detailed explanation here. Figure 3-6 shows the space-oriented grid concept:

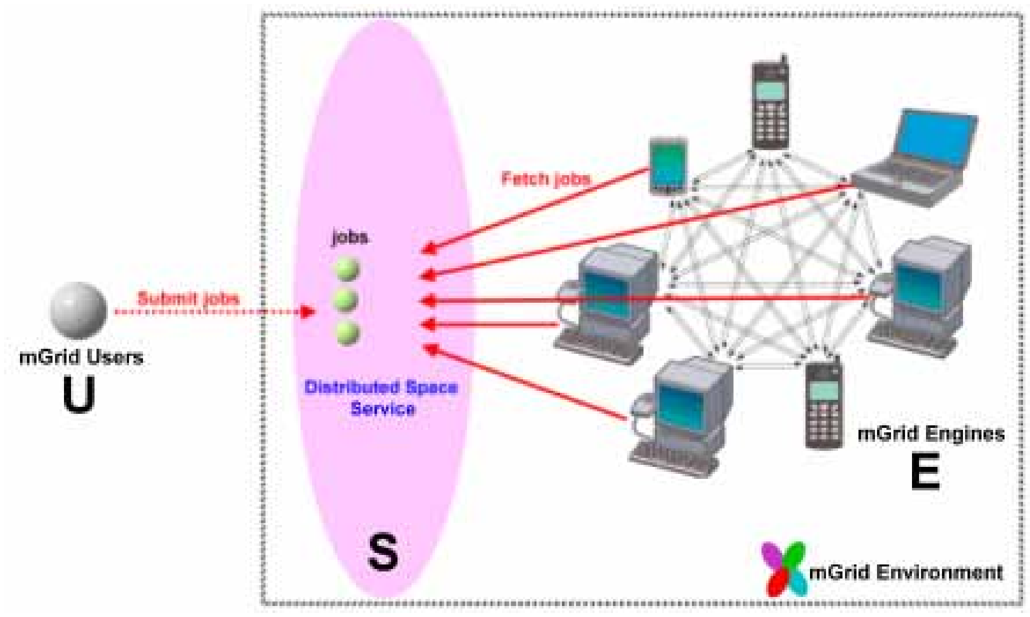


Figure 3-6: The Space-oriented grid architecture used in mGrid Framework

The space works in a very simple manner. Suppose we have multiple users **U**, a single space **S**, and multiple Engines **E**:

1. **U** submits a series of parallel jobs to **S**.
2. **E** fetch the jobs randomly from **S**.
3. **E** process the jobs.
4. **E** put the results back to **S**.
5. **U** collects the results from the **S** for final presentation.

Note that each space need to register itself to the Nucleus, refer to Figures 3-4 & 3-5. After a space has successfully registered itself to a Nucleus, it is then visible to both mGrid users and mGrid Engines for discovery and use. This space discovery process will be explained in more detail in section 3.3 later on. Right now we only need to know that a space acts as the central job exchanging ground for our grid Framework.

So what is a space exactly?

The simple answer is that a space is a piece of temporary memory residing on the network that is consisted of multiple computers. Computers participating in the same space can share each other's memory and storage space. Figure 3-7 depicts a space:

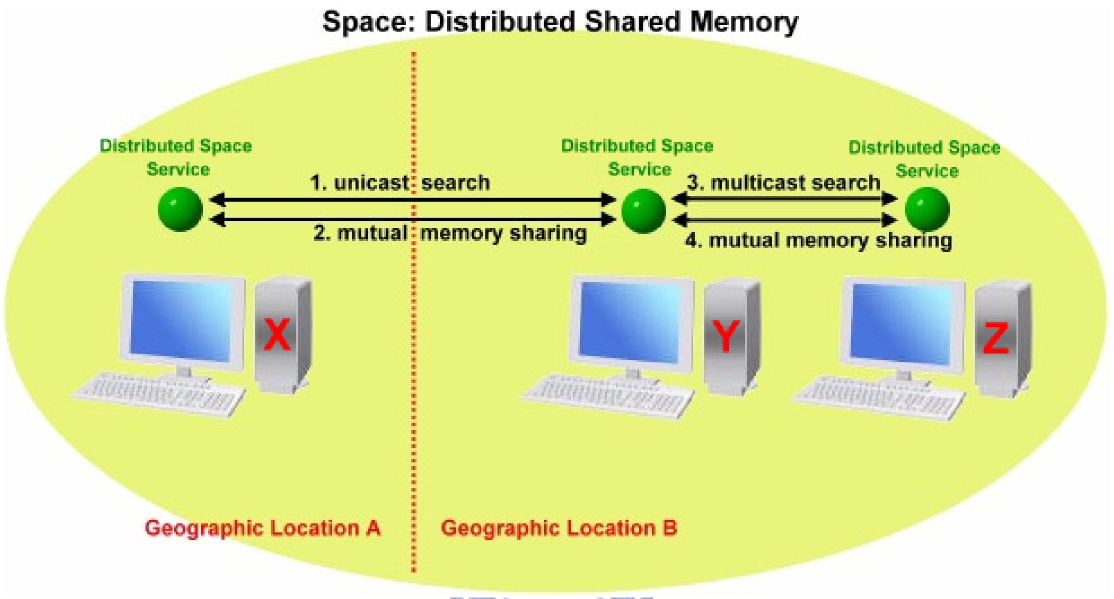


Figure 3-7: a "Space" is consisted of several computing devices

Figure 3-7 illustrates a single space consisted of three personal computers X,Y and Z. computer X sits at geographic location A, while computers Y and Z sits at geographic location B. By initiating a distributed space service on each of the computers, we combine them into a single logical space entity that is consistent and shares memory/storage resources, regardless of their actual geographical whereabouts. In other words, a space service is a virtualization middleware which connects computer memories across the network. Space service uses multicast to search for other space services in the same LAN, while unicast is used if the other space services are located outside of the LAN.

A space only supports three simple operations: ***1. Fetch 2. Put*** and ***3. Read***. These three very basic operations proves to be extremely useful and sufficient. Suppose we have a Genetic Algorithm computation on hand, each step can be first disassembled into tasks. Each task is then ***Put*** into the space by the client. The engines ***Fetch*** these tasks from space and does the processing, then it ***Put*** the results back into the space. Finally, the client ***Read*** the results in space and reassemble them for presentation.

A space utilizes the other services, namely Transaction service, Leasing service, Security service and GC(Garbage-Collector) service, to provide add-on functionalities such as safe-transaction, space garbage-cleaning, space access-authorization and leasing. These final four mGrid platform service components will be introduced in sections 3.2.5, 3.2.6, 3.2.7 and 3.2.8 below.

3.2.5 Transaction Service

A transaction service needs to register with the Nucleus before being used by other service components, see section 3.2.3. Transaction service provide the ACID properties to data manipulations. In simple words, it allows a series of operations to complete altogether, if a single operation in the whole series fail, the transaction fails, and everything gets rolled-back to its initial state.

Our implementation of Transaction service supports the *2-phase commit* protocol. Note that the distributed space service utilizes the transaction service to insure data integrity, the space is required to discover the transaction service through the Nucleus (consult section 3.2.3) before it can be transaction-enabled.

Figure 3-8 illustrates the transaction steps in mGrid platform using UML sequence diagram. The interaction between mGrid Client, the Distributed space service, Transaction service and mGrid Engines is the center focus, we will not show transaction-unrelated steps such as the discovery of services.

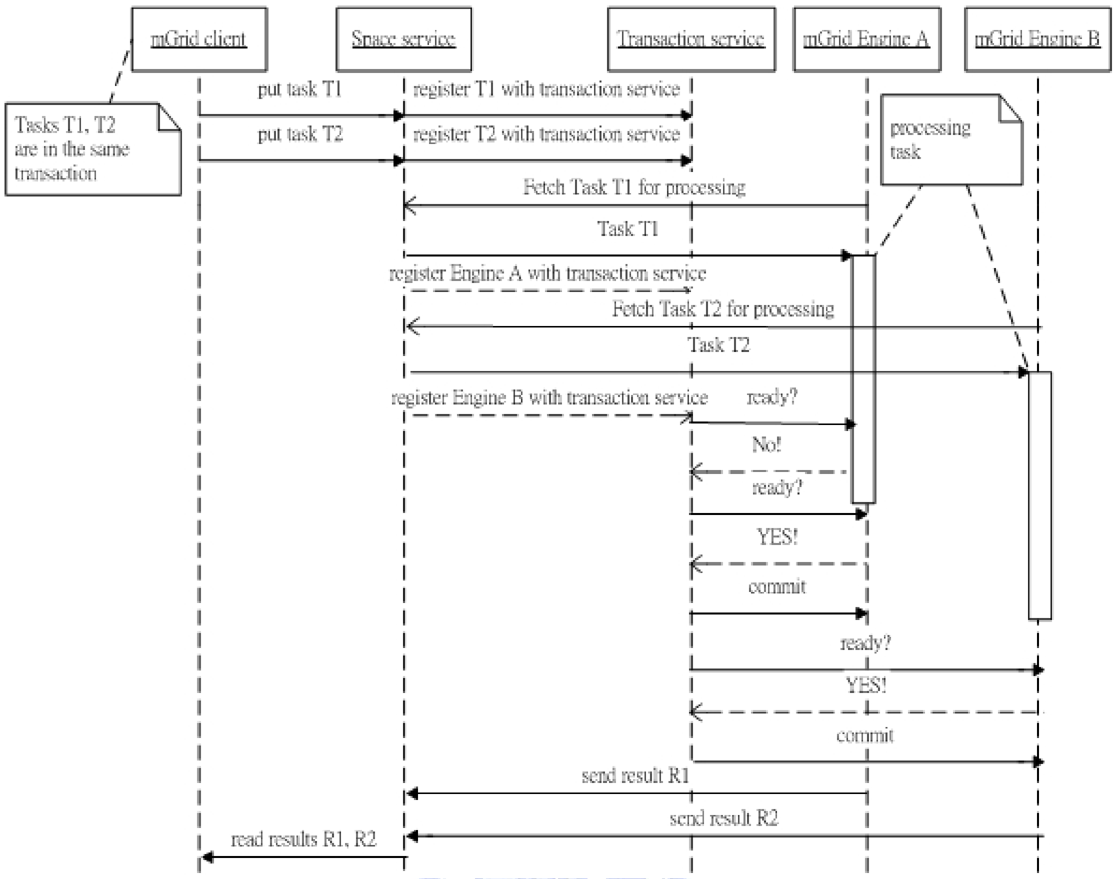

mGrid client
Space service
Transaction service
mGrid Engine A
mGrid Engine B
Tasks T1, T2 are in the same transaction
put task T1
put task T2
register T1 with transaction service
register T2 with transaction service
processing task
Fetch Task T1 for processing
Task T1
register Engine A with transaction service
Fetch Task T2 for processing
Task T2
register Engine B with transaction service
ready?
No!
ready?
YES!
commit
ready?
YES!
commit
send result R1
send result R2
read results R1, R2

Figure 3-8: Transaction service supports the 2-phase commit protocol: Either all or none!

Let us explain Figure 3-8 step by step. Assume tasks T1 and T2 belong to the same transaction, this means T1 and T2 must both complete successfully or neither will. Transaction service keeps this principal in mind and constantly polls Engines A and B using the 2-phase commit protocol. If both Engines A and B succeed in processing T1 and T2 consecutively, then the transaction service sends the *commit* message to both engines, finishing up the transaction. If either engine fail to finish processing a task, then the transaction is considered a failure and roll-back procedure is taken. The system returns to its initial state and re-processes T1 and T2 again.

We use a simple pseudo-code in the next page to demonstrate the transaction algorithm described above:

```
T1, T2 : belong to the same transaction;

Transacted processing of tasks(T1, T2)
{
        Step1. T1, T2 are put into the space.
        Step2. T1, T2 are registered to the Transaction service to be managed.
        Step3. Engine A and Engine B fetch T1 and T2 for processing.
        Step4. Engine A and Engine B are registered to the transaction service.

    While(Engine A || Engine B has not completed processing)
    {
        Step5. Transaction service asks Engines A,B if the processing has completed?
    }

        Step6. commit the transaction and the client read the results from space.
}
```

Listing 3-1: Transaction algorithm pseudo-code

The Space service needs to discover the Transaction service before using it, the discovery procedure is described in Figures 3-4 and 3-5, section 3.2.3.

3.2.6 Leasing Service

Leasing Service needs to register itself before being used. All objects sent to a space has a *lease time* attribute, indicating its TTL(Time-To-Live) within a space. A Leasing service manages a hashtable of object-TTL pair, and constantly removes the expired objects from a space. This allows the unused objects to be recycled and memory resources could be released, thus mitigates the loading of the entire grid.

However, *lease time* have the option to be renewed to prevent it from being discarded by the Leasing service.

3.2.7 Security Service

mGrid platform supports policy-based security model. The Security service reads a policy file before the mGrid platform starts, this policy file include all the policies that has to be obeyed. The following are two simple examples of a policy file:

```
grant {
      permission java.security.AllPermission "", "";
};
```

Listing 3-2: Policy File that allows total access to mGrid platform from anyone.

```
grant {
      permission java.security.AllPermission "", "";
      permission java.net.SocketPermission "192.168.11.2", "connect, refuse";
};
```

Listing 3-3: Allows total access except for incoming connection from IP 192.168.11.2

Security service provides a static method of authentication and authorization for mGrid platform. Currently the platform has been initially tested using the total access policy.

3.2.8 GC Service (Garbage-Collector Service)

GC service is short for Garbage-collector service. Like all the other service components in mGrid platform, it needs to register itself to the Nucleus before being used. GC service differs from the Leasing service described in section 3.2.6 in that it can force all the objects to be cleaned up, regardless of their *leasing time*. This service is useful in situations when the entire platform needs to be restarted.

Other uses of GC Service would be specifying a group of unwanted objects, such as illegal submitted tasks. The GC service removes these tasks without effecting the other regular operation of mGrid platform.

So far we have completed the introduction of the underlying mGrid Platform, and should have a brief understanding of how a space-oriented grid works(see section 3.2.4). mGrid Engine is another key portion of our framework that is built on-top of mGrid Platform. In the following section we will introduce the mGrid Engine.

3.3. mGrid ENGINE

mGrid Engine is built on-top of the mGrid Platform. Figure 3-9 shows the in-depth components that constitutes mGrid Engine:

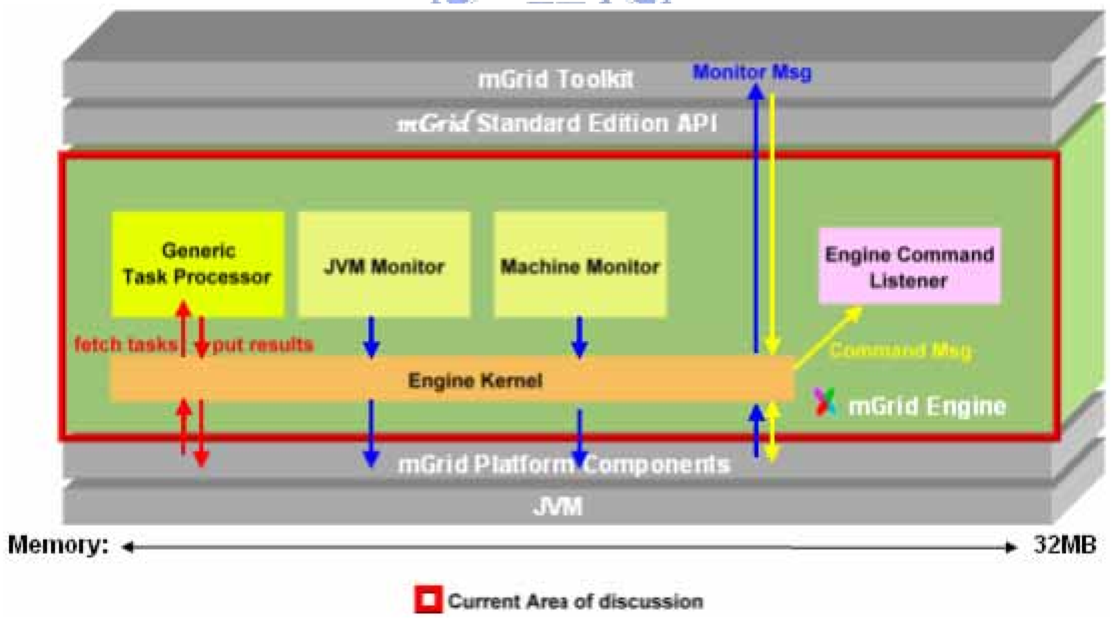


Figure 3-9: In-depth component view of the mGrid Engine, indicated in red

Engines can be deployed on a variety of devices with java support. After successful deployment and execution, engines allow a disperse set of devices to link together and form a virtual supercomputing environment that shares CPU, memory, storage, file resources and so forth. We call this environment the *mGrid Environment*.

Developers then utilizes the mGrid APIs (see chapter 4) to write various grid applications that access the resources harnessed within mGrid environment. These applications includes protein modeling, ray-tracing and prime number searching programs.

Now that we have a high-level idea of mGrid Engine's role, we begin introducing the inner-level components that propels the engine, starting with the core of the Engine: Engine Kernel.

3.3.1 Engine Kernel

Engine Kernel is the main component which enables the engine to communicate with the underlying mGrid Platform. The engine kernel has four major responsibilities:

- Initializes the engine, search for the space service instance provided by the mGrid Platform (see section 3.2.4 for distributed space service introduction).
- Fetches tasks from space, pass them to the Generic Task Processor for processing, then put results back to space.
- Acts as the middleman for monitoring messages sent to mGrid Toolkit (see section 3.4 for Toolkit monitoring details).
- Acts as the middleman for control messages sent from mGrid Toolkit (see section 3.4 for Toolkit controlling details).

If we look at engine kernel technically, an engine kernel assists the engine to search for space instances within mGrid Platform, and constantly *fetch* tasks from the space for processing. Figure 3-10 illustrates how an engine kernel interacts with a space using UML sequence diagram:

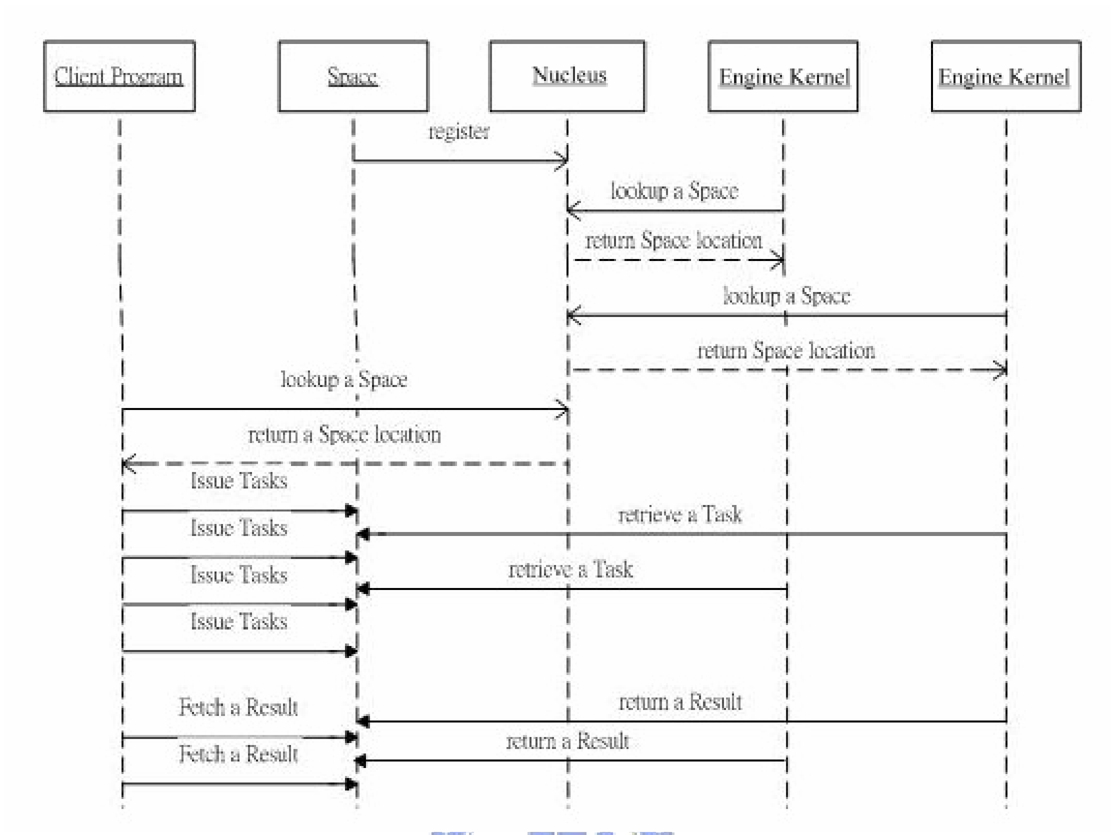


Figure 3-10: Engine kernel constantly fetch tasks from space.

A mGrid client uses multicast to search for the Nucleus service (section 3.2.3) and use it to lookup a space instance(section 3.2.4). After obtaining the space, the client program then feed the space with tasks that needs to be processed by the mGrid Environment. On the other side, engine kernel also uses multicast to search for a Nucleus and use it to lookup a space, then randomly *fetch* any available tasks that currently resides within the space. When the processing is completed, a result object will be fed into the space by the kernel, where it will be *read* by the client and reassembled with other results objects for presentation.

Note that the engine kernel only fetches the tasks from the space, it does not process it! Instead, the Generic Task Processor introduced in the next section manages the task processing.

3.3.2 Generic Task Processor (GTP)

After the engine kernel fetched a task from space, this task is passed to the Generic Task Processor(GTP) for processing.

As the name implies, the GTP is generic enough to handle different sorts of logical calculations, this is achieved through the dynamic class loading feature of the java programming language. In other words, when the GTP receives a task, the task's computational code will be dynamically loaded into the processor. The GTP does not have any previous knowledge of the task logic that it will process, everything loads in on the fly.

A naïve version of GTP pseudo-code looks like listing 3-4, it simply fetches a task from space, process it, then feed the result back to space:

```
While(true)
{
        aTask = EngineKernel.takeTaskFromSpace();
        result = aTask.process();
        EngineKernel.putResultInSpace(result);

        println("a task is processed");
}
```

Listing 3-4: A naïve GTP implementation pseudo-code

The above GTP pseudo-code has a critical flaw. What if the engine fails after it *fetched* a task but did not successfully process it? This task would then be lost. For a system with high-reliability demand this is not an acceptable situation.

Therefore our GTP implementation makes use of mGrid Platform's Transaction service (consult section 3.2.5 for detailed description on how transaction service works). By integrating transaction service into GTP, we ensure a highly-reliable grid system. A modified version of GTP pseudo-code is listed below:

```
While(true)
{
    TransactionService txn = Platform.createTransactionService();
      aTask = EngineKernel.takeTaskFromSpace();
    if(aTask == null)
    {
       txn.abort();
       return;
    }
      result = aTask.process();
    if(result == null)
    {
       txn.abort();
       return;
    }
      EngineKernel.putResultInSpace(result);
    txn.commit();

      println("a task is processed");
}
```

Listing 3-5: A modified GTP implementation pseudo-code – better solution

Listing 3-5 illustrates GTP with transaction support. If an engine fail to fetch a task from space, or if task processing fails, the transaction service will *abort* the transaction and attempt to roll-back the entire computation. Only when the calculation is guaranteed to be successful, will the transaction be considered finished and *committed*.

In summary, GTP acts as the central processing unit in mGrid Engine.

3.3.3 JVM Monitor

JVM Monitor digs the local machine's JVM profile for display in the mGrid Engine graphical user interface. Furthermore, it periodically sends heartbeat messages in the form of *EngineInfoEntry* containing the JVM attributes to a remote mGrid Toolkit software for administration purposes. All messages generated by the JVM Monitor goes through the engine kernel, which in turn passes the message to the distributed space service on mGrid Platform, where it is read by mGrid Toolkit for display.

JVM attributes includes JRE version, operating system name, OS patch level, JDK version and so forth.

See section 3.4.1 for *EngineInfoEntry* details.

3.3.4 Machine Monitor

Machine Monitor reflects the local machine's CPU and memory status every 1000ms. The status is displayed in the mGrid Engine graphical user interface. Furthermore, it periodically sends heartbeat messages in the form of *EngineInfoEntry* containing the machine's latest status to a remote mGrid Toolkit software for monitoring purposes. All messages generated by the Machine Monitor goes through the engine kernel, the kernel passes the message to the distributed space, where it is read by mGrid Toolkit for display. Machine status includes CPU and memory usage level.

See section 3.4.1 for *EngineInfoEntry* details.

3.3.5 Engine Command Listener

Engine Command Listener waits for commands sent from the mGrid Toolkit, such as shutting down an engine or setting processing condition thresholds. Command messages sent from mGrid Toolkit first go through the engine kernel, which in turn passes them into the distributed space service on the mGrid Platform, where they are fetched by the Engine Command Listener component for analysis and execution. Command messages are concealed in an *EngineCommandEntry* class. See section 3.4.2 for further information regarding *EngineCommandEntry*.

At this point, we have completed the technical introduction of mGrid Engine. However, engines are by themselves a full-fledged software with simple-to-use graphical user interfaces. Appendix has a thorough user tutorial on mGrid Engine.

Last but not least, we discuss mGrid Toolkit in the next section. mGrid Toolkit is an utility tool that simplifies the administration of any mGrid network environment.

3.4. mGrid TOOLKIT

mGrid Framework provided a simple-to-use utility tool called *mGrid Toolkit*, which enables easy administrating and monitoring of existing mGrid environments.

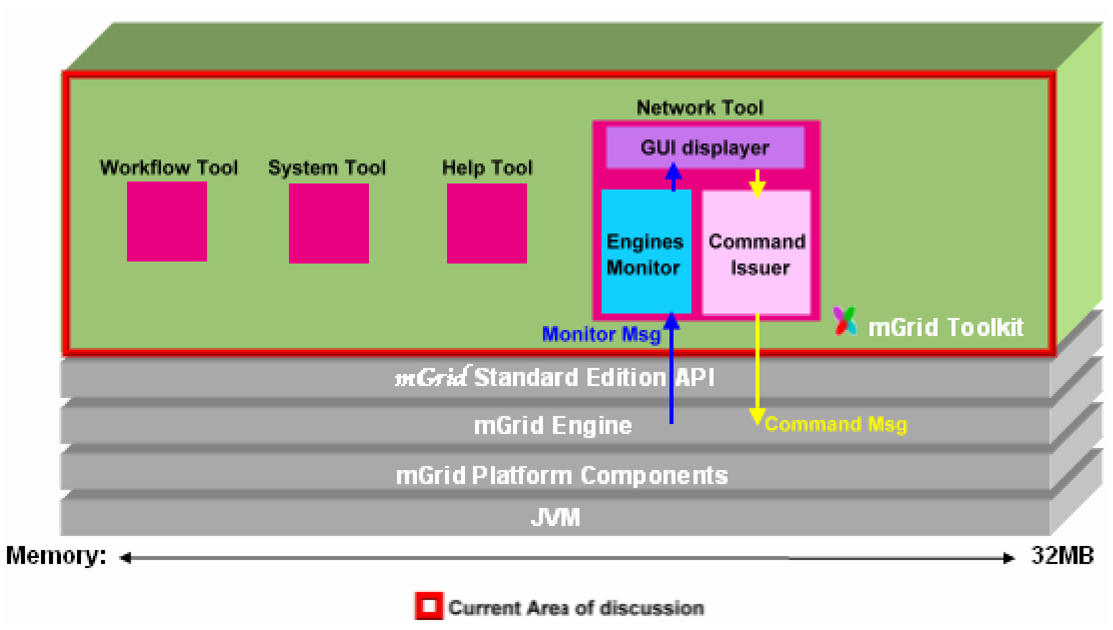


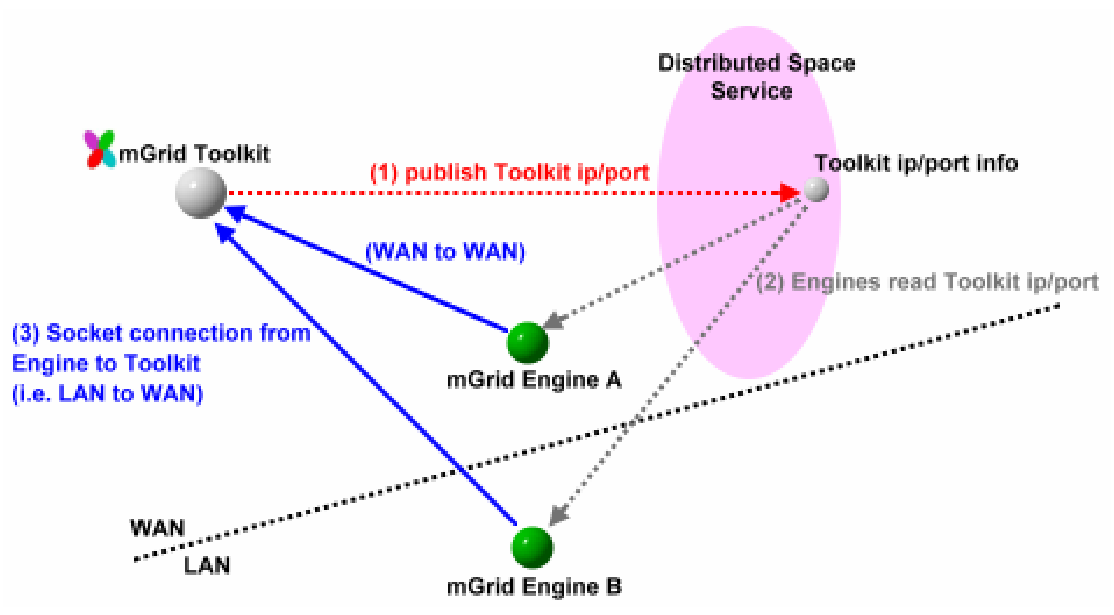
Figure 3-11: In-depth component view of the mGrid Toolkit, indicated in red

Figure 3-11 illustrates the tools included in mGrid Toolkit, namely Network tool, Workflow tool, System tool and Help tool. At this stage we have fully-implemented the Network tool, thus it will be the focus of this section.

But before we start to introduce the components within the Network Tool, we need to understand how mGrid Toolkit initially connects with mGrid Engines. Keep in mind that engines might randomly spread across WAN and different LANs. There are two possible situations:

Scenario 1. The toolkit resides in public network (WAN).

Scenario 2. The toolkit resides in private network (LAN).



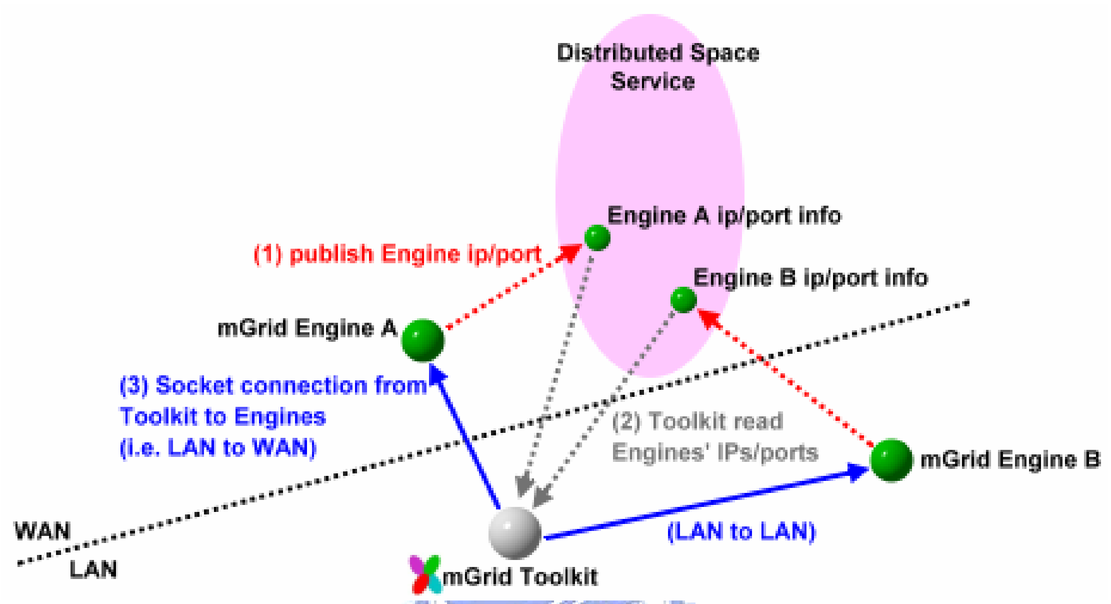
Distributed Space Service
mGrid Toolkit
(1) publish Toolkit ip/port
Toolkit ip/port info
(WAN to WAN)
(2) Engines read Toolkit ip/port
(3) Socket connection from Engine to Toolkit (i.e. LAN to WAN)
mGrid Engine A
WAN
LAN
mGrid Engine B

Figure 3-12: mGrid Toolkit and Engine initial linking steps - When the Toolkit resides in WAN

Figure 3-12 illustrates the first scenario. In order for a Toolkit residing in public network to communicate with Engines (which might reside in both public or private networks), the Toolkit needs to publish its location information, namely IP/port, to the space service of the mGrid Platform. Engines A and B *read* this location information object and actively establish a socket connection TO the toolkit using the IP/port pair.

Toolkits in this scenario waits passively for the socket connection.

However, if the toolkit resides in a private network, as stated in scenario two, the algorithm needs to be slightly modified. Figure 3-13 depicts this situation:



Distributed Space Service
Engine A ip/port info
Engine B ip/port info
(1) publish Engine ip/port
mGrid Engine A
(3) Socket connection from Toolkit to Engines (i.e. LAN to WAN)
(2) Toolkit read Engines' IPs/ports
mGrid Engine B
(LAN to LAN)
WAN
LAN
mGrid Toolkit

Figure 3-13: When the Toolkit resides in LAN, the algorithm needs to be slightly modified

In Figure 3-13, the toolkit resides in a private network. If we naïvely apply the algorithm steps given in Figure 3-12, initial socket connections between the toolkit and engines A & B will fail, for in this scenario the toolkit's IP address is a private one, socket connections simply cannot be made to a private IP address!

Instead, engines A and B must now publish their location information (IP/port) to the distributed space service running on mGrid Platform. The toolkit *read* engine A and B's IP/port objects from the space, and then actively establishes socket connections consecutively TO mGrid Engines A and B using corresponding IP/port pairs. Engines in this scenario waits passively for the incoming connection.

Combining the algorithms described in Figures 3-12 and 3-13, mGrid Toolkit is endowed with the capability to monitor and control thousands of mGrid Engines spanning across perplex LAN and WAN architectures concealed in today's internet.

Once the socket connections were successfully built, the Network tool can then be applied to monitor and control the engines simultaneously.

Now we introduce each of the components within the Network tool (consult Figure 3-11 for the component view).

3.4.1 Engines Monitor

In sections 3.3.3 and 3.3.4 we mentioned that an engine's JVM profiles and machine status can be monitored by the mGrid Toolkit, this is achieved with the Engines Monitor component contained in Network Tool. Periodically an engine serializes an *EngineInfoEntry* java object across the network, using the socket established previously with the toolkit.

The *EngineInfoEntry* class contains information such as CPU status, memory status, JVM version and so forth.

On the other side, the Engines Monitor de-serializes this *EngineInfoEntry* through an ObjectInputStream class provided with J2SDK5.0[4]. It then unwraps the *EngineInfoEntry* object and analyze the information concealed within.

3.4.2 Command Issuer

The network tool uses the Command Issuer component to transmit control commands to any specific engine. The command Issuer utilizes the ObjectOutputStream class in J2SDK5.0[4] to serialize an *EngineCommandEntry* object across the socket connection established previously to an engine running on the network.

The *EngineCommandEntry* class contains the command for the engine to execute, such as shutting down an engine or setting a processing threshold for an engine.

On the other side of the network, an engine de-serializes the *EngineCommandEntry*, unwraps the object and executes the command concealed within.

3.4.3 GUI Displayer

The GUI Displayer has one single purpose: create a graphical user interface to display the real-time engine status contained in *EngineInfoEntry* (section 3.4.1) and to allow administrators to send *EngineCommandEntry* (section 3.4.2) to manipulate the behavior of remote engine instances.

At this point, we have completed the technical introduction of mGrid Toolkit. However, mGrid Toolkit by itself is a full-fledged software with intuitive graphical user interfaces. See the appendix for mGrid Toolkit user's tutorial.

In summary, mGrid Toolkit contains several useful tools such as the Network tool. Network tool allows easy administration of the entire mGrid environment spanning across different LANs and WAN.

3.5. System Non-Functional Issues: Scalability and Flexibility

In section 1.4 we mentioned that apart from performance, a good grid system should be made highly-scalable and highly-flexible. In this section let us examine whether these characteristics exists for mGrid. mGrid Platform is implemented using the space-oriented architecture (see section 2.4.3), this has three advantages:

- Natural Load Balancing: mGrid Engines *fetch* tasks from a space for processing, only when it completed the current task, will it fetch another task. This implies that devices with more CPU resources can process more tasks during a fixed period of time, compared to devices with less CPU resources (see section 3.3: mGrid Engine).

- Dynamic Grid Expansion: Engines can search for a space dynamically upon start, without any human intervention. Thus adding new devices to a mGrid network is extremely convenient. Engines can also withdraw from a mGrid network freely (see section 3.3: mGrid Engine).
- No Single-Point-of-failure: a logical Space is actually physically distributed on multiple machines (see section 3.4.2: Distributed Space service), thus even when a few space service fails, the mGrid platform still remains operational.

The above three points endowed mGrid Framework with a high level of scalability. As for Flexibility, the Generic Task Processor (see section 3.3.2: Generic Task Processor) uses java's dynamic class loading capabilities which allows the task logic to be loaded into the engine on the fly for processing. An engine does not need any previous knowledge of task logics. This enables developers to write a variety of innovative applications using mGrid APIs.

3.6. SUMMARY

mGrid Framework is consisted of four portions: mGrid Platform, mGrid Engine, mGrid Toolkit and mGrid API. In this chapter we talked about the previous three. Both the engine and toolkit relies on the grid infrastructure created by the platform. All participants with an engine activated is a legal entity within a *mGrid environment*, where a variety of resources can be shared among each other. Finally, the mGrid toolkit simplifies the administration of *mGrid environments*. See appendix for a complete user's tutorial on mGrid Engine and Toolkit.

4. PROGRAMMING INTERFACE DESIGN

4.1. FOREWORD

The mGrid Framework is consisted of four portions: mGrid Platform, mGrid Engine, mGrid Toolkit and mGrid API. In chapter 3 we introduced the previous three. In this chapter we will probe into the last item: *mGrid API*. Figure 4-1 illustrates a high-level view of the entire mGrid API library:

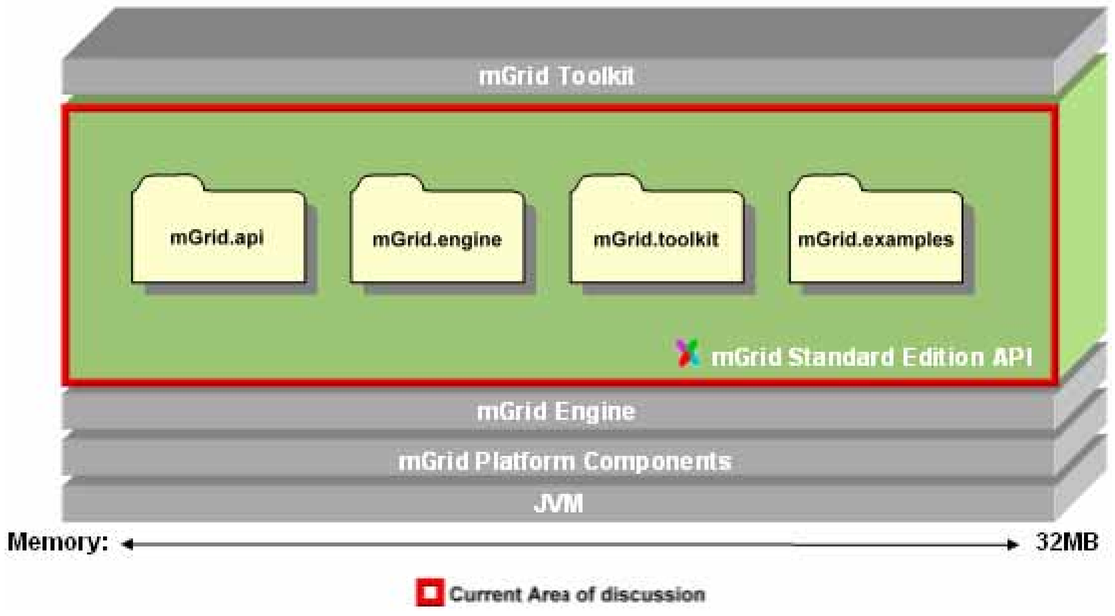


Figure 4-1: High-level view of the entire mGrid API library, indicated in red

Developers utilizes the libraries provided by mGrid API to write various creative grid applications that makes use of the resources harnessed within a *mGrid environment* (see section 3.3: mGrid Engine).

Currently the mGrid API is divided into four sub-packages:

- **mGrid.api:** the core of mGrid API. Classes in this package enable developers to discover space services (see section 3.4.2: Distributed Space service), to build different categories of task entries and to create a diverse set of engine commands.
- **mGrid.engine:** an implementation of mGrid Engine (consult section 3.3: mGrid Engine). Multiple engine-related utility classes are also included.
- **mGrid.toolkit:** contains an implementation of mGrid Toolkit (consult section 3.4: mGrid Toolkit) and a set of toolkit-related utility classes.
- **mGrid.examples:** presently includes a 3D graphics demo program written with *mGrid.api* that utilizes the CPU resources harnessed in a mGrid Environment.

The packages will be introduced respectively in subsequent sections. We first begin with the *mGrid.api* package.

4.2. mGrid.api PACKAGE

The *mGrid.api* package is the core of mGrid API. Our development objective is to make it simple to use and easy to expand.

Simplicity is our primary concern.

The package has three major categories of classes:

1. *SpaceAccessor* class: enable grid application developers to access platform functions such as searching and using a distributed space service (consult section 3.4.2: Distributed Space service).

2. *TaskEntry*, *ResultEntry*, *Command* classes: enable developers to define the logics of grid task chunks to be submitted to mGrid Engines for processing.

3. *EngineInfoEntry*, *EngineCommandEntry* classes: Developers extends these two classes to add more engine monitoring attributes and to build an extended set of engine commands respectively.

Figure 4-2 shows the UML class diagram of mGrid.api package:

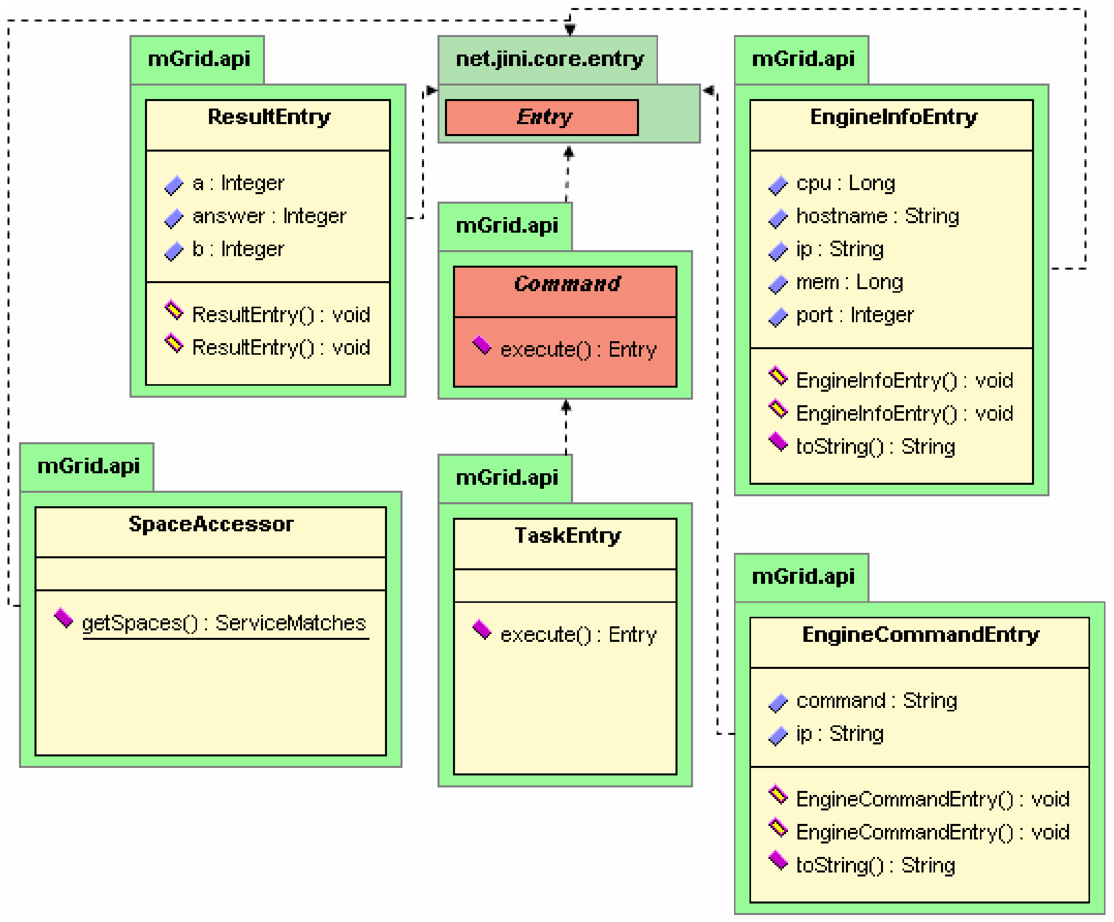


Figure 4-2: UML Class diagram of mGrid.api. Simplicity is our primary objective

SpaceAccessor class has one simple function getSpaces() which returns a distributed space service for the developers to operate on. Furthermore, you should sub-class *TaskEntry* and implement your computation logic in its execute() method. We refer to the object generated by the class which extends *TaskEntry* simply as "a task".

A task object is passed into a space, where it is fetched by a remote mGrid Engine.

Once an engine obtains a task object, it dynamically loads the logic code within the task's execute() method and start processing (consult section 3.3.2: Generic Task Processor). A class sub-classing *ResultEntry* is returned to the space after processing completes, within contains the computation results.

Listing 4-1 is an example code of utilizing the *SpaceAccessor*, *TaskEntry* and *ResultEntry* classes:

```
import mGrid.api.*;
//...

Space space = SpaceAccessor.getSpaces();

TaskEntry task = new MyTask();
space.put(task);

//wait for processing to complete
ResultEntry template = new ResultEntry();
ResultEntry result = space.read(template);

System.out.println(result.toString());
```

Listing 4-1: A simple example of utilizing SpaceAccessor, TaskEntry and ResultEntry

When you obtained a space reference using the *SpaceAccessor*, you can call the *put()* method on the space object to put a task in a space, call *read()* and *fetch()* methods to read and fetch an entry from space respectively. Note that for *read()* and *fetch()* methods you will need to specify a template first. In listing 4-1 a template with the type *ResultEntry* is specified, this means an entry with type *ResultEntry* will be read or fetched from the space.

Keep in mind that you should implement your code logics within the execute() method that comes with the *TaskEntry* class, by extending *TaskEntry*. Listings 4-2 & 4-3 depicts example classes extending *TaskEntry* and *ResultEntry*:

```
import mGrid.api.*;
//…
public class MyTask extends TaskEntry
{
    private Space space;
    public MyTask()
    {
      space = SpaceAccessor.getSpaces();
    }

    //place your computation logic in this method!
    public Entry execute()
    {
            ResultEntry result = new MyResult();
            space.put(result);
            return result;
    }
}
```

Listing 4-2: Example class extending TaskEntry

```
import mGrid.api.*;
//…
public class MyResult extends ResultEntry
{
    public String toString()
    {
        return "I am a result!!";
    }
}
```

Listing 4-3: Example class extending ResultEntry

Finally, *EngineInfoEntry* and *EngineCommandEntry* classes allows you to access a remote engine's information and specify a command for an engine respectively.

Listing 4-2 fetches a remote mGrid Engine's information and displays it on screen:

```
import mGrid.api.*;
//…

Space space = SpaceAccessor.getSpaces();

EngineInfoEntry template = new EngineInfoEntry();
template.ip = "192.168.11.2";
EngineInfoEntry engineInfo = space.fetch(template);

//print the remote engine's information
System.out.println(engineInfo.hostname);
System.out.println(engineInfo.ip);
System.out.println(engineInfo.port);
System.out.println(engineInfo.cpu);
System.out.println(engineInfo.mem);
```

Listing 4-4: Example code that fetches a remote engine's info and displays its contents

Listing 4-2 requested the space for the *EngineInfoEntry* object of a remote engine with the IP address of 192.168.11.2. We first create a template object with the type *EngineInfoEntry* and setting its String ip field to "192.168.11.2". Next, we call the fetch() method on the space reference, passing in the template as parameter. If a matching *EngineInfoEntry* currently exists in space, it is retrieved. Finally, we can print engine information such as hostname, IP, port, CPU usage rate, memory usage rate, using engineInfo.hostname, engineInfo.ip, engineInfo.port, engineInfo.cpu and engineInfo.mem fields on the *EngineInfoEntry* you retrieved from a space respectively.

Next, we give a simple example on how to send a "shutdown" command to a remote mGrid Engine using *EngineCommandEntry* class. This is demonstrated in listing 4-3:

```
import mGrid.api.*;
//...

Space space = SpaceAccessor.getSpaces();

EngineCommandEntry engineCommand= new EngineCommandEntry();
engineCommand.ip = "192.168.11.3";
engineCommand.command = "System.exit(0)";

//send the shutdown command to the engine with IP=192.168.11.3
space.put(engineCommand);
```

Listing 4-5: Example code that sends a "shutdown" command to a remote engine

The example code in listing 4-3 executes a simple task: Shuts down a remote engine with the IP address of 192.168.11.3.

First, We create an object of *EngineCommandEntry* type and set its String ip field to "192.168.11.3", and its String command to "System.exit(0)". Next we simply put this *EngineCommandEntry* object into space. At this point, the mGrid Engine with the ip address of 192.168.11.3 should asynchronously retrieve the command object from the space, parse the String command field for the code it should execute (see section 3.3.5: Engine Command Listener). In our case, the engine shuts itself down.

We have finished a brief introduction on mGrid.api package. As you can see, by conducting a few simple calls you are empowered to monitor an entire cluster of mGrid Engines, send tasks for them to process, retrieve results, and conduct a variety of commands on any specific engine instance on the network.

Next we introduce the mGrid.engine package.

4.3. mGrid.engine PACKAGE

We provided a full-fledged mGrid Engine implementation in the *mGrid.engine* package, along with an engine-related utility class. In this section we will not list the detail class diagrams of the engine code, since this is not relevant to the development of mGrid applications. Instead, see section 3.3: mGrid Engine, for an introduction on how engines works. Here we will focus solely on the utility class: *CpuUsage* class.

Figure 4-3 illustrates the UML class diagram of *CpuUsage* class:

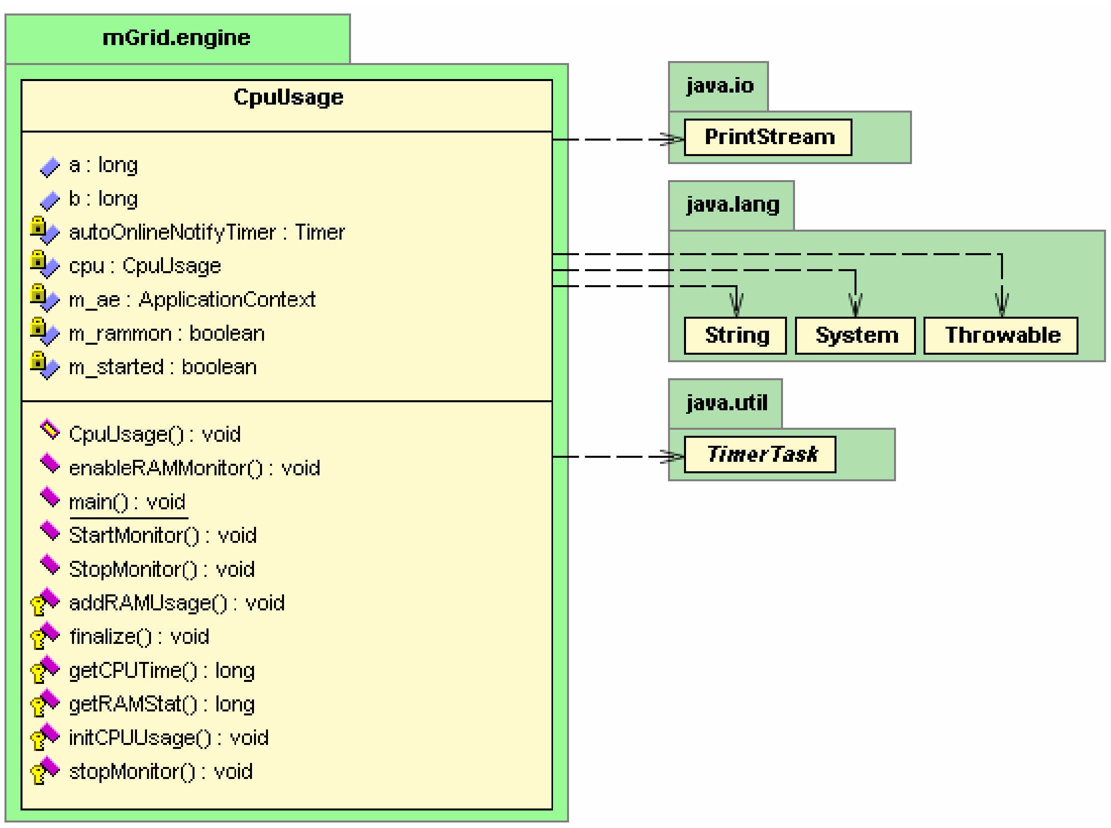


Figure 4-3: UML class diagram of CpuUsage utility class within mGrid.engine package

Application developers utilizes the *CpuUsage* class to detect the CPU and memory status of the local machine, and to send this information to the mGrid Toolkit software for remote monitoring (see section 3.4: mGrid Toolkit).

Furthermore, experienced engineers have the option of writing their own engine implementation using the methods provided in *CpuUsage* class, combined with the classes in *mGrid.api* package. Listing 4-4 shows an example code of using the *CpuUsage* class:

```
import mGrid.engine.*;
//…

CpuUsage engine = new CpuUsage();

//start sending monitoring messages to toolkit
engine.startMonitor();

System.out.println(engine.getCPUTime());
System.out.println(engine.getRAMStat());

//stop sending monitoring messages to toolkit
engine.stopMonitor();
```

Listing 4-6: an example code segment using the CpuUsage class

In listing 4-4, we first create an object of *CpuUsage* class, this initializes the monitoring thread on the local machine. Next, by calling startMonitor() method, the device's status messages are constantly sent across the network to the toolkit software. Methods getCPUTime() and getRAMStat() returns the current CPU and memory usage rate of the local machine respectively. Finally, stopMonitor() stops sending monitoring messages to the toolkit.

At this point we have introduced how to apply the *mGrid.api* and *mGrid.engine* packages to a grid application. This should be sufficient for most grid-based programs. In the following section, we dive into the *mGrid.toolkit* package.

4.4. mGrid.toolkit PACKAGE

We also supply a fully-featured mGrid Toolkit software in the *mGrid.toolkit* package. Apart from the toolkit software, another utility class called *MonitoringThread* is also at the disposal of mGrid application developers. *MonitoringThread* class contains a hashtable of all the engines currently under monitor. Figure 4-4 depicts the UML class diagram of the *MonitoringThread* class:

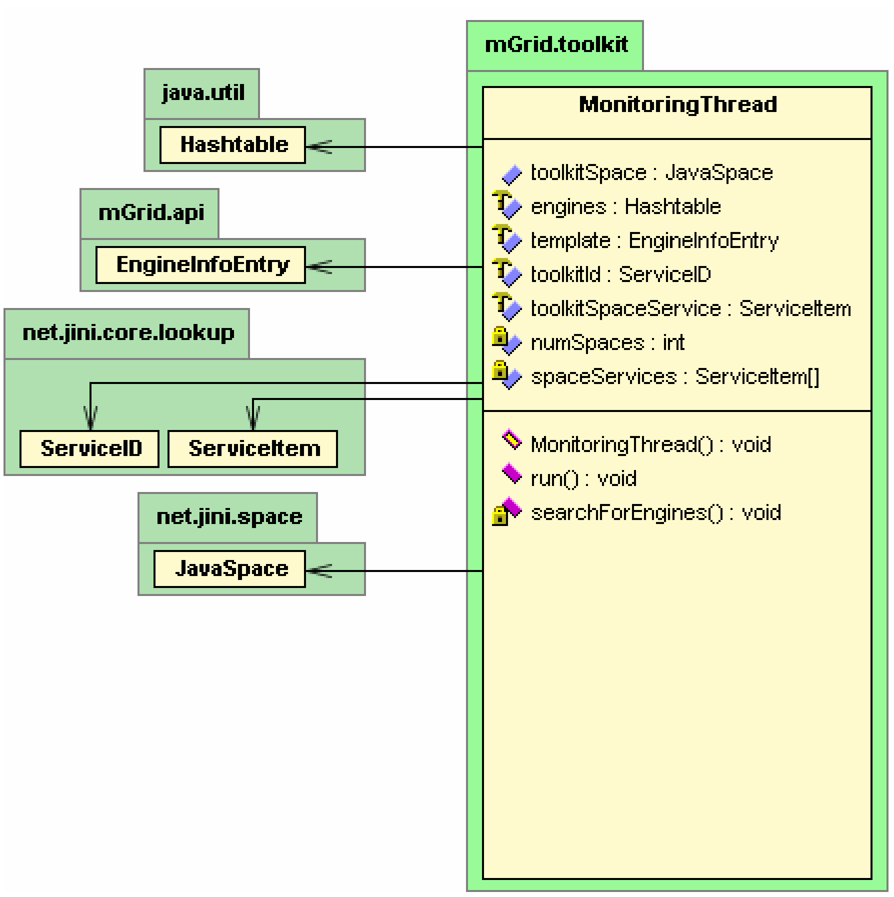


Figure 4-4: UML Class diagram of MonitoringThread utility class within mGrid.toolkit package

When you initialize a *MonitoringThread* instance, a new thread is started and runs in the background. This thread continuously looks for all the engines currently running within a mGrid environment, and populates its Hashtable engines field with remote engines' information. Hashtable engines field contains engineIP-*EngineInfoEntry* pair, by iterating through all entries in the hashtable, a developer can retrieve a set of *EngineInfoEntry* objects containing useful engine information, see listing 4-4 for uses of *EngineInfoEntry* class.

Listing 4-7 below shows an example code displaying all the mGrid Engines currently running on the network using *MonitoringThread* class:

```
import mGrid.toolkit.*;
//…

MonitoringThread toolkit = new MonitoringThread();
toolkit.start();

//hashtable containing all the engines currently alive on the network
HashTable allEngines = toolkit.engines;

//iterate through all the entries in the hashtable, listing out all remote engine
Enumeration e = allEngines.elements();
while(e.hasMoreElements())
{
      EngineInfoEntry engineInfo = (EngineInfoEntry) e.nextElement();

      //print each remote engine's information
      System.out.println(engineInfo.hostname);
      System.out.println(engineInfo.ip);
      System.out.println(engineInfo.port);
      System.out.println(engineInfo.cpu);
      System.out.println(engineInfo.mem);
}
```

Listing 4-7: Example code that uses MonitoringThread class to display all engines on network

Note that only a machine that has a mGrid Engine software running, or have called the startMonitor() method on the *CpuUsage* class (see section 4.3), can it be a legal candidate for detection by the *MonitoringThread* class.

With *mGrid.toolkit*, *mGrid.api* and *mGrid.engine* packages in hand, a programmer can now fully-utilize the strength of parallel computing provided by the mGrid Framework.

In the last package, namely *mGrid.examples*, we give a demo application written with the previous three API packages.

4.5. mGrid.examples PACKAGE

Developers can consult the code in this package to further assist them on how to write real-world mGrid applications. *mGrid.examples* package contains a demo program which utilizes the classes in *mGrid.api*, *mGrid.engine* and *mGrid.toolkit* packages.

The demo program consisted of a window running 3D graphics rotation. However, instead of conducting the 3D calculation on the local machine, the computation is divided into multiple independent task chunks that can be passed into a mGrid environment for a cluster of mGrid Engines to process in a parallel fashion. This means that when you only have a single mGrid Engine running on your network, the rotating speed of the 3D graphics is minimal, but as you start more engine instances on other machines, it immediately accelerates!

See appendix section on setting up a mGrid Environment and running this demo.

4.6. RECOMMENDED mGrid APPLICATIONS

mGrid API enables programmers to develop numerous innovative application. But which sort of application can fully-utilize the power of our framework?

Here, we recommend four categories of appropriate applications:

- Category 1: All returned results needs to be pieced together in order to produce a final result, and each chunk of task produces a single result. Image-processing applications such as ray-tracing fall into this category.
- Category 2: Returned results are independent of one another, but each task chunk produces its own result. Statistical analysis applications such as customer behavior analysis programs fall into this category. Each computer analyzes its own customer behavior.
- Category 3: Some, but not all task chunks produce results. Searching engines fall into this category. Search tasks are dispatched onto multiple computers, while only a few will return the search result which fits your searching criteria.
- Category 4: Only a single task will give you the correct result. Programs written to break encrypted messages falls into this category. Multiple tasks will be issued to the grid, while only a single one will return the correct decrypted message.

Note that not all applications are suitable for a grid network. All four categories of applications we give above share a similar characteristic: parallelism. Keep in mind that only parallel programs can fully appreciate the power of mGrid Framework.

4.7. SUMMARY

In this chapter we introduced the entire mGrid API. Simplicity is our main objective while designing the programming interfaces. In sections 4.2 to 4.5, we use numerous easy-to-understand code segments to teach how to integrate mGrid API into your own application. Finally, we recommend four primary categories of applications that is appropriate to be written with mGrid Framework. Now you should have the ability to judge whether or not to develop your program using mGrid, and how.

5. EXPERIMENT AND EVALUATION

In this chapter we experiment the performance of mGrid Framework using the Linpack benchmark[26]. Linpack benchmark is widely accepted as the de facto testing software for industrial supercomputers, such as the top500[27].

We will first briefly explain what Linpack benchmark does, then compare our results with multiprocessor machines and clusters. However, to determine the good and bad of a system, many aspects needs to be taken into consideration. These aspects include performance, scalability, utilization percentage and cost. We will look into these individual issues respectively.

5.1. LINPACK BENCHMARK

The Linpack benchmark used in our experiment randomly generates a dense 1000x1000 system with one right hand side, Ax=b.

Figure 5-1 shows an example of Ax=b. In our case, *k*=1000.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}$$

Figure 5-1: Our test case. Ax=b where k=1000

For a matrix with size *k*, there are 2/3*k*^3+O(*k*^2) floating point operations to be performed, including both additions and multiplications. The calculation is based on gaussian elimination with partial pivoting.

In Linpack benchmark, the matrix product can be split into submatrices and performed in parallel. Each submatric calculation is implemented as an independent task.

In the following two sections, we compare our benchmark results against multiprocessor machines and clusters respectively.

5.2. Comparison with Multi-Processor Machines

Multiprocessor machines differs from clusters in that all the CPUs resides in the same address space, and shares common physical memory. Multiprocessor machines intends to increase the overall computation speed of a system.

In this section, we compare our system with various commercial multiprocessor products.

5.2.1 Performance: Multiprocessor vs. mGrid

We designed two scenarios for mGrid:

- **Scenario one:** 1, 2, 4, 8 compute nodes. Only the first node runs the distributed space service (see section 3.2.4: Distributed Space Service).
- **Scenario two:** 1, 2, 4, 8 compute nodes. All nodes runs an instance of distributed space service.

Note that each compute node in our mGrid environment is a Pentium4 with 1700MHz CPU, 512MB RAM, running J2SDK5.0. Nodes are interconnected using 100Mbps Ethernet.

the result is shown in table 5-1[26]:

Machine	**1 CPU**	**2 CPU**	**4 CPU**	**8 CPU**	**Cost of 2 CPUs**
HP AlphaServer ES80 7/1150(1.15GHz)	1184	3424	6584	11410	*
Cray SV1ex-1-32(500 MHz)	1554	2947	5358	8938	*
HP 9000 rp8420-32	2905	5435	9478	14150	**$ 93,000**
NEC SX-4/1	1944	3570	6780	12780	**$ 52,680**
Compaq Server ES40(667MHz)	1031	1923	3804	7905	**$ 23,900**
IBM eServer pSeries 610 Model B80	1451	2521	4396	8302	**$ 31,500**
HP SuperDome	1497	2506	4319	8055	*
mGrid Scenario one	**1322**	**2640**	**5277**	**10541**	**$ 4,909**
mGrid Scenario two	**1330**	**2659**	**5316**	**10630**	**$ 4,909**

Table 5-1: Performance comparison with commercial multiprocessor machines (Mflop/s)

Figure 5-2 illustrates the line chart of table 5-1:

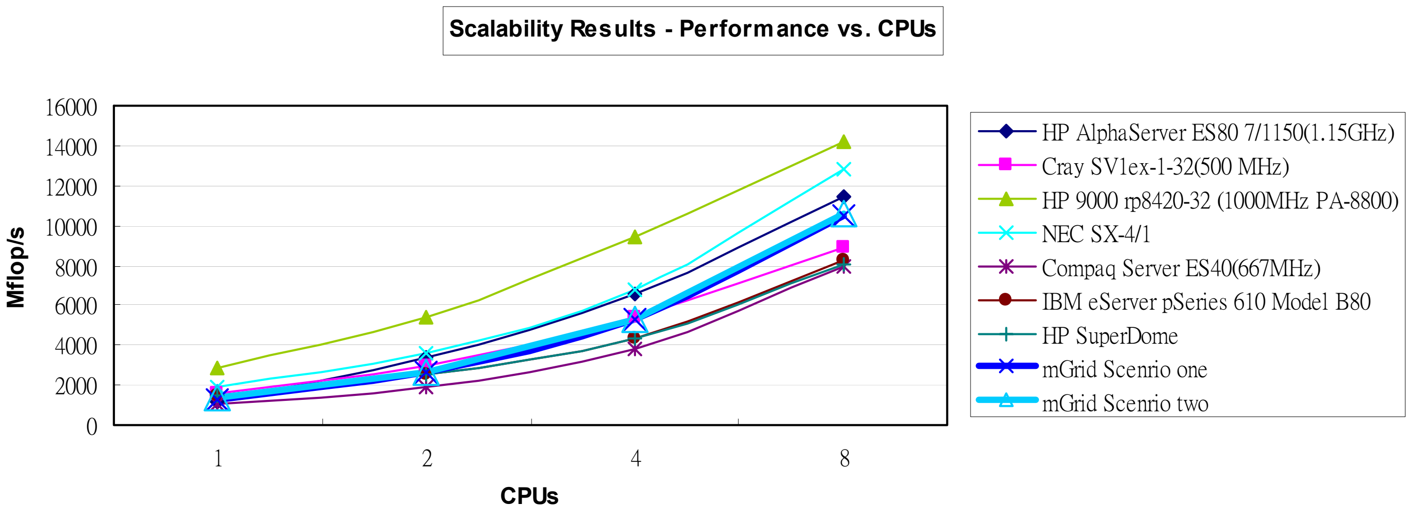


Figure 5-2: Performance comparison with multiprocessor machines, chart view

5.2.2 Utilization: Multiprocessor vs. mGrid

The processing performance is direct proportional to the speed of CPU used, thus with different systems using different CPUs, merely comparing performance is not rational.

Instead we need to focus on the utilization of each CPU. For instance, the best performed system: the HP 9000 rp8420-32 with 1 CPU can reach 2905 Mflop/s, theoretically with 2 CPUs it should reach 5810 Mflop/s, yet in reality it only reached 5435 Mflop/s, this implies 6.454% of processing power is wasted. Furthermore, with 8 CPUs, it should theoretically reach 23240 Mflop/s, in reality it only achieved 14150 Mflop/s, it implies a 39.113% loss of computation capability!

On the contrary, with mGrid Framework, less than 0.3% of CPU power is left idle. This result shows that mGrid Framework can fully utilize your compute nodes, while multiprocessor machines will waste more computation resources as you add more CPUs to the system.

The under-utilization situation of each system is shown in Table 5-2:

Machine	**1 CPU**	**2 CPU**	**4 CPU**	**8 CPU**
Cray SV1ex-1-32(500 MHz)		5.1801802	13.803089	28.104891
HP 9000 rp8420-32 (1000MHz PA-8800)		6.454389	18.433735	39.113597
NEC SX-4/1		8.1790123	12.808642	17.824074
Compaq Server ES40(667MHz)		6.7410281	7.7594568	4.1585839
IBM eServer pSeries 610 Model B80		13.128877	24.259132	28.480358
HP SuperDome		16.299265	27.872411	32.740481
mGrid Scenario one		**0.1512859**	**0.2080182**	**0.330938**
mGrid Scenario two		**0.037594**	**0.075188**	**0.093985**

Table 5-2: CPU Resource wasted (%)

From Table5-2, we can conclude that (average approximation):

Commercial multiprocessor products	**1 CPU + 1 CPU ≈ 1.512 CPUs**
mGrid Framework	**1 CPU + 1 CPU ≈ 1.985 CPUs**

Table 5-3: CPU Resource utilization comparison between multiprocessor products and mGrid

Table 5-3 is an approximate calculation. At this point we have shown that mGrid Framework can utilize a set of dispersed computing resources much better than multiprocessor machines.

5.2.3 Scalability: Multiprocessor vs. mGrid

Now let's see the price it takes for systems to scale. For multiprocessor machines, if the CPU quantity exceeds the maximum number a single machine can contain, this often means one thing: to scale further, purchasing a second machine is inevitable. Let's say you have a multiprocessor machine with a maximum of *n* CPU slots, if the processing power of *n+1* CPUs is required, you will have to purchase a whole new machine pre-installed with 1 CPU, thus leaving *n-1* slots empty. This is simply not cost-effective. On the other hand, scaling up is simple for mGrid. We scale the size of a mGrid Environment in table 5-4:

Machine	**8 CPU**	**9 CPU**	**10 CPU**	**11 CPU**
mGrid Scenario one	**10541**	**11860**	**13181**	**14199**
mGrid Scenario two	**10630**	**11955**	**13300**	**14632**

Table 5-4: mGrid System scales easily (Mflop/s)

As we mentioned in chapter 1, scalability is one of our primary concerns. Here we show that adding more CPUs to a mGrid environment is much more simpler and cost-effective than adding more CPUs to multiprocessor machines.

5.2.4 More on Performance: The number of Spaces matters

Before we end this sub-section, one more interesting effect deserves to be discussed. In mGrid scenario one, we run a single space service on the first machine, while in scenario two, all machines runs an instance of space service. The performance of scenario two is clearly better than that of scenario one. See tables 5-1 & 5-4. The reason is as follows: If we have a mGrid environment with 2 compute nodes, and these two nodes both runs an instance of space service, there is an approximate 50% chance that the task-to-be-processed resides in the same machine as the mGrid Engine. Thus decreases the propagation time of fetching a task from a remote space.

We can assume that for most cases, the performance outcome is direct proportional to the number of space services activated.

5.3. Comparison with Clusters

A cluster is a commonly found computing environment that connects multiple independent workstations residing on the same LAN(Local Area Network). Each workstation is referred to as a "computing node", and has its own set of CPUs and memory. A regular cluster differs from mGrid in two ways:

- Compute nodes within a cluster usually runs the same operating system, while mGrid spans across a variety of operating systems.
- Cluster compute nodes often resides within the same LAN, while mGrid spans across multiple LANs and WANs.

Here we compare our system with multiple eminent cluster computers. Our goal here is to show that our system brings lower TCO(Total Cost of Ownership) and better scalability than the commercial clusters, offering the same computing capability.

5.3.1 Performance: Clusters vs. mGrid

First we look at the Linpack Performance of the following commercial clusters, as shown in table 5-5:

Machine	Num of Nodes	Gflop/s
Sun HPC 6500(400MHz 8MB L2 Cache)	18	13.05
CRAY T3E-1200E (600 MHz)	16	13.41
SGI Origin 2000 (250 MHz)	32	13.22
Intel Paragon XPS-35 (50 MHz, OS=R1.1)	512	15.2
Compaq GS140 cluster	24	15.31
mGrid Scenario one	**11**	**13.87**
mGrid Scenario two	**11**	**14.29**

Table 5-5: Performance comparison with commercial clusters

Cluster architecture, unlike multiprocessor machines, can harness all the available CPU resources just as well as mGrid does. Thus our assumption here is that all the compute nodes within a cluster (and within mGrid environment) is close to 99-100% utilized. The performance is shown in table 5-5. However, we do not intend to compare the absolute performance, since each cluster has very dissimilar hardware and is interconnected using different network technologies. Instead, we will compare two things. First, we compare the TCO (Total Cost of Ownership) of owning a mGrid environment with equal computing capability as the commercial clusters listed above. Second, we show that unlike traditional clusters, mGrid can effectively avert the SPF problem (Single-Point-of-Failure).

5.3.2 Total Cost of Ownership: Clusters vs. mGrid

As we mentioned in section 2.2, high TCO is the primary factor that prevents individuals or SMEs from adopting grid-like technology, thus our objective is aimed at offering a low-cost grid solution. Table 5-6 depicts the costs of various commercial cluster computers[28].

Machine	TCO (million $USD)	Gflop/s
Sun HPC 6500(400MHz 8MB L2 Cache)	0.3	13.05
CRAY T3E-1200E (600 MHz)	0.14	13.41
SGI Origin 2000 (250 MHz)	0.85	13.22
Intel Paragon XPS-35 (50 MHz, OS=R1.1)	1.92	15.2
Compaq GS140 cluster	1.64	15.31
mGrid Scenario one	**0.027**	**13.87**
mGrid Scenario two	**0.027**	**14.29**

Table 5-6: TCO(Total Cost of Ownership) comparison of mGrid with commercial clusters[28]

A mGrid environment with 11 Pentium4 1700MHz costs approximately $0.027mn USD[20], while the TCO of other commercial clusters with similar computation power ranges from $0.3mn ~ $1.92mn USD. This implies that on average a commercial cluster with more or less the same computing power is 36 times more expensive than our mGrid solution.

This is further illustrated in Figure 5-3:

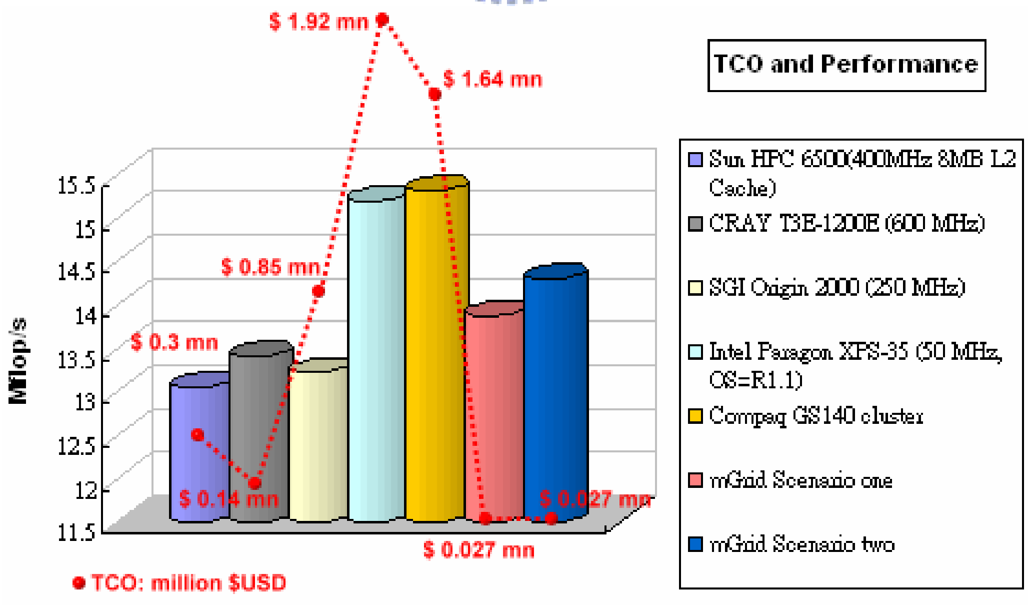


Figure 5-3: Performance & TCO comparison with clusters, mGrid is much cost-effective!

We have shown that mGrid is a cheap and effective shortcut to grid computing. With a fairly reasonable TCO, mGrid proves itself to be an excellent entry point to grid computing for individuals and SMEs.

5.3.3 Single-Point-of-Failure Issue: Clusters vs. mGrid

Now that we proved mGrid is economically more cost-effective than many commercial clusters, we turn our focus back to the technical aspect once more. Traditional clusters suffers from the severe Single-Point-of-Failure (SPF) problem, by single-point-of-failure we mean that when the dispatch server of a cluster fails, the entire backend cluster is immediately rendered useless (consult section 2.4.1: Client-Server Architecture). mGrid, on the other hand, do not have the SPF problem. As long as at least a single distributed space service is alive, the mGrid environment can continue on processing jobs, table 5-7 shows the situation when mGrid scenario two suffers from space failure of different levels:

Machine	**0 fail**	**1 fail**	**2 fail**	**3 fail**
mGrid with 11 compute nodes	14632	14520	14320	14199

Table 5-7: Situation considering the number of spaces failed (Mflop/s)

mGrid can tolerate with space failures, yet failures do bring performance downgrade of acceptable level.

5.4. DISCUSSION

mGrid is a low-cost grid solution that most SMEs can afford. Comparing with multiprocessor machines, it has greater scalability and allows better utilization of CPU resources. Furthermore, it averted the SPF problem that most traditional cluster commercial products suffer. A concluding comparison is shown below, we normalized all of our experiment data into a rating scale ranging from 0 to 1. A 1 shows the best rating, while a 0 indicates the poorest rating :

Name	**Multiprocessor**	**mGrid Framework**	**Cluster/Grid**
Linpack Performance	0.965	1.000	0.997
Linpack Utilization	0.731	1.000	1.000
TCO	0.009	1.000	0.027
Scalability	0.003	1.000	0.857
SPF	1.000	1.000	0.000

Figure 5-4: Overall comparison between mGrid, Multiprocessor & grids/clusters

In this final section, we will discuss the mGrid mathematically. The question is this: Suppose we create a mGrid Environment using the following four computers[26], what do we get?

Node	**CPU**
1	Pentium4 1700MHz, 1330Mflop/s
2	Intel/HP Itanium 800MHz, 580 Mflop/s
3	AMD Opteron 1200MHz, 443 Mflop/s
4	AMD Athlon 1530 MHz, 832 Mflop/s

Table 5-8: Suppose we combine four CPUs together into a mGrid virtual grid, what do we get?

First we calculate how many operations per cycle each processor does. For node one:

$$\frac{1330\text{ Mflop}}{1\text{ sec}} \times \frac{\frac{1}{1700\text{ M}}\text{ sec}}{1\text{ cycle}} = \frac{0.764\text{ operations}}{1\text{ cycle}}$$

using similar method we obtain 0.725 (op/cycle) for node two, 0.369 (op/cycle) for node three and 0.544 (op/cycle) for node four. Since we are conducting parallel processing, this means for the combined mGrid environment, there is a total of 0.764+0.725+0.369+0.544=2.402 operations per cycle. According to our benchmark, the overall floating point calculation of mGrid environment with these four nodes is 3160 Mflop/s. Thus:

$$\frac{\text{3163 Mflop}}{\text{1 sec}} \times \frac{\text{1 cycle}}{\text{2.402 operations}} = \frac{\text{1316819317 cycle}}{\text{1 sec}} = \text{1316 MHz}$$

we see the overall frequency of the resulting mGrid environment is approximately 1316MHz. Our conclusion is, by adding more computers to a mGrid environment, the operations per cycle the system can do grows, yet the frequency will be each of the processors' average.

In other words, the frequency of a mGrid environment does not necessarily reflect the performance. Even though you have a lower frequency, you still can achieve better performance in parallel computing.

6. CONCLUSION, BUSINESS OPPORTUNITIES & FUTURE WORKS

6.1. CONCLUSION & BUSINESS OPPORTUNITIES

There are various options in solving problems that requires supercomputing, yet the high TCO (Total Cost of Ownership) of supercomputing intimidated the SMEs from adopting such technology. mGrid framework proves to be a low-cost, pure software-based grid computing solution that can reduce the entry barrier of obtaining a grid infrastructure.

Furthermore, this thesis also demonstrated the advantage of utilizing the asynchronous, space-oriented architecture. mGrid showed reasonable performance, superior utilization, greater scalability, higher flexibility than many commercial supercomputing products such as multiprocessor machines and cluster computers. Our architectural design also avoided common technical flaws found in grid products, such as the Single-Point-of-Failure problem.

Last but not least, the framework also provides utility tools and a set of Application Programming Interfaces(APIs) that simplifies the process of grid application development, thus optimizes overall productivity. Developers must focus on design and development rather than hunting for resources hidden within the enterprise.

Even though mGrid framework is designed primarily for scientific computing, its high flexibility enables it to be used in various innovative areas such as digital home entertainment. By deploying mGrid Engines on java-enabled platforms such as cellular phones, STBs (Set-Top Boxes), and other multimedia devices, mGrid can quickly harness all the multimedia resources hidden within each device, and allows resource sharing among a network of devices. This concept is depicted in Figure 6-1:

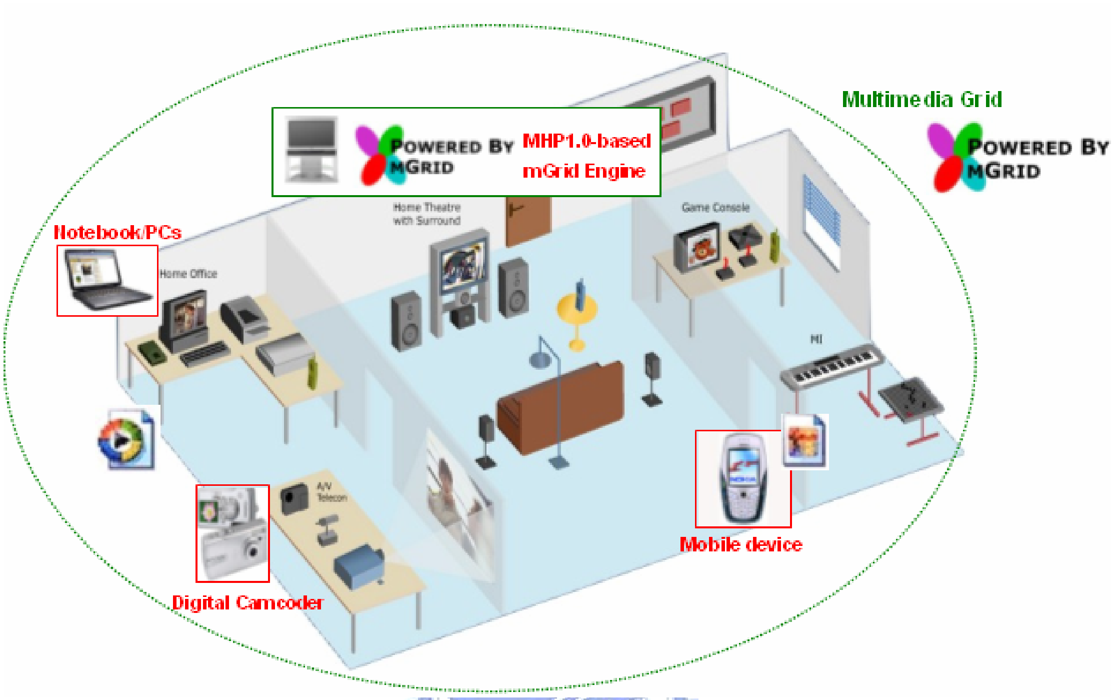


Figure 6-1: Business opportunity: mGrid can be seen as the middleware for home networks

IDC predicted a continuous 20% growth in digital home market, reaching a hundred billion USD worth of revenue by the year of 2010. With proper marketing strategy, mGrid has the opportunity to play the role of "ammunition supplier" in this future war of digital home entertainment.

6.2. FUTURE WORKS

There are multiple improvements available for mGrid Framework. First, the functionality of the mGrid Toolkit can be further extended. At this point only the network tool is operational, other tools such as designing tool, which enables application developers to compose simple grid programs by means of graphical user interface can be added.

Many additional functions can be added to mGrid Engines as well, such as remotely setting the threshold of each engine, allowing engine to process tasks only if its CPU/memory usage rate is under that threshold.

Finally, the mGrid Platform itself can be improved. More services must be added into the platform if it is to be made commercial.

REFERENCES

[1] Chris Kwak and Robert Fagin, "Internet Infrastructure & Services", Bear, Stearns & Co., May 2001.

[2] Sun Microsystems, "SUN Grid Overview", http://www.sun.com/service/sungrid/overview.html.

[3] Dept. of Computer Science, University of Tennessee, "Linpack benchmark – Java version", http://www.netlib.org/benchmark/linpackjava/.

[4] Sun Microsystems, "J2SE, J2EE, J2ME", http://www.javasoft.com/.

[5] M.P. Papazoglou & D. Georgakopoulos, "Service-Oriented Computing", Communications of the ACM, Vol. 46, pp. 25-28, October 2003.

[6] Sun Microsystems, "Activation Daemon", http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/rmid.html.

[7] Sun Microsystems, "JNDI", http://java.sun.com/products/jndi/.

[8] Sun Microsystems, "SUN JINI network technology", http://www.jini.org/.

[9] Rob Bjornson and Andrew Sherman, "Grid Computing & the Linda Programming model", Dr. Dobb's Journal, Boulder, Vol.29,N.9, pp.16-17,20,22,24, Sept.2004.

[10] Reseach group of Hiroaki Isobe, Takehiro Miyagoshi, and Kazumari Shibata, Koyoto University, "NEC Earth Simulator", http://www.es.jamstec.go.jp/esc/eng/.

[11] IBM, "IBM ASCI White", http://www.llnl.gov/asci/news/white_news.html.

[12] University of California at Berkeley, "SETI@Home", http://setiathome.ssl.berkeley.edu/.

[13] Stanford University, “Folding@Home”, http://folding.stanford.edu/.

[14] GIMPS, http://www.mersenne.org/prime.htm.

[15] “Grid Computing: A Vertical Market Perspective 2005-2010”, The Insight Research Corporation, Feb 2005.

[16] Giovanni Flammia, “Peer to Peer is not for Everyone”, IEEE Intelligent systems, Vol. 16, No. 3, pp. 78-79, May/June 2001.

[17] Linda Programming Model, http://www.netlib.org/pvm3/book/node16.html.

[18] D. S. Meliksetian, J.-P. Prost, A. S. Bahl, I. Boutboul, D. P. Currier, S. Fibra, J.-Y. Girard, K. M. Kassab, J.-L. Lepesant, C. Malone, and P. Manesco, “Design and implementation of an enterprise grid”, IBM Systems Journal on Grid Computing, Vol. 43, No. 4, pp. 646-664, 2004.

[19] Daniel A. Menasce, “MOM vs. RPC: Communication Models for Distributed Applications”, IEEE Internet Computing Magazine, pp. 90-93, March/April 2005.

[20] Y.M. Teo and X.B. Wang, “ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing”, Proceedings of IFIP International Conference on Network and Parallel Computing, pp. xx, Springer-Verlag Lecture Notes in Computer Science, Wuhan, China, October 2004.

[21] Globus Alliance, “Globus Toolkit”, http://www.globus.org/.

[23] Anand Natrajan, Anh Nguyen-Tuong, Marty A. Humphrey, Andrew S. Grimshaw, “The Legion Grid Portal”, Grid Computing Environments, vol. 14, No. 13–15, pp. 1365-1394, 2002.

[24] Sun Microsystems, “Project JXTA Overview”, http://www.jxta.org/.

[25] GridSim, http://www.buyya.com/gridsim/.

[26] Jack J. Dongarra, Computer Science Department, University of Tennessee “Performance of Various Computers Using Standard Linear Equations Software”,

Technical Report CS-89-85, University of Tennessee, Computer Science Dept., The report is available electronically.
URL ftp://www.netlib.org/benchmark/performance.ps

[27] University of Mannheimtop & University of Tennessee, “Top 500 supercomputers”, http://www.top500.org/.

[28] SAIC, http://www.saic.com/supercomputing/.

APPENDIX: USER TUTORIAL

▪ Recommended Development Flow

Figure Appendix-1 shows our recommended development flow for mGrid framework:

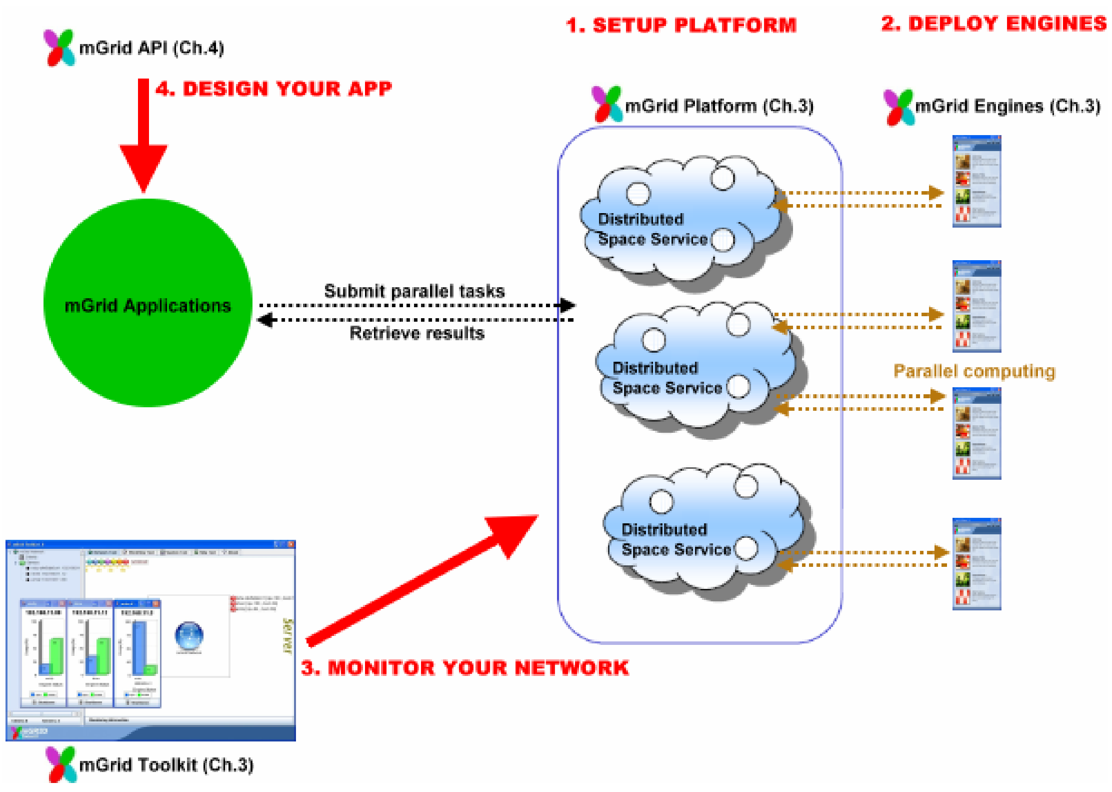


Figure Appendix-1: Recommended mGrid Framework development flow

You will need to start the mGrid Platform services first. After you have successful setup the platform, at least one mGrid Engine must be deployed on your network. At this point, you can run the mGrid Toolkit to monitor the *mGrid Environment* you have just created. Next, utilize the mGrid API to write your own applications! See the following sections for setup instructions.

▪ mGrid PLATFORM QUICK SETUP

1. Install J2SDK 1.2 or above. See reference [4] for installation instructions.

2. Copy the mGrid package to your computer. This computer must have the J2SDK pre-installed and basic network connectivity. (e.g. copy to **C:\<mGrid Package>**)

3. Start the mGrid Platform by double-clicking the following batch files consecutively: **(0)erase.bat**, **(1)http-server.bat**, **(2)activation.bat**, **(3)Nucleus.bat**, **(4)txn.bat** and **(5)space.bat**.

   These files can be found in **<mGrid Package>\bin\start**.

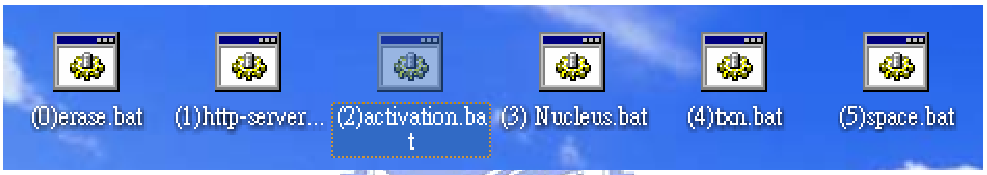


Figure Appendix-2: The batch files that starts the mGrid Platform, simple and straight-forward

4. Done! At this point the mGrid Platform is fully initiated!

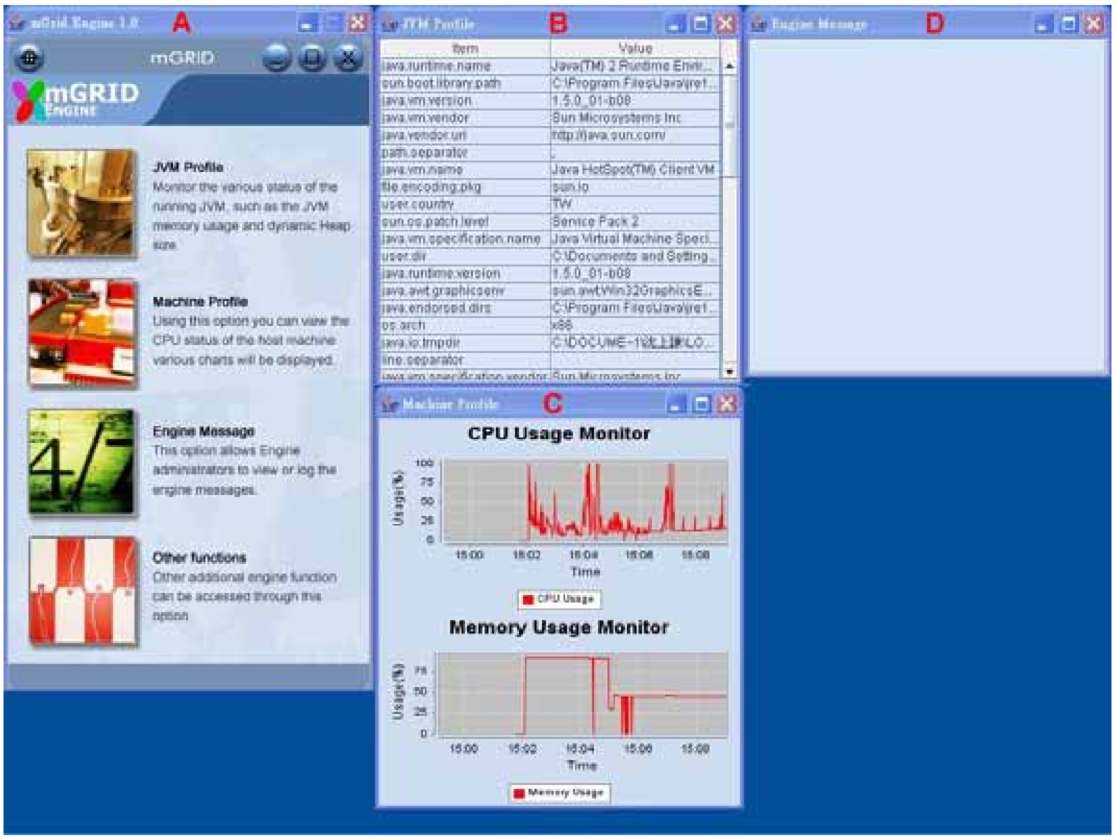
   Now that you have the mGrid Platform running, you need to start at least one mGrid Engine instance on the network to form a *mGrid Environment*. You can then use the mGrid API to write various innovative grid applications that utilizes the resources harnessed within a mGrid environment, see chapter 5 for mGrid API programmer's guide.

▪ mGrid ENGINE QUICK GUIDE

1. You need to start at least one engine instance on your computer to form a *mGrid Environment*. Double click the batch file **run_GridEngine.bat** to start the mGrid

Engine software. This file can be found in **<mGrid Package>\bin\engine**.

2. You should now see the mGrid Engine graphical user interface:



JVM Profile
Monitor the various status of the running JVM, such as the JVM memory usage and dynamic Heap size.

Machine Profile
Using this option you can view the CPU status of the host machine various charts will be displayed.

Engine Message
This option allows Engine administrators to view or log the engine messages.

Other functions
Other additional engine function can be accessed through this option

Item	Value
java.runtime.name	Java(TM) 2 Runtime Envir...
sun.boot.library.path	C:\Program Files\Java\jre1...
java.vm.version	1.5.0_01-b08
java.vm.vendor	Sun Microsystems Inc.
java.vendor.url	http://java.sun.com/
path.separator	;
java.vm.name	Java HotSpot(TM) Client VM
file.encoding.pkg	sun.io
user.country	TW
sun.os.patch.level	Service Pack 2
java.vm.specification.name	Java Virtual Machine Speci...
user.dir	C:\Documents and Setting...
java.runtime.version	1.5.0_01-b08
java.awt.graphicsenv	sun.awt.Win32GraphicsE...
java.endorsed.dirs	C:\Program Files\Java\jre1...
os.arch	x86
java.io.tmpdir	C:\DOCUME~1\[illegible]LO...
line.separator	
java.vm.specification.vendor	Sun Microsystems Inc.

Figure Appendix-3: mGrid Engine graphical user interface

The interface consists of four parts:

A. Main control panel. Call up the JVM, system usage and task message panels.

B. JVM Panel. Show the JVM profile of the local system. Information such as JVM version, OS patch level and so forth are displayed.

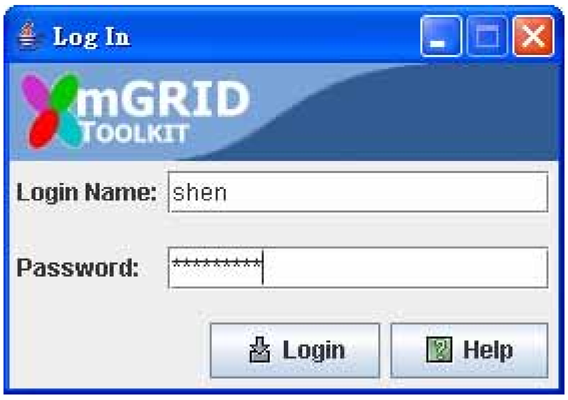
C. System Panel. Show the CPU and memory usage of the local machine. You can zoom-in or zoom-out the CPU/memory diagrams by dragging the portion you wish to inspect.

D. Task message Panel. Show the messages while processing a task. The messages show what tasks are currently being processed and whether an error occurred. it also reflects the processing speed of the engine.

▪ mGrid Toolkit Quick Guide

1. You can start a mGrid Toolkit to monitor any existing *mGrid environment*. Double click the batch file **run_GridToolkit.bat** to start the mGrid Toolkit, this file can be found in **<mGrid Package>\bin\toolkit**.

2. A login screen should appear, type in your username and password.



Login Name: shen

Password: *********

Figure Appendix-4: mGrid Toolkit login screen

3. After you successfully logged in. You should see the toolkit interface. This is depicted in Figure Appendix-4 on the following page.

   The entire mGrid network environment is now right before your eyes, and as clear as crystal! You can monitor the CPU/memory status of each device running a mGrid Engine instance, acquire their machine and JVM information. Furthermore, by clicking the shutdown button you can remotely shutdown the corresponding mGrid Engine.

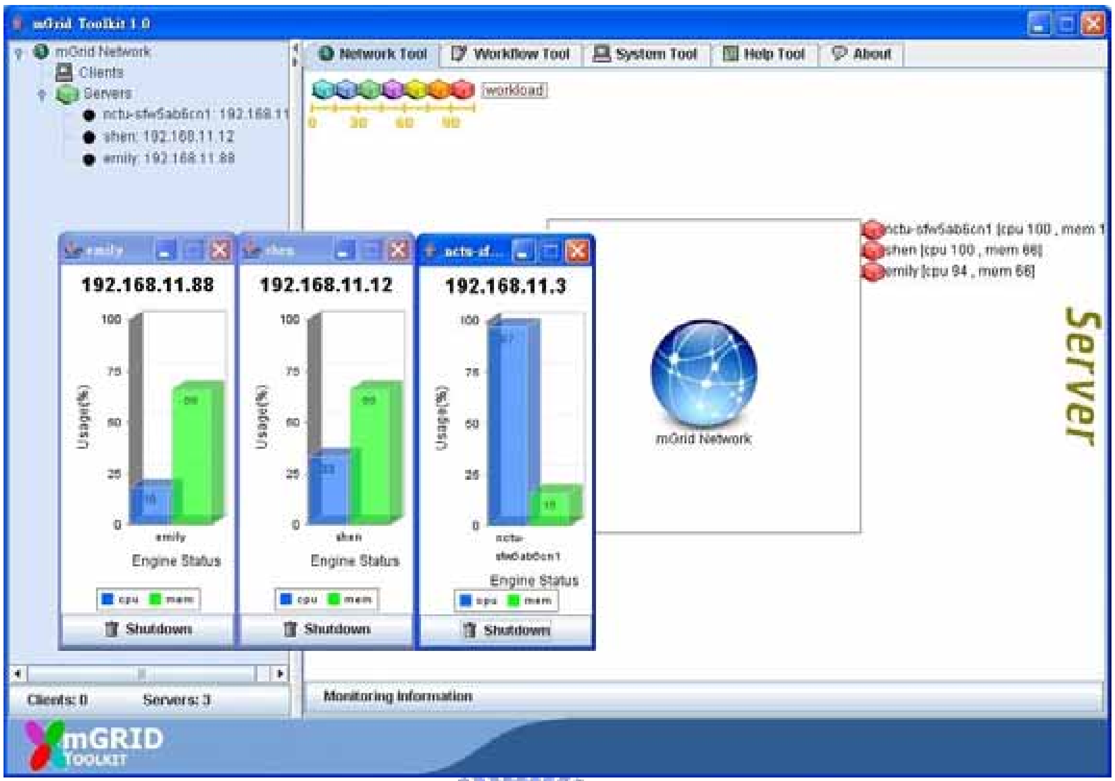


Figure Appendix-5: mGrid Toolkit main interface. Clear view of a complicated grid network!

▪ Running An Example

1. At this point you should have everything you need in place. To try out an example, double click the batch file **run_Rotator3D.bat** to start the demo program, this file can be found in **<mGrid Package>\bin\demo1**.

2. You should see a screen containing three 3-dimensional objects rotating. This is depicted in Figure Appendix-5 on the next page.

   The idea is this: the program creates a large quantity of tasks for processing. With a single mGrid Engine, the processing is extremely slow, thus the speed of rotation is also slow. However, as you start more engines on your network, the processing accelerates, thus the speed of rotation comparatively becomes faster!

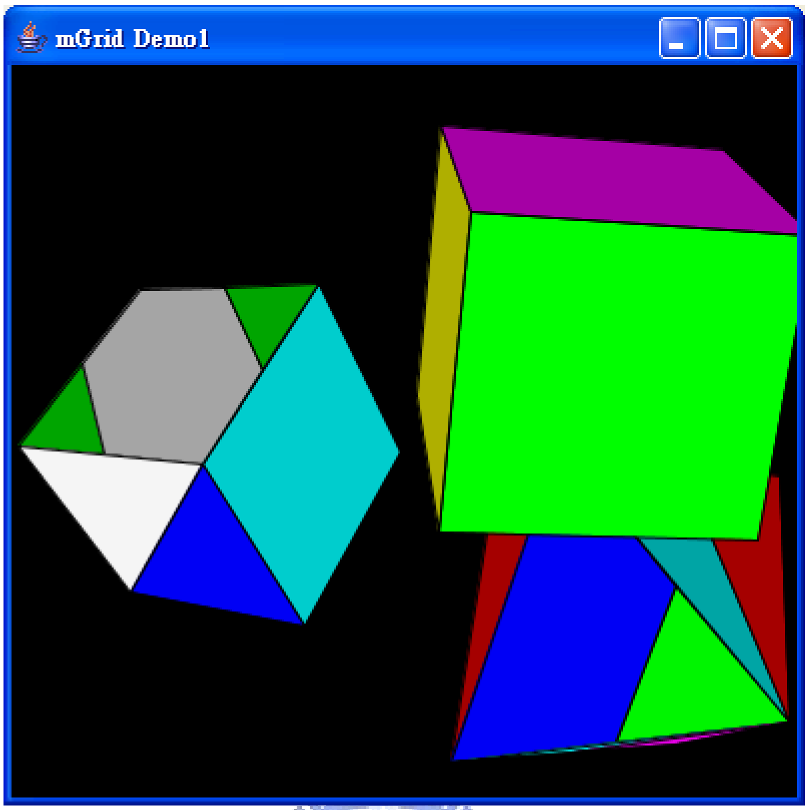


Figure Appendix-6: 3D rotating demo program. The more mGrid Engines, the faster it rotates!

3. You can refer to the demo source codes on how to effectively utilize the mGrid API to write grid programs of your own. The source code can be found in the folder **<mGrid Package>\mGrid\src\examples\ex1**.

This concludes our tutorial on mGrid Framework.