SAFRAN: A DISTRIBUTED AND PARALLEL APPLICATION DEVELOPMENT FRAMEWORK FOR NETWORKS OF HETEROGENEOUS WORKSTATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HAMZA GÖLYERİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

APRIL 2005

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe KİPER
Head Of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Müslim BOZYİĞİT
Supervisor

Examining Committee Members

Prof. Dr. Payidar GENÇ	(METU, CENG)	
Prof. Dr. Müslim BOZYİĞİT	(METU, CENG)	
Prof. Dr. Semih BİLGEN	(METU, EEE)	
Dr. Atilla ÖZGİT	(METU, CENG)	
Dr. Cevat ŞENER	(METU, CENG)	

ABSTRACT

SAFRAN:
A DISTRIBUTED AND PARALLEL APPLICATION DEVELOPMENT FRAMEWORK FOR NETWORKS OF HETEROGENEOUS WORKSTATIONS

GÖLYERİ, Hamza
M.Sc., Department of Computer Engineering
Supervisor: Prof. Dr. Müslim BOZYİĞİT

April 2005, 113 pages

With the rapid advances in high-speed network technologies and steady decrease in the cost of hardware involved, network of workstation (NOW) environments began to attract attention as competitors against special purpose, high performance parallel processing environments. NOWs attract attention as parallel and distributed computing environments because they provide high scalability in terms of computing capacity and they have much smaller cost/performance ratios with high availability. However, they are harder to program for parallel and distributed applications because of the issues involved due to their loosely coupled nature. Some of the issues to be considered are the heterogeneity in the software and hardware architectures, uncontrolled external loads, network overheads, frequently changing system characteristics like workload on processors and network links, and security of applications and hosts.

The general objective of this work is to provide the design and implementation of a Java™-based, high performance and flexible platform i.e. a framework that will facilitate development of wide range of parallel and distributed applications on networks of heterogeneous workstations (NOW). Parallel and distributed application developers are provided an infrastructure (consisting of pieces of executable software developed in Java and a Java software library) that allows them to build and run their distributed applications on their heterogeneous NOW without worrying about the issues specific to the NOW environments.

The results of the extensive set of experiments conducted have shown that Safran is quite scaleable and responds well to compute intensive parallel and distributed applications.

Keywords: Distributed and Parallel Computing, Java, Network of Heterogeneous Workstations (NOW), Parallel and Distributed Application Development Frameworks

ÖZ

SAFRAN:
HETEROJEN İŞ İSTASYONU AĞLARI İÇİN PARALEL VE DAĞITIK UYGULAMA GELİŞTİRME ALTYAPISI

GÖLYERİ, Hamza
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Danışmanı: Prof. Dr. Müslim BOZYİĞİT

Nisan 2005, 113 sayfa

Yüksek hızlı bilgisayar ağları teknolojilerindeki hızlı gelişmelerle birlikte iş istasyonları ağı (*İİA*) ortamları, özel amaçlı, yüksek performanslı paralel bilgisayar mimarilerine rakip olarak dikkat çekmeye başladılar. Hesaplama kapasitesi bakımından sağladıkları yüksek ölçeklenebilirlik ve sağladıkları çok daha düşük fiyat/performans oranlarıyla İİA'lar paralel ve dağıtık bilgi işlem ortamları olarak dikkat çekiyorlar. Fakat İİA'ların dağıtık doğalarına bağlı birçok problem, İİA'lar üzerinde çalışacak paralel ve dağıtık uygulamaların geliştirilmesini oldukça zorlaştırmaktadır. Dikkate alınması gereken önemli noktalardan bazıları yazılım ve donanım mimarilerindeki heterojenlik, kontrolsüz ve düzensiz harici yükler, ağ iletişimden dolayı oluşan performans düşmeleri, çok sık değişen işlemci ve ağ bağlantıları üzerindeki yükler gibi sistem özellikleri ve uygulamaların/iş istasyonlarının güvenliğidir.

Genel olarak bu tez çalışmasının amacı heterojen iş istasyonu ağları üzerinde çalıştırılabilecek paralel ve dağıtık uygulamaların geliştirilmesini kolaylaştıracak Java™ tabanlı, yüksek performanslı ve esnek bir uygulama geliştirme platformu ve altyapısı tasarlamak ve gerçekleştirmektir. Paralel ve dağıtık uygulama geliştiricilere sunulan (Java ile geliştirilmiş yazılım parçalarından ve bir Java yazılım kütüphanesinden oluşan) altyapı, geliştiricilerin bu altyapı üzerine dağıtık ve paralel uygulama geliştirip, geliştirdikleri uygulamaları heterojen İİA'lar üzerinde İİA'lar gibi dağıtık sistemlere özel sorunlarla ilgilenmeden çalıştırabilmesini sağlamaktadır.

Yapılan geniş çaplı deneylerin ve testlerin sonuçları göstermektedir ki, Safran hesaplama ağırlıklı paralel ve dağıtık uygulamalar için yüksek ölçeklenebilirliğe sahiptir ve oldukça başarılıdır.

Anahtar Kelimeler: Dağıtık ve Paralel Bilgi İşlem, Java, Heterojen İş İstasyonu Ağları (İİA), Paralel ve Dağıtık Uygulama Geliştirme Altyapıları

To my little sister Sibel with love

ACKNOWLEDGEMENT

First of all, I would like to express my sincere gratitude and thanks to my thesis advisor Prof. Dr. Müslim BOZYİĞİT for his continuous guidance, advice, criticism, and insight all throughout this work.

To my parents, I give my great gratitude for their sustained support and encouragement through my life.

I also would like to express my thanks to my supervisors, fellow workers and friends at CyberSoft for their understanding and support.

Finally to my most loved, Sibel, I would like to offer my regret for the time that I would very much like to spent with her but instead that I had to give to this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CAT	Computer Aided Tomography
CPU	Central Processing Unit
CRCW	Concurrent Read Concurrent Write
DSM	Distributed Shared Memory
HPC	High Performance Computing
HTTP	Hyper Text Transfer Protocol
IP	Internet Protocol
IPC	Inter-process Communication
JVM	Java Virtual Machine
LAN	Local Area Network
MPI	Message Passing Interface
MPP	Massively Parallel Processors
NOW	Network of Workstations
OS	Operating System
PC	Personal Computer
PVM	Parallel Virtual Machine
RMI	Remote Method Invocation
SPI	Service Provider Interface
TCP	Transport Control Protocol
TIES	Two-phase Idempotent Execution
UFS	Unix File System
UML	Unified Modeling Language

CHAPTER 1

INTRODUCTION

1.1 Background and Motivations

Although the capacity and performance of computing systems improve at a high rate, the computational requirements of some scientific and commercial applications are also constantly growing. Many fundamental problems called *grand challenge problems* in science and engineering that have broad economic and scientific significance require such massive amounts of computational resources and performance that their solution on single or sequential systems is unacceptably slow. Moreover, some problems simply cannot be solved sequentially due to the nature of the problems themselves. Some of the problems that require high performance and excessive computational resources are related to:

- Earth sciences
    - Numerical weather modeling and forecasting
    - Seismic exploration
    - Oceanography
- Life sciences
    - Cancer research
    - Drug research
    - CAT (Computer aided tomography)
- Engineering applications
    - Finite element analysis
    - Computational aerodynamics
    - Particle physics
    - Aerospace applications

- Artificial Intelligence
    - Image and speech processing
    - Computer vision

High Performance Computing (HPC) and Parallel Computing deal with the solution of these grand challenge problems. Traditional HPC is based on dedicated architectures called Massively Parallel Processors (MPP) and supercomputers that consist of large numbers of high performance, tightly coupled processing elements. A relatively recent trend is based on connecting low-end individual computing systems (such as workstations or PCs) over high-speed computer networks to form high-end workstation clusters and network parallel computing systems [21], [22], [23], [24]. These solutions are relatively expensive with high cost/performance ratios and of limited utility as they are designed and dedicated to limited, specific parallel and distributed applications. Moreover, they are available only to limited number of researches due to their cost and physical availability, further limiting their utilization.

Continuous and rapid advances in telecommunication and computer technologies (especially in processing power of microprocessors and in communication network capacity) combined with steady decrease in costs of these technologies made networked computers a commodity. It is observed that for more than a decade the processing capacity of microprocessors is doubled every 18 months, and this trend is expected to continue for at least another decade. At the same time communication network capacity increased exceedingly at the local, metropolitan, national and even global level. All these advances allowed almost all institutions build workspaces in which people use many computers connected to each other with high-speed networks. Also the number of personal computers connected to the worldwide computer network i.e. the Internet grew exponentially and reached hundreds of millions [25].

On the other hand, studies on these widely available networked computer environments, especially LANs, have shown that for the most of their operation times, the computers in such environments are used for tasks that are not computationally intensive such as file editing, e-mail reading and Web surfing. In other words, these systems are idle and doing no computation at all for the most of their lifetimes. A typical machine on the average has a 90% idle processor time even during peak times of its use [20].

All the above discussion i.e. existence of grand challenge problems, widespread use of networked computers, and the fact that these systems are idle most of the time motivated many recent researches in development of systems for harnessing the aggregate idle, under-utilized computational power of these well-networked computers to form powerful parallel and distributed computing systems.

1.2 Objectives

The main objective of this thesis work is to design and implement a framework called Safran (software libraries and supporting infrastructure) that is easy to use (in terms of programming, configuration and management) for building distributed and parallel applications on a network of heterogeneous workstations. Safran aims to allow parallel and distributed application developers to easily utilize aggregated idle computational resources of their networked computing systems. The key features of the framework will be:

- **Support for Heterogeneity:** The framework should have uniform support for different software (OS) and hardware (processor, machine, and network) architectures. Users of the framework should be totally abstracted from all kinds of interoperability problems associated with heterogeneous systems and be able to develop and run their applications without needing to deal with any heterogeneity related issues.
- **Virtualization:** The framework should provide the programmers and applications a single virtual parallel machine view of the whole network with high levels of location transparency, fault tolerance, reliability, and robustness.

- **Adaptability:** The runtime system should easily, transparently and automatically adapt to the dynamics of a general workstation network. The participating machines should be able to easily and transparently join and leave the system at any time for any reason including machine and network failures or due to workstation owner preferences. The framework should shield programmers and applications from dynamically changing properties of the system; for example the programmer should not deal with machine failures.
- **Transparent Load-Balancing:** The framework should provide pluggable and configurable (possibly dynamic) load balancing infrastructure transparent to the programmer and applications.
- **Accessibility:** The services of the framework and runtime system should be easily accessible by users from anywhere on the network. For example, the programmers should be capable of running their applications on the system from any machine on the network.
- **Minimized Overhead:** The framework should keep the overhead associated with installation, configuration, and management of the software infrastructure at a minimum level. For example, new machines should be easily and transparently added to a running system.
- **Flexible and Extensible:** The framework should be designed in such a flexible way that future extensions and modifications to the features and services of the framework can be made easily in a pluggable way.

The framework is made up of two parts:

1. A software library that provides high level Java APIs that allow developers to easily write distributed and parallel applications that they can run on their heterogeneous workstation network without having to deal with the system level problems.
2. A set of executable software components (daemons processes on workstations, brokerage services etc.) written in Java. These applications

form the basic infrastructure of the system and they provide the system level services that application developers do not have to deal with.

The provided APIs will support two different computation and programming models:

1. A low level, distributed-objects based computation model similar to RMI and CORBA but with additional concepts and services like remote object creation, object migration etc. In this lower level programming model the application programmer is not provided high level services such as automatic load balancing, fault tolerance etc. On the other hand, the application developer has much more flexibility for developing many types of distributed and parallel applications because of the availability of basic, lower level but complete set of computational building blocks.
2. A relatively higher level computation and programming model based on the high level concept of **distributed computations** that are composed of independently and parallel executable **tasks** is provided. In this model, using the provided API, application developer explicitly expresses his/her computation to the system as a set of sub-computations that can be performed in parallel at different nodes of the system. Scheduling and load-balancing will be performed by the runtime automatically and transparent to the application. In fact, programmer will not be able to control how sub-computations are mapped to the processing nodes because programs will be written for a virtual parallel machine with an unbounded, unknown and dynamically changing number of processing nodes.

1.3 Thesis Layout

In **Chapter 2**, we give a survey of some of the previous works with significant contribution and that attracted most attention. We also provide a discussion and comparison of these works in terms of architecture and supported computation and programming models. **Chapter 3** makes an introduction to general and high level architecture and architectural concepts of Safran such as the abstract computational models it supports, the programming interfaces it provides, and its requirements and assumptions about its runtime environment. In **Chapter 4**, we provide detailed information about Safran's design including the logical layers and abstractions it's composed of, interfaces and interactions of layers, services, sub-systems, logical and physical entities provided by each layer etc. **Chapter 5** presents the experiments we conducted to test and evaluate Safran, and the results we obtained about the performance characteristics of Safran. **Chapter 6** draws conclusions on the effectiveness, strengths and weakness of the system. Finally, **Chapter 7** suggests some possible future extensions to the design and implementation of Safran.

CHAPTER 2

RELATED WORK

Starting in early 90's a large number of researches have attempted to exploit the intrinsic parallelism and latent computational power in distributed system such as heterogeneous LANs and even the Internet for running intensive computations, which were traditionally undertaken with specially designed high-performance MPPs and supercomputers. However, these distributed, heterogeneous computing environments are much harder to program and maintain for parallel and distributed applications because of the issues involved due to their loosely coupled nature. Some of the issues to be considered are: the heterogeneity in the software and hardware architectures, uncontrolled dynamic execution environment, less efficient network communication (high latency, low bandwidth), low reliability, priority for workstation ownership, and security and privacy of applications and hosts.

A large number of software infrastructures and libraries were developed that dealt with these issues of loosely coupled network environments to provide services or desirable execution environment properties for facilitating development of distributed and parallel applications on these system.

Earlier works like PVM[23] and MPI[21] tried to solve basic heterogeneity problems and provided a low level, complete set of explicit message passing primitives for communication and coordination of tasks (or distributed processes), and new distributed computing constructs like remote process creation and execution. Some other low-level libraries such as Linda [26] derived systems and Agora [1] provided Distributed Shared Memory (DSM) based communication facilities. Later works were build on these low-level frameworks to provide some other higher level facilities like load monitoring, adaptive load balancing, fault tolerance, check-pointing, and process migration subsystems for creating high performance, easily programmable, reliable frameworks[16][18][19][27][28]. Although these systems enabled development of distributed and parallel applications on heterogeneous workstation networks, they showed some limitations in achieving a flexible parallel and distributed programming environment. Due to the low-level programming interface (API) they provide, they are relatively difficult to program. They failed in totally shielding the programmer from heterogeneity of the underlying systems. For example, programmers needed to develop, compile, maintain and distribute different versions of their executable code for each different architecture in the system. Also these systems have high setup, configuration and management overhead. Moreover, these systems are not automatically, transparently and easily scalable i.e. they are usually limited to a single network domain. All in all, these frameworks are not flexible in many respects.

The Java Programming Language[31][32] is rapidly being adopted as one of the most important and widely used languages for parallel and distributed application development due to the excellent features it offers that are lacking in traditional languages such as C, C++ and FORTRAN. Java’s attraction is mainly due to its clear and effective solution to the portability and interoperability problem associated with heterogeneous machines and operating systems with its standardized virtual machine architecture whose implementation is available for almost all systems. A Java program once compiled can be run on any system that has a Java Virtual Machine (JVM)[33] installed. Moreover its features such as automatic memory management (garbage collection), rigid security infrastructure, extensive support for network programming, multi-threaded programming, synchronization mechanisms, object serialization, and code mobility makes it an excellent choice for distributed, network-parallel programming. Some other frameworks like Remote Method Invocation (RMI)[29][36], Jini[34], and JavaSpaces[30] that are build on standard Java to extend platform’s capabilities further simplify development of large scale distributed and parallel applications.

Recently, significant number of Java-based systems has been developed to support distributed and parallel programming on heterogeneous workstation networks and even on the Internet. SuperWeb[2], Javelin[3], Janet[4], Charlotte[5], Ajents[7], KnittingFactory[6], JavaParty[8], ParaWeb[9], JavaSymphony[10], JPVM[11], JavaNOW[12], JAVM[13], ALTAS[14], and Ninflet[15] are some of the works carried out in the academia with different properties in terms of objectives, architecture, scalability, supported computational model, type of programming model provided etc. In the following we give an overview of the most directly related works with significant contribution and that attracted most attention.

2.1 SuperWeb, Javelin, JANET (University of California, Santa Barbara)

Javelin[3], originally a prototype of SuperWeb[2], was reported in 1997 as a Java-based infrastructure for global computing. The goal of Javelin is to harness the Internet's vast, growing, computational capacity for ultra-large, coarse-grained parallel applications. The work that was started with SuperWeb has been continued with new versions named Javelin++, Javelin 2.0, Javelin 3.0, CX, JICOS, and currently JANET[4], each with improvements in performance, scalability, computation and programming model. SuperWeb and the first version of Javelin were designed based on Java applets running on web browsers. Starting with Javelin++ the system is based on standalone Java applications instead of applets due to various limitations of applet-based architecture.

Although various improvements made, the essential architecture remained almost same: The whole system is based on three system entities named clients, brokers, and hosts. A *client* is a process seeking computing resources, a *host* is a process offering computing resources, and a *broker* is a process that coordinates the allocation of computing resources. Hosts offer their resources to the world by registering to the brokers. Brokers essentially form a directory of available hosts and they coordinate resource consumption across clients and hosts. As the computational model, Javelin supports *piecework computations* (also called master/slave, manager/worker, or bag-of-tasks): adaptively parallel computations that decompose into a set of sub-computations, each of which is autonomous (does not require communication or coordination with other sub-computations), apart from scheduling work and communicating results. Parallel matrix multiplication, ray tracing, and Monte Carlo simulations are some of good examples of piecework computations. Javelin also supports *branch-and-bound* computations. Javelin achieves scalability and fault-tolerance by its network-of-brokers architecture and by integrating distributed deterministic *work stealing* with a distributed deterministic *eager scheduling* algorithm.

2.2 Charlotte (New York University)

Charlotte[5] is one of the first Java-based frameworks of its kind. Similar to Javelin, its architecture is mainly based on Java applets embedded into Web pages. It aims to utilize the Web as a parallel meta-computer but it clearly does not scale much and experiences bottlenecks due to the limitations of its applet-based architecture.

Charlotte provides distributed shared memory (DSM) architecture within the language, at the data type level i.e. through classes so it does not modify the Java Virtual Machine (JVM), nor does it rely on a preprocessor or require any kind of external runtime support (ex. OS support). For every basic data type in Java, there is a corresponding Charlotte data type implementing its distributed version. The consistency and coherence of the distributed data is maintained by the Charlotte runtime systems. The provided DSM memory-consistency semantics is Concurrent Read, Concurrent Write Common (CRCW-Common) i.e. one or more entities can read a shared variable, and one or more entities can write a variable as long as they write the same value.

Charlotte programs are written for a virtual parallel machine with an unbounded number of processors sharing a common namespace i.e. the programmer has no knowledge of how many machines will execute a computation. The main entities in a Charlotte program are a manager (i.e. a master task) executing serial steps and one or more workers executing parallel steps. The manager process creates an entry in a well-known Web page for the active computation and volunteer users load and execute the worker processes as Java applets embedded into web pages by pointing their browsers to this page. In essence, the programming model supported by Charlotte is master-slave (master-worker, or bag-of-tasks) programming model. The computation is first divided into a large number of small computational units, or task. Then participating machines pickup and execute a task one at a time until every task has been executed.

Charlotte achieves load balancing and fault masking by implementing two concepts: *eager scheduling* and *two-phase idempotent execution* (TIES). Eager scheduling aggressively assigns and re-assigns tasks until all tasks are completed when the number of remaining task becomes less then the available machines. Concurrent assignment of tasks to multiple machines prevents slow, un-accessible, or faulty machines from slowing down the progress of the computation. Multiple executions of a task (which is possible when using eager scheduling) can result in incorrect program state. TIES ensures idempotent memory semantics in the presence of multiple executions. TIES guarantees correct execution of shared memory by reading data from the master and writing it locally in the workers' memory space. Upon completion of a worker the dirty data is written back to the master who invalidates all successive writes, thus maintaining only one copy of the resulting data. Moreover Charlotte employs *dynamic granularity management* (bunching) to mask latencies associated with the process of assigning tasks to machines. Bunching is achieved by assigning a set of task to a single machine at once. The size of bunches is computed dynamically based on the number of remaining tasks and number of available machines.

Charlotte has a number of problems. The primary function of the manager is scheduling and distributing works. But it is also responsible for communication of workers (i.e. it implements the DSM) as all applet-to-applet communication is routed through it. So it is a potential bottleneck. Secondly, Charlotte requires that a manager run on a host with an HTTP server. If only one machine with an HTTP server is available and if more than one application needs to be running on the system concurrently, then multiple managers have to be running on this only machine. In this situation such a machine can possibly become a communication bottleneck. Moreover multiple assignment and execution of the same task with eager scheduling might cause excessive data traffic leading further performance problems. All in all, Charlotte does not seem to scale enough to meet its goal of utilizing the Web as a big meta-computing platform.

2.3 KnittingFactory (New York University)

KnittingFactory[6] is the successor of Charlotte from the same research group. With KnittingFactory some of the limitations and deficiencies of Charlotte is eliminated. KnittingFactory addresses the following problems:

- Searching and finding other members of a collaboration session.
- Ability to run a distributed application on a machine without an HTTP server.
- Direct communication between applets.

A distributed registry is implemented based on Web servers and standard Web browsers. A registry accepts requests for partners, stores these requests, and deletes them again upon request. A user who wants to participate in a distributed computation simply points his/her browser to one of these registries. Charlotte required that master processes of applications run on a machine with a running HTTP server. KnittingFactory factory solves this problem by automatically embedding a HTTP server in each application. So now applications can run on any machine. In Charlotte, all the communication between applets is routed through the master process. KnittingFactory supports direct communications between worker processes (applets) by exploiting a non-standard (maybe a bug) property of Sun JVM to pass RMI object references between applets.

2.4 Ajents (York University, University of Waterloo)

Ajents[7] is a Java based framework that provides necessary infrastructure for building parallel and distributed applications. Unlike many other similar frameworks Ajents does not make modifications to the Java language or to the JVM and no preprocessors, special compilers, or special stub compilers are required. It is a collection of pure Java classes (a library) and servers (also implemented in Java) so it runs on any standard compliant JVM. Every host in the system runs a simple lightweight server called Ajents Server as a daemon process.

Ajents greatly simplifies the task of writing distributed applications by providing features that are not available in standard Java. Actually it is built on standard Java RMI technology but extends the distributed object model of RMI by providing support for:

- **Remote Object Creation:** While RMI allows local referencing and remote method invocation on statically created remote objects, Ajents supports dynamic creation and referencing of objects on remote hosts that are running Ajents Server.
- **Remote Class Loading:** To be able to instantiate objects on remote hosts or to move around objects (i.e. object migration) between remote hosts, Ajents transparently loads the binary executable code of the object to the remote host.
- **Asynchronous Remote Method Invocation:** Java RMI only allows invocation of methods on remote objects synchronously (or serially). Ajents supports asynchronous RMI i.e. a process can call a method of a remote object and continue execution until an arbitrary time when the result of the method call is available.
- **Object Migration:** Ajents allows migration of objects between heterogeneous hosts without a preprocessor, and without modification to the virtual machine, compiler or stub compiler. This is implemented using check-

pointing, rollback and restarting mechanisms. Any object can be migrated while it is executing by interrupting its execution, moving the most recent check-pointed state of the object and restarting the currently executing method.

The main idea for programming in Ajents environment is to have an Ajents Server running on every host in a heterogeneous environment and writing applications in Java using the Ajents class library calls to use resources on these servers by creating remote objects, invoking their methods, and moving around (migrating) them between hosts as necessary.

Ajents framework does not support transparent dynamic load balancing but of course users can implement custom load balancing on top of Ajents framework. Although not clearly explained, it has some scheduling mechanism that allows selection of an appropriate host to migrate objects.

Unlike Javelin, Charlotte and KnittingFactory type of frameworks, Ajents does not dictate any high level distributed or parallel computation model. Instead, it provides relatively low level and complete, easy to use framework for building any kind of distributed applications or even higher-level frameworks like Javelin on top of Ajents. In another view, it's simply an extension of distributed object model of RMI technology with new distributed programming facilities.

2.5 JavaParty (University of Karlsruhe, Germany)

Although Java platform includes RMI technology for distributed programming it is not easy and straightforward to write distributed and parallel applications for distributed shared memory architectures like clusters of workstations. JavaParty [8] transparently adds distributed, remote objects to Java simply by declaration, avoiding complexity, disadvantage and programming overhead of socket based, or RMI based programming. JavaParty is targeted towards and implemented on clusters of workstations. It extends the Java language simply and transparently with a preprocessor and a runtime system for distributed parallel programming in heterogeneous workstation clusters.

JavaParty adds the *remote* keyword to the Java language. The programmer simply attributes classes that should be spread across the distributed environment with the *remote* keyword. JavaParty uses a preprocessor, which converts remote classes into pure Java code with RMI hooks. The change to the language is designed to simplify RMI programming, placing the burden of creating and handling remote proxies upon the preprocessor, simplifying the programming task.

Objects of these remote classes can transparently be created on remote hosts and referenced locally as if they are local objects. JavaParty is location transparent i.e. it maps created remote objects to hosts transparently. The compiler and runtime system deals with locality and communication optimization using pluggable distribution strategies. The programmer can only influence distribution and mapping of object to hosts by developing and inserting code that directs the strategy's placement decisions. Besides distribution strategies at object creation time, JavaParty monitors the interaction of remote objects and if it is appropriate (or is requested by the programmer) can schedule object migration between hosts to enhance locality.

2.6 JavaSymphony (University of Vienna)

Basically JavaSymphony [10] extends the distributed object framework provided by Java RMI Technology with new high level constructs, similar to Ajents and JavaParty.

High-level distributed and parallel programming frameworks that do not provide programmer control over locality of data by automatically distributing and migrating objects can easily lead to loss of performance as the underlying runtime system has little information about the distributed computation. In contrast to most existing systems, JavaSymphony provides the programmer with the flexibility to control data locality and load balancing by explicitly mapping objects to computing nodes.

The key features of JavaSymphony that greatly simplifies performance-oriented distributed and parallel programming are:

- **Dynamic Virtual Distributed Architectures**
  The programmer can dynamically define and modify virtual distributed architectures that impose a virtual hierarchy on distributed system of physical computing nodes. Virtual architectures consist of a set of components: computing node, clusters (collection of nodes), sites (collection of clusters), and domains (collection of sites). Virtual architectures effect how automatic mapping and migration of objects are done for locality and load balancing decisions.
- **Access to system parameters**
  JavaSymphony provides access to a large variety of periodically monitored system parameters such as CPU load, idle times, available memory, network latency and bandwidth, etc. Programmer can use these parameters for load balancing decisions.
- **Automatic and User-Controlled Mapping of Objects**
  The programmer can control the creation and mapping of objects to specific components of the virtual architectures. If the programmer does not provide explicit mapping of objects the JavaSymphony runtime offers automatic mapping based on periodically monitored systems parameters.
- **Automatic and User-Controlled Object Migration**
  JavaSymphony supports both automatic and user-controlled migration of objects based on periodically monitored systems parameters.
- **Asynchronous, Remote, and One-sided Method Invocation**
  JavaSymphony supports both synchronous and asynchronous remote method invocation. Moreover for methods that do not return any value it supports one-sided method invocation.
- **Selective Remote Class Loading**
  JavaSymphony automatically loads binary class codes only to the nodes on which they are actually needed. This feature can reduce overall memory requirements of an application.

JavaSymphony has been influenced by Ajents programming model for remote object creation, asynchronous remote method invocation and remote class loading. Its most significant contributions over Ajents are its support for virtual architectures, one-sided method invocations and access to system parameters.

2.7 JavaNOW (DePaul University)

JavaNOW[12] aims to provide an environment for parallel computing on an ordinary network of workstations that is both expressive and reliable. It creates a virtual parallel machine similar to the Message Passing Interface (MPI) model, and provides distributed shared memory (DSM) similar to Linda memory model but with a flexible set of primitive operations. It can be view as a hybrid system of PVM, MPI, and Linda. It provides a simple mechanism to start tasks on remote hosts (as found in PVM), has a small number of expressive and complete primitives to support producer/consumer style communication (as found in Linda) and finally has collective operation that can be performed on shared objects (as found in MPI).

In a JavaNOW system, processes coordinate and communicate through a distributed associative shared memory similar to tuple-space model and using PVM and MPI like primitives that are build solely on this shared memory model. The Linda like DSM is implemented as a totally distributed and load balanced data structure (actually a distributed hash-table). So it differs from MPI and PVM in that it does not provide direct point-to-point inter-process communication (IPC) primitives but rather provides a producer/consumer model of IPC. Mutually exclusive access operations to the shared memory make it easy to implement distributed synchronization primitives like locks, mutexes, and semaphores.

JavaNOW does not provide dynamic resource management but requires that the user statically specify the list of machines on which the application will run. Also it does not provide dynamic load-balancing at work distribution. It just uses a simple hashing scheme for load balancing the distributed shared memory.

2.8 Discussion

The presented works generally can be grouped in two according to the computational and corresponding programming models they support or dictate:

- **Low-level Frameworks:** These kind frameworks provide a lower-level programming interface which is usually an extension of Java RMI distributed object framework with additional services and features such as remote object creation, object migration, asynchronous remote method call etc. They mostly aim to extend Java RMI to support and ease the development of general distributed applications based on the concept of *distributed-objects*. Most of them are directly built on Java RMI technology as another layer of library (API). As the programming concepts, primitives and elements they provide (such as synchronous/asynchronous remote method invocation, remote object creation, object migration etc.) are low-level and basic, they do not dictate any high-level parallel and distributed computational model and they allow a richer variety of distributed and parallel applications to be developed on them. On the other hand, they usually do not provide higher level services like load balancing, job scheduling, and fault tolerance because of the flexibility they provide. Some examples of these low-level frameworks are Ajents, JavaParty, and JavaSymphony.

- **High-level Frameworks:** These kinds of frameworks provide higher-level programming interfaces that correspond to high level parallel and distributed computing concepts like *parallel applications*, *tasks*, and *jobs*. For example, application programmers are required to express their distributed computation as a set of parallel executable sub-computations (usually named as *tasks* or *jobs*) to the framework using the provided high-level API and the underlying infrastructure transparently takes care of execution of sub-tasks and collection of results. Because of the layer of abstraction provided to developers, the frameworks relieve developers from low-level system problems such as dispatching works to remote machines, executing them remotely, and collection of the results. Also such frameworks are able to provide some other important services such as scheduling, load balancing, and fault tolerance. On the other hand, the users are usually restricted to some single kind of computational model as they leave most of the low-level functionality to the framework. Some examples of high-level frameworks are SuperWeb, Javelin, Janet, and Charlotte.

Safran provides programming interfaces similar to the ones used in both low-level and high-level frameworks. A user application can be written against a low-level API similar to Java RMI with the concept of both synchronous and asynchronous method invocations on remote objects, creation of objects on remote machines, and migration of objects between remote machines. Most important contribution of Safran is its easy to use and familiar programming environment. Developers can create remote objects on remote machines, call methods on them, migrate them using simple, familiar syntax and programming concepts. Using Safran, developers can also develop applications using another high-level API, which is similar to the presented high-level frameworks but again by providing more clear and easy to use interfaces.

CHAPTER 3

ARCHITECTURAL CONCEPTS AND DESIGN

In this chapter, we make an introduction to general features and design considerations of Safran. We explain the high-level architectural organization and architectural concepts of a Safran system. Next, internal architecture and logical layering of Safran's sub-systems are introduced.

3.1 Introduction

In this section, an introduction to features and design considerations of Safran is made. Technical requirements and expectations from Safran, the abstract computational models it supports, the programming interfaces it provides and its requirements and assumptions about its runtime environment are explained.

3.1.1 General Requirements

The main objective of Safran is to provide an infrastructure that is easy and flexible to use in terms of programming, setup, configuration, and management for building distributed and parallel applications on a general network of heterogeneous workstations. The key features of Safran are:

- **Support for Heterogeneity:** Safran has uniform support for different software (OS) and hardware (processor, machine, and network) architectures. Users of Safran (usually application programmers) are abstracted from interoperability problems associated with heterogeneous systems and are able to develop and run their applications without needing to deal with any heterogeneity related issues.

- **Virtualization:** For certain kinds of parallel applications, Safran provides the programmers and applications with a single virtual parallel machine view of the whole network with high levels of location transparency, fault tolerance, reliability, and robustness.
- **Adaptability:** The runtime system of Safran adapts to the dynamics of a general workstation network automatically and transparently. The participating machines can easily and transparently join and leave the system at any time for any reason including machine and network failures or due to workstation owner preferences. The framework shields programmers and applications from dynamically changing properties of the system; for example the programmer does not need to deal with machine failures.
- **Transparent Load-Balancing:** Safran provides load-balancing infrastructure and services, which is transparent to the programmer and applications.
- **Accessibility:** The services of the Safran runtime are easily accessible by users from anywhere on the network. For example, the programmers can run their applications on a Safran system from any machine on the network.
- **Minimized Overhead:** Safran keeps the overhead minimum associated with installation, configuration, and management of the software infrastructure. For example new machines can be easily and transparently added to a running system.

3.1.2 Assumptions and System Requirements

The only key requirement of Safran environment is a standard Java Runtime Environment (JRE) of version 1.5 or later.

3.2 Computational and Programming Models

Safran provides support for two different kinds of computational models that correspond to two different levels of programming models. The one that we named as the low-level programming model provides relatively low-level concepts and constructs based on the idea of distributed-objects. The other and primary computation model is based on the higher-level concepts of distributed applications that can be divided into independent and parallel executable sub-computations.

3.2.1 The Low-Level Computation Model

The low-level programming model is supported through a low-level API that provides a set of programming concepts and constructs similar to that of *distributed-objects based* programming frameworks such as RMI. Besides supporting the concept of calling methods on remotely published objects, Safran's low-level programming model introduces new facilities and services that makes distributed-objects based programming easier than RMI programming. This programming model provides the following facilities:

- **Easy creation of objects in the address space of remote processes:** Developers can create objects on remote machines just with a single system call and get back a local reference that allows access to the remote object. No manual setup is required for things like registries or name services, or class file servers.
- **Exposing local objects to remote access through well-known names:** Similar to RMI, Safran allows publishing a local object with a well-known name so that remote processes can access the object with this well-known name.
- **Calling methods on remote objects synchronously or asynchronously:** Methods on remote objects can be called synchronously just like local method calls. Also asynchronous method calls can be made on remote object so that results or return values can be retrieved asynchronously without serially waiting the execution of the called methods. Moreover, one-way method calls, in which the method is called and the caller is not interested in the result of method call, can be made easily.

- **Automatic and transparent creation of dynamic proxies:** Whenever a remotely exposed object is accessed or an object is created on a remote process, a local proxy that provides the same interface as the remote object is created dynamically and transparently. As this local proxy object provides the same interfaces as the remote object, the developer can call the methods of the remote object using the familiar local method call syntax (`object.methodName(parameters…)).`
- **Automatic and transparent dynamic class loading:** Class definitions (executable, binary Java class files) are automatically and transparently transferred to the remote machines whenever they are needed. Application developers do not need do anything manually to deploy their binary class files. Whenever the class definition (binary class file) of an object is required (when an object is created remotely, when an object is migrated to a different machine, when a copy of a remote object is retrieved etc.) Safran transfers the class definition to the required machine. Users do not need to any manual setup to distribute the class files to the distributed machines.
- **Object migration:** Developers can move (or migrate) remote objects between different machines with simple system calls. Also a local object can be moved to a remote machine and it can be accessed remotely later on.
- **Object state check-pointing:** State of remote objects can be check-pointed by getting an exact, local copy of a remote object.

The computational model that corresponds to this low-level programming model is theoretically equivalent to message-passing type of parallel and distributed computing model. Message passing systems like PVM and MPI provide much lower-level programming interfaces with explicit message passing primitives. On the other hand, distributed-objects based systems provide higher-level programming constructs and concepts like calling methods on remote objects which theoretically corresponds to passing messages to remote processes. But of course, passing messages between processes through a remote method call interface is much high-level and easier to use than explicit message passing systems like PVM and MPI. So the low-level, distributed-objects based programming model provided by Safran theoretically supports the message passing type distributed and parallel computational model i.e. theoretically applications that can be developed using explicit message passing systems can also be developed using Safran's low-level programming model.

3.2.2 The High-Level Computation Model

The high-level and primary computation model supported by Safran is called piecework computations (master-worker, or bag-of-tasks) in which the computation is explicitly divided into many sub-computations that can be performed in parallel at different nodes of the system. Piecework computations are parallel computations that decompose into a set of sub-computations, each of which is autonomous (does not require communication or coordination with other sub-computations), apart from scheduling work and communicating results. Parallel matrix multiplication, distributed ray tracing, and Monte Carlo simulations are some good examples of piecework computations.

The corresponding high-level programming model is provided by an API that allows developers express their distributed and parallel computation to Safran as a set of independently and parallel executable sub-tasks. Safran executes the sub-tasks on the remote nodes of the system in parallel and collects the result and makes them available back to developers' application. Safran has the total control on how and where the sub-tasks are executed so it can transparently provide high-level services such as load balancing and fault tolerance.

Note that in this computational model, Safran does not allow communication between sub-computations. This limitation is not due to technical difficulties in providing communication infrastructure for pieces of a distributed computation. Instead, it's due to the challenging problem that would appear if communication were allowed between distributed sub-computations. This problem is the classical problem of getting global snapshots of a distributed system. If communication is allowed between distributed processes or pieces of a distributed computation, states of communicating processes become dependent on each other. In such systems, if it is required to checkpoint the system state for later rollbacks, the whole distributed state (all parts of the distributed computation) must be check-pointed at once, otherwise later rollbacks yield inconsistent global system states. Safran aims to provide fault tolerance by checkpoint-and-rollback protocols. If communication between sub-computations were allowed, Safran would need to checkpoint the state of the whole distributed computation whenever it needs to checkpoint the state of a single sub-computation for later rollbacks. Otherwise the checkpoints would be inconsistent and subsequent rollbacks would yield to inconsistent global states. Both supporting transparent fault tolerance using checkpoint-rollback and supporting communication between sub-computations would require Safran to implement a solution to the challenging distributed global system state snapshot problem which is out of scope of this thesis work.

So, in the limited scope of this work, we are either required to support transparent fault tolerance by not allowing sub-computations to communicate, or otherwise allow sub-computations to communicate but do not provide transparent fault tolerance. Safran opts to provide transparent fault tolerance based on checkpoint-and-rollback of sub-computation by not allowing communication between them. Future extension to Safran might provide communication infrastructure based on shared memory abstraction but it is out of scope of Safran's current design and objectives.

The high-level programming model of Safran was initially designed as the primary and the only programming model. The low-level, distributed-objects based programming model of Safran was initially designed as a supporting infrastructure for the primary high-level programming interface. In other words, the programming model and the API that provides the high-level computational model actually is built on the infrastructure that provides the low-level distributed-objects based programming model. Later, the infrastructure that was initially designed to just internally support the primary programming model was extended to a general-purpose distributed programming infrastructure and publicly exposed to users of

Safran as a general purpose programming model to support more kinds of distributed and parallel applications.

3.3 System Architecture

In this section high-level, architectural design of Safran *that supports the primary high-level computational and programming model* (as described in detail in **Section 3.2.3**) is explained. Physical and logical entities (components) of the system, their relations with each other and the subsystems and services they implement are detailed.

3.3.1 Overview

In Safran there are logically there different system entities. These are named **Hosts**, **Brokers**, and **Applications**. Applications act as the clients or users of the whole system. They run the central, controlling, serial logic of the distributed and parallel computations of users. Hosts provide computational resources (mostly CPU cycles) of the physical machines they reside to the Safran system by getting and running independently executable parts of Applications. Brokers generally provide brokerage services by getting Applications and Hosts meet. **Figure 3.1** shows the high level logical organization of entities of a sample Safran system.

These entities form the overall system by implementing or participating in the implementation of different subsystems. Note that these entities are logical in that they do not represent physical machines that they reside on. For example, on a single machine more that one Host might be running or a machine might be hosting both a Host and an Application, or even a Broker. On the other hand they are implemented as OS processes (usually daemon or service processes) so they are not totally low-level software abstractions.

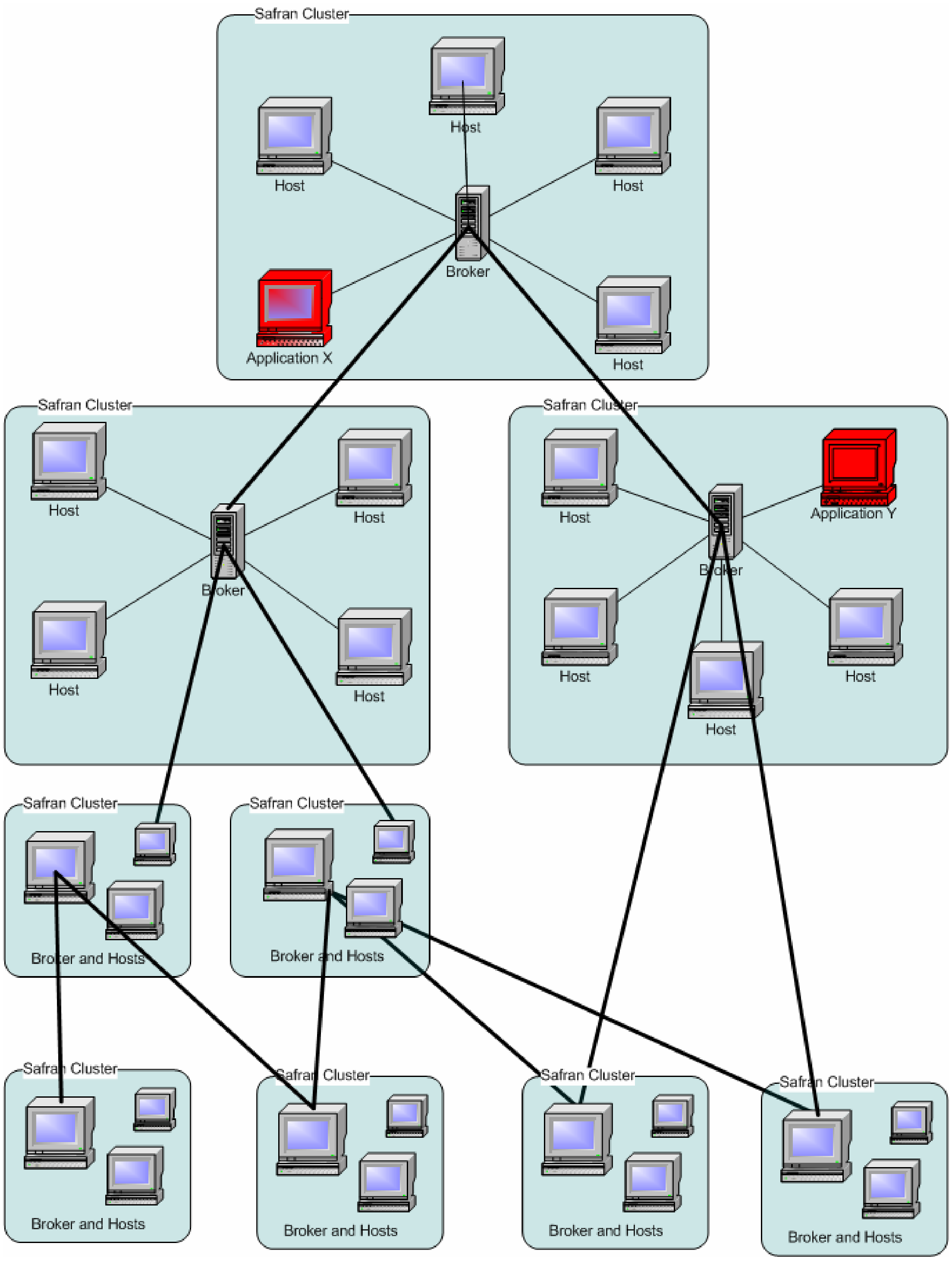

Figure 3.1 Architectural organization of entities of a sample Safran system.

3.3.2 Brokers

The general, high level and visible service provided by brokers is meeting Hosts and Applications. Brokers generally act as registries of Hosts. They are contacted by Applications that need Hosts for running different, concurrently executable parts of the Applications. So, their main responsibility in the whole system is allocation of Hosts to Applications. To fulfill their responsibility effectively, they implement subsystems such as host registry, host scheduling and allocation, and load balancing.

Brokers are implemented as daemon (or service) processes that run in the background and they usually do not show user interface elements other than the management user interface that is invoked on demand by users of the machines. They are generally passive in that they wait for requests for different operations from Hosts and Applications and fulfill them. Other than processing requests, they do not perform any ongoing intensive processing. Hosts and Applications can contact brokers by just knowing the network address (host name or IP address) of the machine on which the broker resides because all brokers service at a preconfigured fixed port. For example, when a Host or Application wants to contact a broker, whom they believe to be residing on the machine A, they request a reference from Safran infrastructure to the entity running on the machine A and at the preconfigured broker port. They are handled a reference to the found entity (if found) by the Safran infrastructure, which actually happens to be a reference to the only Broker residing on the machine. Later they communicate with the broker with this reference. An important implication of this design is that, unlike Hosts and Applications there can be only one Broker running on a single machine.

Hosts and Brokers can be manually configured to use a specific Broker running on a machine at a specific network address. Also they can be configured to automatically discover and use the Brokers that are running on their own network. To enable automatic discovery of Brokers by Host and Applications, Brokers broadcast periodic messages to the network announcing their availability. If configured to discover and use a Broker automatically, Hosts and Applications monitor the broadcasted messages from Brokers and when they determine the location (network address and port) of a Broker they automatically contact it. This service of automatic management of relations between Safran entities greatly improves the setup and management procedures of Safran clusters. One can just start Brokers, Hosts and Applications on the different machines and all entities can automatically find and contact each other.

The most important subsystem implemented by Brokers is Host registry subsystem. Hosts that want to participate in a Safran system contact (either manually or by automatic discovery) and register to a broker by providing information (availability, machine capacity, and usage policy) about themselves to the broker. Brokers keep a list of Hosts registered to them and some detailed information about computational capacity of each host. Hosts actively update the information about them either periodically or in the case of state changes that are above configured thresholds. The aim is to provide brokers as much as up-to-date information, which they use for scheduling and load balancing decisions. Also, registered Hosts should notify the Broker periodically (for example every one 60 seconds) to inform it that they are alive and still servicing. If a Broker does not get a notification from a Host that registered to it for some time longer than a configurable timeout value, it automatically un-registers and removes it from its list.

A Broker and Hosts registered to that Broker form a logical Safran Cluster. This logical organization does not need to map physical network topology, i.e. Hosts that are on totally different physical networks can register to the same Broker to form a Cluster. Usually, due to manageability and performance considerations, Hosts register to Brokers that are physically more accessible to them. For example, it is most common that all Hosts running on the same LAN (Local Area Network) register to a single Broker that is also connected to the same LAN. A Cluster is the smallest possible complete Safran system but clusters can be connected to each other to form larger and high capacity systems. As Brokers are the central entities of Safran Clusters, connection and communication between Clusters is handled by Brokers. Each Broker can be configured to register to one or more other Brokers to let them know that it (i.e. its cluster) wants to use resources of those other clusters. When a Broker registers to another Broker, it starts to receive periodic or updated information from that Broker about the cluster it controls. This information is some high-level, not much detailed data that gives a general idea to the Broker about the other cluster's overall computational capacity. Brokers use this information about the other Brokers they registered when making decisions that effect how they use those other Clusters. For example, when a Broker cannot fulfill a Host request of an Application itself, it uses the information about the other Brokers it registered when choosing the broker it will forward the request.

The relations and communication between Brokers (i.e. Safran Clusters) is unknown and transparent to Host and Applications. Hosts and Applications only communicate and depend on the Broker of their own Cluster. This design makes the system scalable and easily extensible. Individual Clusters can be combined and connected dynamically through Brokers to form larger and more capable systems.

As mentioned, the sole, main responsibility of Brokers is to allocate Hosts to Applications that need these Hosts to run tasks on them. When an Application needs to run one of its tasks on a Host, Safran infrastructure contacts the Broker of its Cluster and makes a request for an available Host. The Broker first tries to select an available Host from the ones that registered to it, i.e. from its own Cluster. If none of the registered Hosts is available, it forwards the request to each of the Brokers of the other Clusters it knows. This way an Application running in one Cluster can transparently use Hosts of other Clusters. In other words, all resources of the network of Clusters can be made available to any Application running in any Cluster.

When a Broker is allocating or scheduling Hosts to Applications it uses the information about Hosts that registered to it, other Brokers it registered to, and also requirements of the Applications to balance the usage of system resources. So, an important part of load balancing subsystem is implemented by Brokers during their scheduling or Host allocation decisions. As mentioned, a Broker's load balancing decisions are based on the information collected from Hosts registered to the Broker, other Brokers, and Applications that request Hosts from the Broker.

3.3.3 Hosts

Hosts are entities that provide computational resources (usually CPU cycles) to a Safran system. Similar to Brokers they are implemented as daemon processes. They execute as background processes and run Applications' tasks. More than one Host might be residing on the same machine, although usually this is not the case.

Hosts make available their resources to the Safran system by registering to a Broker. *At any time, each Host can be registered to only one Broker.* Other entities (usually Applications) of the system can only contact and access a Host through the Broker it registered to. So, when a Host is not registered to a Broker it is totally inaccessible to the system. Hosts register to Brokers at their startup. A Host determines and contacts the Broker to register either according to the manual user configuration or by automatically discovering the Broker. The user who starts the Host agent (process) configures whether the Host should use a user specified Broker or should automatically find a Broker to register.

When a Host is registering to a Broker it provides initial, static information about itself to the Broker. This information includes indicators about its hosting machine's computation capacity such as physical memory, number of CPU's, and capacity of CPU's. Later on, it provides dynamically changing information about itself either periodically or in case of changes in it state that are beyond some configured thresholds. For example, it provides current and 1, 5, 15, and 30 minutes average values for available physical memory and CPU utilization. Also it might inform the Broker when its available physical memory drops by 10%, or when its CPU utilization increases by 50%, or when it is exclusively allocated to an Application. The Hosts must also send simple periodic notifications to the Brokers they registered to, to let them know that they are still up and running. Otherwise, Brokers will unregister the Hosts if they do not get notification from the Hosts for some time longer then a configurable timeout period (ex. 30 seconds).

When an Application needs a Host, it sends a request to a Broker. When a request for a Host comes to a Broker either directly from an Application or from the Broker of another Cluster, it selects a registered and available (in terms of current status information, usage policy and Application requirements) Host and it handles a reference of the selected Host to the Application. At this point the Broker is done and does not do anything for communication and coordination of the Host and the Application. It even does not update its information about the Host because whenever the Application starts to use the Host allocated to it, the Host's status information will be changed and the Host will inform the Broker about this change. In other words, the broker does not try to actively track the Host it allocated to an Application or keep information about its usage. During the Application's usage of the Host, the Host already informs the Broker about its status changes. So the Broker's only responsibility is to allocate a Host to an Application in a load balanced manner and leave them alone. It is not directly interested in what is going on between the Host and the Application. In this design the responsibility of Brokers is minimized and made very specific. It does not keep any state or runtime information about running applications, so its failure does not directly effects running applications or bring them down. If a Broker crashes when an Application is running in its Cluster, the Application does not fail because Brokers does not maintain any information about Application. The Application just cannot run new tasks until the Broker of the Cluster becomes available again.

When a Host is allocated to an Application, it is used by that Application exclusively. So the same Host cannot be allocated to more then one Application at the same time. The reason behind this restriction is the fact that Safran is designed for computationally intensive applications so it is assumed that when a Host is allocated to an Application the Application uses Host's all computational capacity. Whenever an Application is done with a Host it explicitly frees it and the Host can be allocated to other Applications.

3.3.4 Applications

Applications are users (i.e. clients, or consumers) of Safran systems and Safran resources. Applications are software developed by end-users (developers or programmers) using building blocks (software libraries i.e. APIs) provided by Safran infrastructure. *These applications are the type of applications that are built on the high-level programming model of Safran (see* ***Section 3.2.2*** *and* ***Section 3.4.6****)*.

Each Application might perform different computations or solve different problems but they share a common structure imposed by the underlying Safran infrastructure as they all use the same common libraries, APIs, and Safran services. For an application to be able to use services or resources of a Safran system effectively, not only it must be developed using the provided APIs by Safran, but also it must be developed adhering to rules and guidelines that are required but cannot be imposed syntactically by the APIs.

General structure of a Safran application is as follows: First it initializes the Safran infrastructure for by making a system API call. Next, it decomposed its computation into smaller pieces according to its specific problem that it is trying to solve and handles them to Safran. For each piece of sub-work (i.e. a Task), Safran requests a Host from the Broker of the Cluster. It dispatches each Task to one Host, execute it there and get the result of each. Finally it notifies the Application that the result of the executed Task is ready. Application combines the results of sub-tasks locally to form the final, combined result of the problem. In another words, the Application entity acts as the central, controlling logic of a distributed/parallel computation. It decomposes a computation into concurrently executable sub-computations, lets Safran execute them in available Hosts in parallel, and finally gets the result of each from Safran to compose the result of the whole computation. Most of the plumping work such as requests for Hosts, allocation of them, dispatching of tasks to Hosts are all handled by the Safran libraries transparently to the programmer i.e. the programmer does not need to write code for doing all these. The Safran APIs used by the programmer provides the programmer and application a single system image of the whole Safran system, which actually consists of complex networks of Hosts, Brokers and the clusters they form.

Other then being developed on top of Safran libraries using Safran APIs, Applications are not constrained in any way. They can be executed on any machine that has Safran libraries installed and that has network connection to a machine hosting a Broker. On the machine that will execute an Application only the Safran libraries are required. There is no need to run any kind of other system service or daemon process like a Host, or a Broker.

3.4 Logical Design

This section presents an overview of the design of the Safran's software infrastructure in terms of layering of services, and the abstractions each layer provides to the others.

3.4.1 Overview

General structure of Safran's infrastructure is depicted in **Figure 3.2**. The entire infrastructure is organized as layers, each of which uses lower-level services of its underlying infrastructure and provides higher abstractions to the layers above them. The following sections explain the responsibilities and services of each layer.

3.4.2 The Java Language and Runtime Environment

As shown in **Figure 3.2** Safran is totally built on the Java platform and runs in a Java Virtual Machine (JVM). Also the user applications that are built on Safran must be developed using the Java Programming Language, so they are also running in the JVM.

The Java platform, which consists of the Java programming language and the Java Runtime Environment, has a number of features that greatly facilitates distributed system programming. Java's built in features such as automatic garbage collection, rigid security infrastructure, extensive support for network programming, multi-threaded programming, synchronization mechanisms, object serialization, reflection, code mobility, dynamic class loading and general extensibility mechanisms make it an excellent choice for distributed, network-parallel programming. Safran takes advantage of many of these built-in features and services of the Java platform to produce a system which would be far more difficult to create with other development systems.

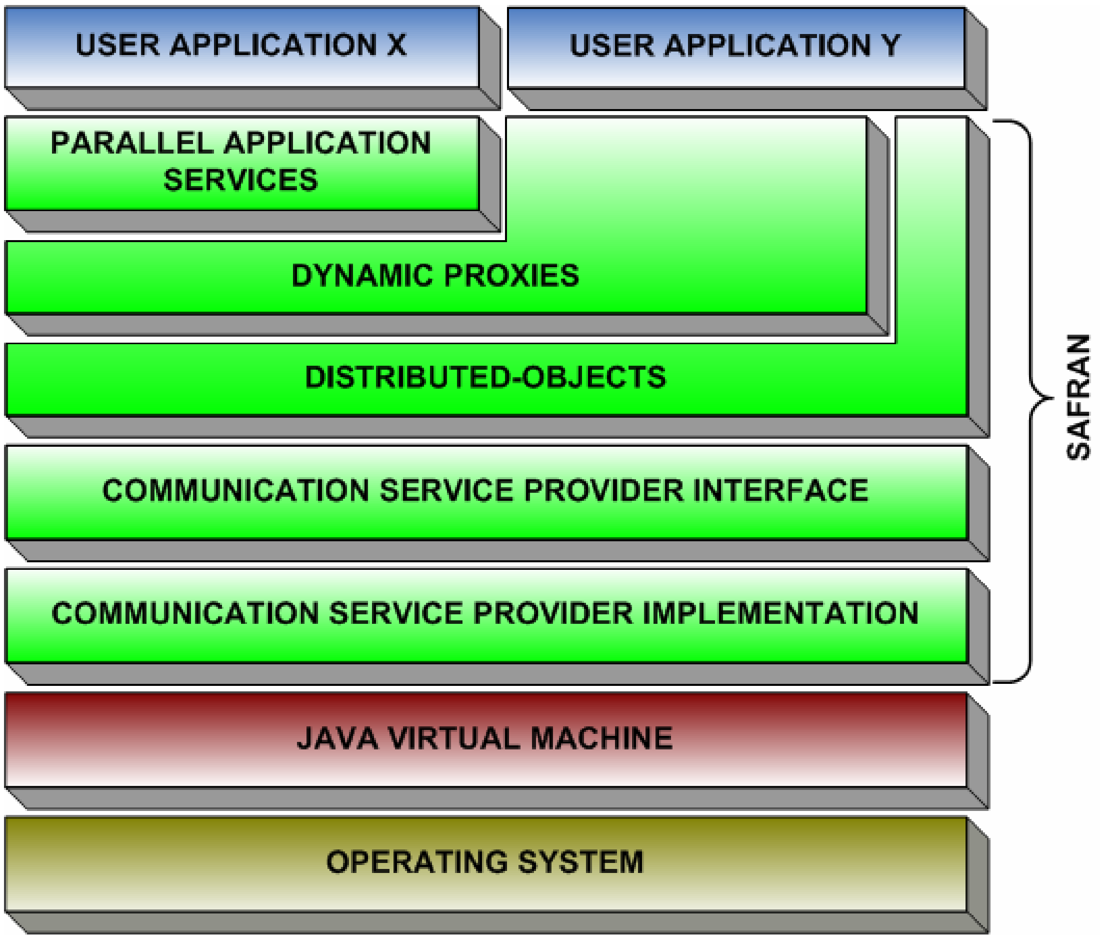
USER APPLICATION X
USER APPLICATION Y
PARALLEL APPLICATION SERVICES
DYNAMIC PROXIES
DISTRIBUTED-OBJECTS
COMMUNICATION SERVICE PROVIDER INTERFACE
COMMUNICATION SERVICE PROVIDER IMPLEMENTATION
JAVA VIRTUAL MACHINE
OPERATING SYSTEM
SAFRAN

Figure 3.2 Logical design of Safran's infrastructure.

The Java platform consists of a set of open specifications and standards which were initially developed by Sun Microsystems Inc. Sun Microsystems and many other commercial and non-commercial institutions provide mostly free and sometimes open-source implementations of these specifications for many different platforms.

The main specifications that define the Java platform are the Java Language Specification[31], Java Virtual Machine Specification[33], and the Java Core API Specification[35].

The Java Language Specification describes the Java programming language as a fourth generation, general purpose, high-level, object-oriented programming language. When a Java program is compiled, the Java compiler produces binary Java class files that contain Java byte-code instead of a native, platform specific executable binary. The Java byte-code is a platform independent, assembly like language that is design to be interpreted by a virtual machine. Java Virtual Machine Specification describes a stack based virtual machine to interpret the Java byte-code. There are many virtual machine implementations for many different platforms including general purpose operating systems, hand-held devices, and even embedded systems implemented as either software or directly in the hardware. A Java virtual machine sits on the native system (operating system or hardware) and executes standard Java byte-code. So when a Java program is compiled into Java byte-code, the same compiled byte-code can be executed without any modification on any system that provides a standard Java virtual machine. This capability of being able to run the same program in different platforms is referred to as *write-once run-everywhere*^{TM}. The Java Runtime Environment (JRE) is a runtime execution environment which loads and executes Java programs. JRE consists of the Java Virtual Machine and the standard set of core class libraries which are described by the Java Core API Specification. Together with the Java Virtual Machine Specification, the Java Core API Specification enables the portability and platform independence of Java programs.

The most important feature of the Java Platform for Safran is its support for portable and platform independent programs. As Safran is totally built on the Java Platform, it is platform independent and can be run on any operating system and machine architecture that provides a standard Java Runtime Environment. In other words Safran's support for heterogeneity of the systems it can run on is provided through the Java Virtual Machine abstraction. Safran itself does not do much to solve the heterogeneity problem: the Java Virtual Machine it runs on already abstracts away most of the heterogeneity problems.

3.4.3 Distributed-Objects Layer

The layer that we called the *Distributed-Objects* forms the core and backbone of the infrastructure of Safran. This layer provides services and abstractions to upper layers for doing distributed (remote) objects based programming. The low-level computational model explained in **Section 3.2.1** is supported by this layer and the corresponding programming model is provided mostly by the upper *Dynamic Proxies* layer.

The primary user of this layer and its services is Safran itself but the interface of the layer is also exposed publicly to user applications that might need some rarely used functionality of this layer such as making asynchronous or one-way method calls on remote objects. As depicted in the **Figure 3.2**, application types that are represented by **User Application Y** have direct access to Distributed-Objects layer. In fact, the interface and infrastructure provided is complete and general purpose enough that in addition to specific end-user applications, some other infrastructures and general purpose frameworks can even be built on this layer.

This layer basically extends the capabilities of RMI, Java's distributed-objects framework, with new distributed-objects based computing concepts and constructs. In addition to the functionality of exposing local objects to remote access, it supports creation of objects on remote machines, getting references to remote objects, calling methods on remote objects synchronously, asynchronously or one-way, check-pointing and migration of objects, and automatic and transparent distribution (dynamic class loading) of class definitions (binary class files) to remote machines.

The programming interface provided by this layer is fairly low level and is not much familiar. For example local references to remote objects are represented by instances of class `RemoteObjectRef`. Access to remote objects (calling methods on them remotely) is provided through the methods of this class. Once an instance of this class is obtained, calling methods on the remote object represented by this instance is done by calling `invokeMethod`, `invokeMethodAsync`, `invokeMethodOneWay` methods of the `RemoteObjectRef` class on the obtained instance. This programming interface is unfamiliar, difficult to use, and prone to errors but it provides all the primitives required for building higher level layers on top of it. The layer above, named Dynamic Proxies is built on this layer and provides a programming model that is much easier to use.

Although this layer and its programming interface are exposed to user applications, it is not intended for general and extensive use by end-user applications. It is mainly provided for applications that require access to some rarely needed but important functionality. For most of the user applications, some other programming interfaces that are much higher-level and much easy to use are provided through the Dynamic Proxies and the Parallel Application Services layers, which are built on the Distributed-Objects layer. So Distributed-Objects layer is mainly a backbone and abstraction for upper layers of Safran itself.

3.4.4 Communication SPI and Implementation Layers

The layers named *Communication Service Provider Interface (SPI)* and the *Network Communication Service* provide pluggable network transport and communication infrastructure for the upper Distributed-Objects layer.

Communication Service Provider Interface layer defines the interface (as a set of Java interfaces) that must be implemented and provided by the lower Network Communication Service layer. So it decouples the interface which is used by the upper Distributed-Objects layer to access to the network from the implementation of the network communication mechanism i.e. from the Network Communication Service layer. The Distributed-Objects layer has static dependency only to the Communication SPI layer and it's de-coupled and abstracted from the actual communication and transport mechanism implementation.

This design makes Distributed-Objects layer independent from the network communication infrastructure and technology. The Network Communication Service layer can be changed without effecting the design and interface of the Distributed-Objects layer. Different implementations of this layer based on different communication and network technologies can be developed and plugged into the infrastructure of Safran without affecting any of the upper layers and user applications build on them. For example, as part of this work we provide an implementation of Safran and in this implementation we chose to use Java RMI technology for our network transport and communication mechanism. So our implementation of the Network Communication Service layer is based on Java RMI technology. Safran's design is flexible enough that someone else can develop a different Network Communication Service layer using some other network communication technology (for example network sockets) and can plug it into our implementation of Safran without changing any other parts and layers of our implementation. Implementation of Safran can integrate with this new Network Communication Service layer and can use it transparently.

3.4.5 Dynamic Proxies Layer

The *Dynamic Proxies* layer builds on the Distributed-Objects layer and basically improves the programming model and interface provided by this layer. As explained in **Section 3.2.1** Distributed-Objects layer provides a programming interface that includes extensive low-level functionality for doing distributed-objects based programming. But the provided programming interface is not familiar and not easy to use. Programmers have to deal with instances of classes that locally represent remote objects i.e. the distinction between a local object reference and a remote object reference is explicit and visible to programmers and programmer must be aware of the distinction between a local object and a remote object.

Dynamic Proxies layer dynamically and transparently generates easy to use proxies for remote objects. Programmers program against these proxy objects that provide exactly the same public interface as the remote objects. So these proxy objects are used by upper layers to access and call methods of remote objects with the convenient and familiar local method call syntax (`obj.methodName(parameters)`) instead of dealing with remote object references that are error prone and difficult to use. When a method is called on a proxy object that locally represents a remote object, the method call is transparently dispatched to the remote object and the result of the method called is returned transparently to the caller. The method call on the remote object syntactically seems just like a local method call and remote objects are accessed and manipulated just like local objects.

To sum up, by providing a unified programming interface for interacting with local and remote objects, Dynamic Proxies layer makes distributed-objects based programming familiar and easy. Besides providing a programming interface to end-user applications, Dynamic Proxies also provides the infrastructure for the upper Parallel Application Services layer .i.e. Parallel Application Services layer is built on the Dynamic Proxies (see **Figure 3.2**).

3.4.6 Parallel Application Services

Parallel Application Services layer provides the infrastructure, abstractions and programming interface (API) for development of parallel and distributed applications based on high level concepts and constructs. The high-level computational model that is explained in detail in **Section 3.2.2** is implemented in this layer and the corresponding programming model is supported by the API provided by this layer to user applications. As depicted in the **Figure 3.2**, application types that are represented by **User Application X** are built on this layer. The architecture and entities such as Brokers and Hosts as described in **Section 3.2** and subsections is totally implemented in this layer. All high level sub-systems and services such as host registry, host monitoring, host allocation, and load balancing are implemented in this layer.

The layers and services of Safran that are implemented by Dynamic Proxies, and the other lower layers forms a quite general purpose infrastructure and framework (middleware) for building different kinds of distributed applications and systems. The Parallel Applications Services layer is built on this general purpose layer as another framework to support development of some completely different kind of applications based on the high level concepts and computational model it introduces.

3.4.7 User Applications

Safran is built on the Java platform. User applications must be developed using the Java programming language and the Java APIs provided by Safran for user applications. The kinds of applications that can be built on Safran are represented in **Figure 3.2** as **User Application X** and **User Application Y**. These actually represent the two main views or two main programming models of Safran presented to programmers.

The primary design requirement of Safran is to support applications of the type represented by User Application X. These applications are built based on the high-level concepts as explained in **Section 3.2.2**. These kind of applications run on the system architecture described in **Section 3.3**. The kind of applications represented by User Application Y is built on the lower-level concepts of distributed-objects as explained in **Section 3.2.1**. These applications do not require the infrastructure and architecture explained in **Section 3.3**.

CHAPTER 4

SYSTEM DESIGN AND IMPLEMENTATION DETAILS

In this chapter we provide detailed information about the internal design of Safran's layers and services. We do not provide complete design specification but instead try to give an insight about how Safran is internally architected.

4.1 Distributed-Objects Layer

As explained in **Section 3.4.5** Distributed-Objects layer, like CORBA and RMI, provides a computational and programming model based on the concept of distributed and remote objects. In this section, we provide design details of some important features of the Distributed-Objects layer and the layers below it.

Distributed-Objects layer mainly provides a client-server type programming model based on remotely accessible objects. In this model, we define **Client** and **Server** as follows: the processes that are hosting remotely accessible objects are called ***Servers*** and the processes that access the objects hosted by Servers and call methods on them are called ***Clients***.

Note that this classification of processes as clients and servers is based on the roles of the processes at a single time and context during their interaction. The actual and physical relation between processes can be symmetric and more complex: at some particular time and context a process might be hosting objects that other process access (thus it is acting as a server for other processes) and at some other time the same process might be accessing remote object in some other process (thus it acting as a client for those processes). In other words a single physical process does not need to be always acting as a client or a server i.e. it can be acting both as a server and a client at the same time by having a totally symmetric relation with other processes. So Distributed-Objects layer and the programming model do not impose the classical client-server type programming but in fact it allows arbitrary complex interaction between distributed entities.

Figure 4.1 depicts the main internal entities of Distributed-Objects layer in a server and a client process. These are conceptual representations and do not completely reflect exact internal implementation. Also note that these entities mostly correspond to Java classes and interfaces in Safran's internal design but they do not belong to the public programming interface (API) of the layer i.e. they belong to the internal infrastructure of the layer. In the following subsections, we briefly describe these entities and their relationship to each other.

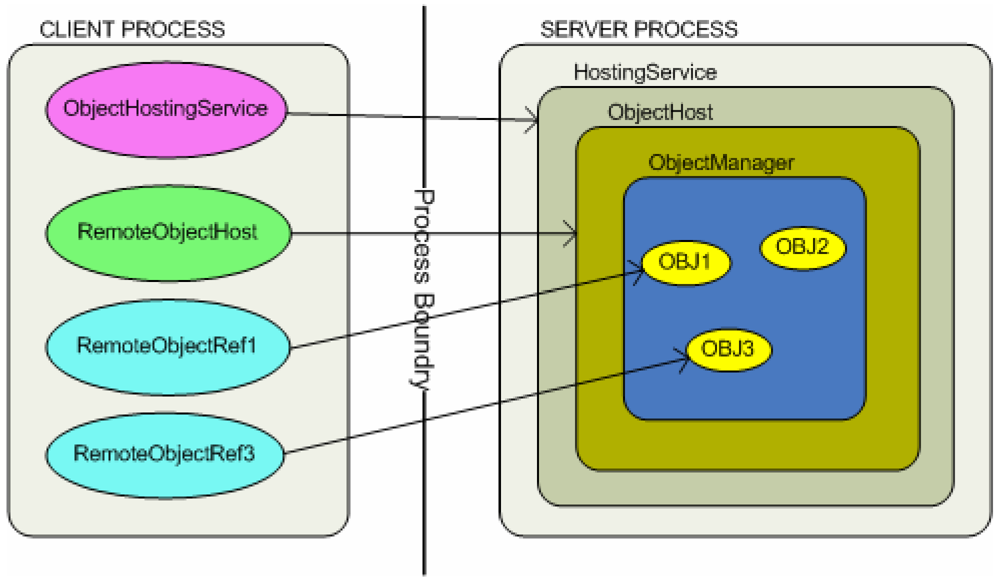

Figure 4.1 Internal entities of Distributed-Object layer

4.1.1 Server Side Objects and Entities

The server process only creates an entity named `HostingService`, which is provided by the Distributed-Objects framework, and just starts it. After the `HostingService` is created and started the process is ready to act as a server.

Remote clients can access the server process, create objects on it and can remotely call methods on the created objects.

When the `HostingService` is created by the server process it internally creates an entity named `ObjectHost` which actually hosts remotely accessible objects with the help of another internal entity named `ObjectManager` which manages the hosted objects. `ObjectHost` provides the actual service of hosting objects and enables the access of clients to hosted objects in various ways. For example, client processes can create objects in the server process, call methods on them, delete them, and get copies of them through `ObjectHost`.

4.1.2 Client Side Objects and Entities

At the client side, access to the server side entity `HostingService` is made through the client side object named `ObjectHostingService`. `ObjectHostingService` acts as the representative of `HostingService` in the client process and provides the interface for controlling a remote `HostingService` such as stopping it, getting its host name or port number.

Object named `RemoteObjectHost` provides client side access to the actual service (hosting remote objects) of the server side entity `HostingService`. In other words, it represents the server side entity `ObjectHost` at the client side by allowing client process to create objects on the server, calling methods on them and manipulating them in some other ways.

`RemoteObjectRef` objects are client side references to remotely hosted objects on the server process. For each remote object on the server process that is accessed by the client process, one and only one corresponding `RemoteObjectRef` object is created in the client process. Once a `RemoteObjectRef` object is obtained in the client process that references a remote object hosted by a server process, client process' subsequent interaction with the remote object is handled through this `RemoteObjectRef` object. In other words, `RemoteObjectRef` is the handle and gateway to the real remote object, which allows client process to access the remote object and to manipulate it.

The following figure is the Unified Modeling Language (UML) class diagram for major classes and interfaces of the Distributed-Object layer.

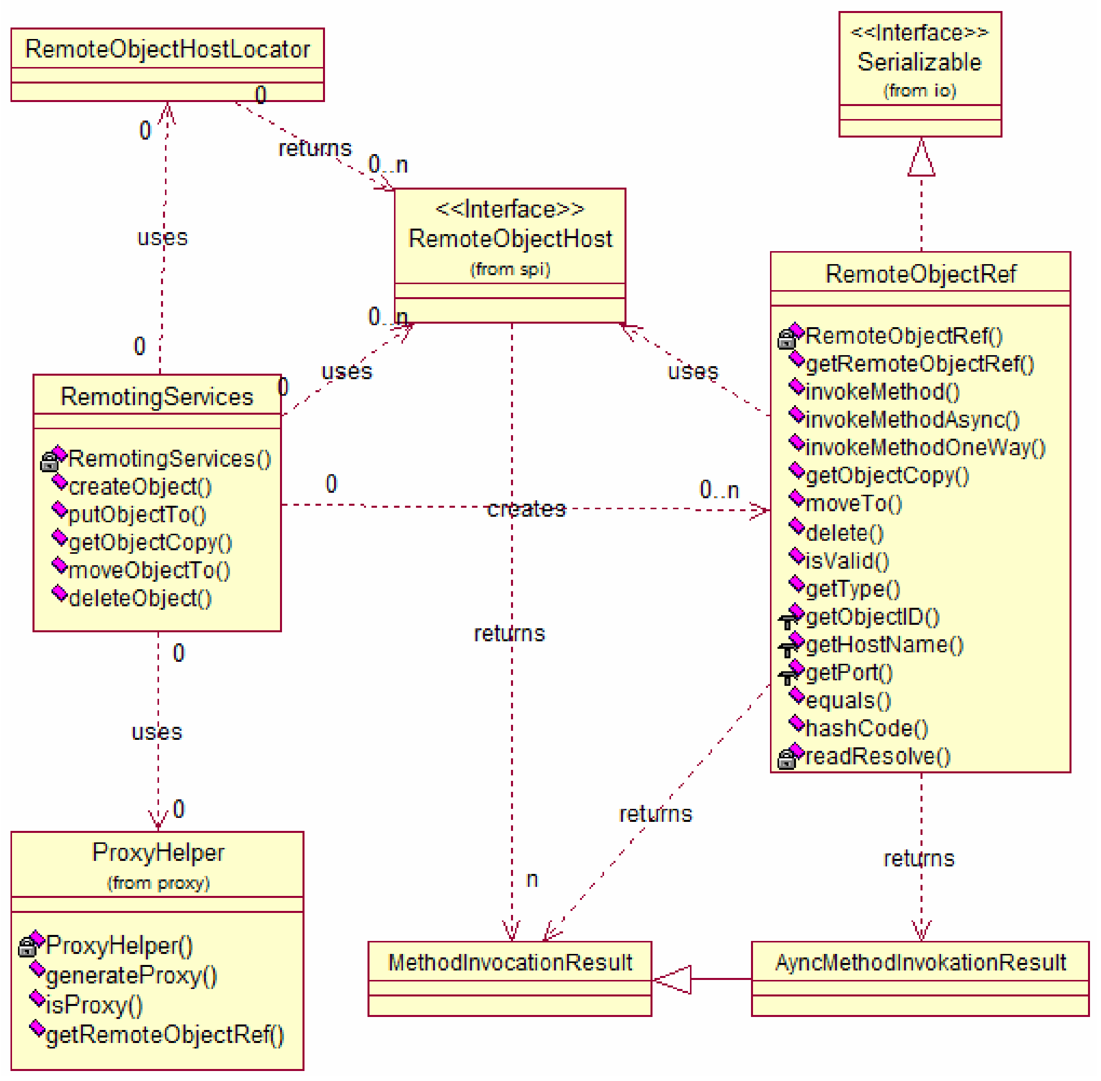

Figure 4.2 Distributed-Object layer UML class diagram.

4.2 Dynamic Proxy Generation

Users of the Distributed-Objects layer seldom need functionality such as synchronous and one-way method calls on remote objects which are provided only through the `RemoteObjectRef` objects. So, for such functionality users had to use complex, error prone, and difficult to use programming interface provided by this class. But for the major and extensively used functionality of making normal, synchronous method calls on remote objects, users are provided a much easy to use API with the help of Java Platform's dynamic proxy generation service. **Figure 4.2** depicts the relations between user code and runtime entities of the Distributed-Objects layer in a sample situation.

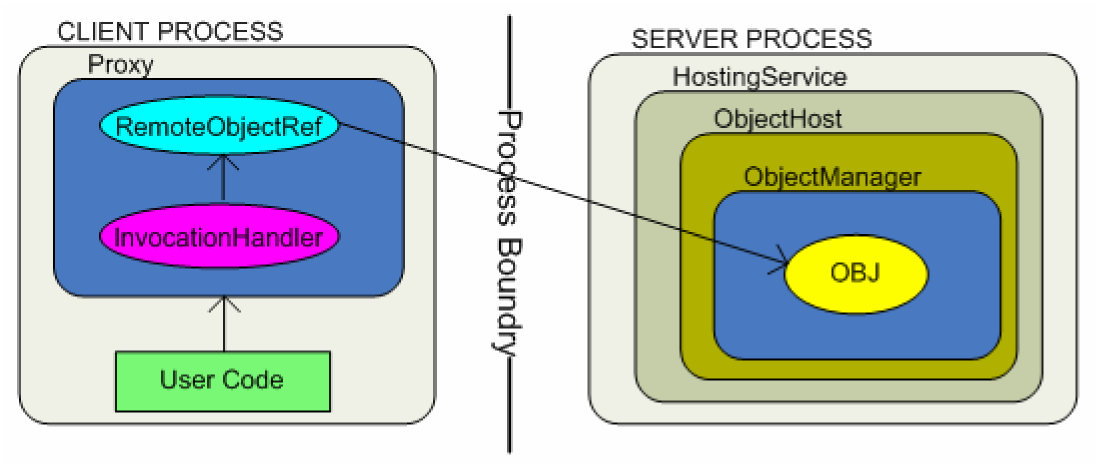

Figure 4.3 Relations between user code and Distributed-Objects layer entities.

When a user of Distributed-Objects layer creates an object on a remote process or accesses an exported object on a remote process, Distributed-Objects layer internally creates a `RemoteObjectRef` object that references the actual remote object. But the user is not given this `RemoteObjectRef` object. Instead, an instance of a dynamically generated proxy class that warps the `RemoteObjectRef` object is created and returned to the user. The created proxy class and its instance provide the exact same public interface as the remote object. In other words, the proxy object returned to the user implements the same set of Java interfaces that the remote object implements. So the user can cast down the proxy object to any of the interface supported by the remote object and call methods of the remote object on the proxy object. Internally the proxy object dispatches the methods called on it to the `RemoteObjectRef` object it wraps, which in tern dispatches the method calls to the actual remote object through the network. The return value and the result of the method calls on the remote object are returned back to the user backwards through the same path. Dynamically and transparently generated proxy objects provide some level of access and location transparency to the user of the Distributed-Objects layer. Users can call methods on remote object as if they were local objects.

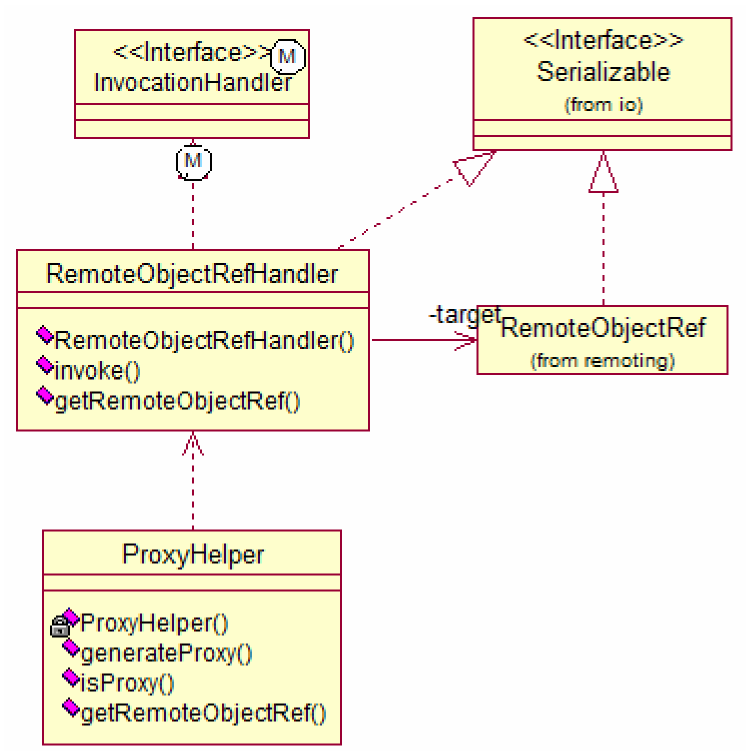

Figure 4.4 Dynamic proxy generation infrastructure of Safran.

Figure 4.4 is the Unified Modeling Language class diagram showing the dynamic proxy generation subsystem of Safran. Generation of dynamic proxies is provided with the help of Java's reflection and dynamic proxy generation functionality. First Distributed-Objects layer determines the list of Java interfaces implemented by the remote object using reflection. Next, it gives this list of interfaces and an `InvocationHandler` to the dynamic proxy API of Java. Java proxy system dynamically creates a class in the JVM that implements all of the provided interfaces and that dispatches the methods called on it to the provided `InvocationHandler`. An instance of this class is created and returned to the user. Whenever the user calls a method on the proxy object, the method call is dispatched to the `InvocationHandler` of the proxy, which in turn dispatch the method call to the wrapped `RemoteObjectRef`. Finally `RemoteObjectRef` dispatches the method call to the real remote object thought the underlying network and transport mechanism.

4.3 Pluggable Communication Infrastructure

Communication Service Provider Interface (SPI) layer basically defines the interface that must be provided by different implementations of the lower Network Communication Service layer. Distributed-Objects layer has static dependency only to this interface definition i.e. it does not have static dependency to the actual implementation of the network communication infrastructure. Communication SPI provides the infrastructure for decoupling Distributed-Objects layer from the underlying network communication mechanism so that the network communication and transport technology that underlies Safran can be changed if required.

Communication SPI layer consist of only four Java interface types that must be implemented by the lower Network Communication Service layer. Each interface defines the set of methods that must be implemented by the Network Communication Service implementations. So each different Network Communication Service implementation must at least provide four Java classes each implementing one of the Java interface defined by the Communication SPI layer. **Figure 4.5** is the UML class diagram of the Communication SPI.

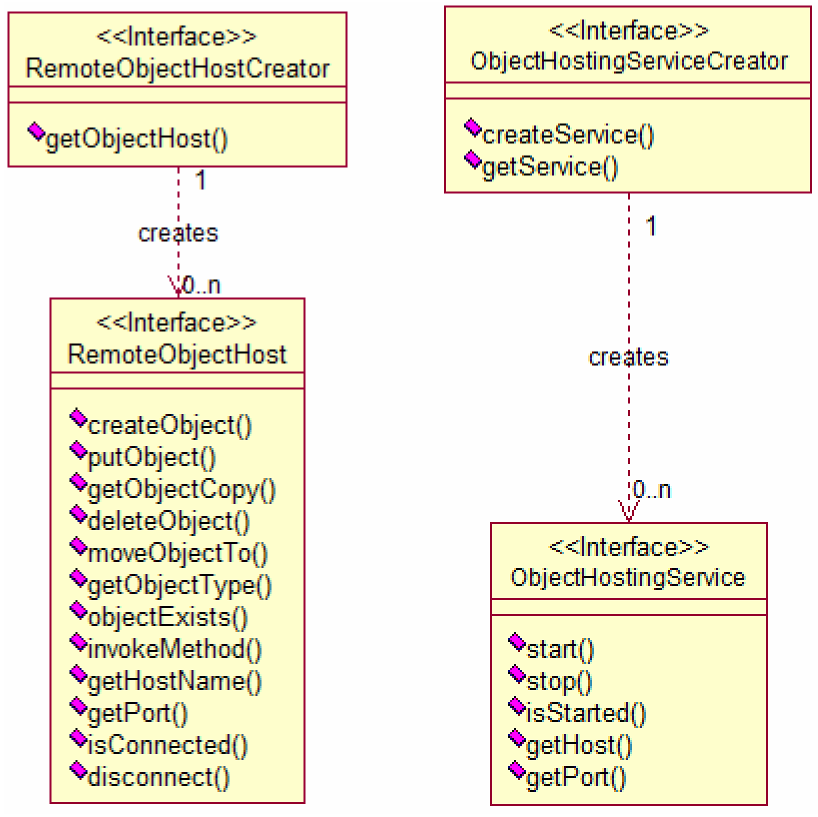

Figure 4.5 Class diagram for Communication SPI layer

Distributed-Objects layer has static (compile time) dependency to the Java interfaces, not to the actual implementation classes. So the Java classes that implement the interfaces and that actually provide the network communication mechanism are dynamically found and loaded by Distributed-Objects layer at runtime through a configuration file. Distributed-Objects layer's configuration file mainly contains the class names of the actual implementation classes. During initialization, Distributed-Objects layer reads its configuration file to find out the names of the classes that implement the underlying network communication mechanism. Next it loads the classes and creates instances of them using Java Reflection API. So the connection between the actual implementation of the Network Communication layer and Distributed-Objects layer is only the configuration file of the Distributed-Objects layer.

Figures 4.6 and **Figure 4.7** are the UML class diagrams for the parts of Distributed-Objects layer that enable plug-ability of the underlying communication mechanism.

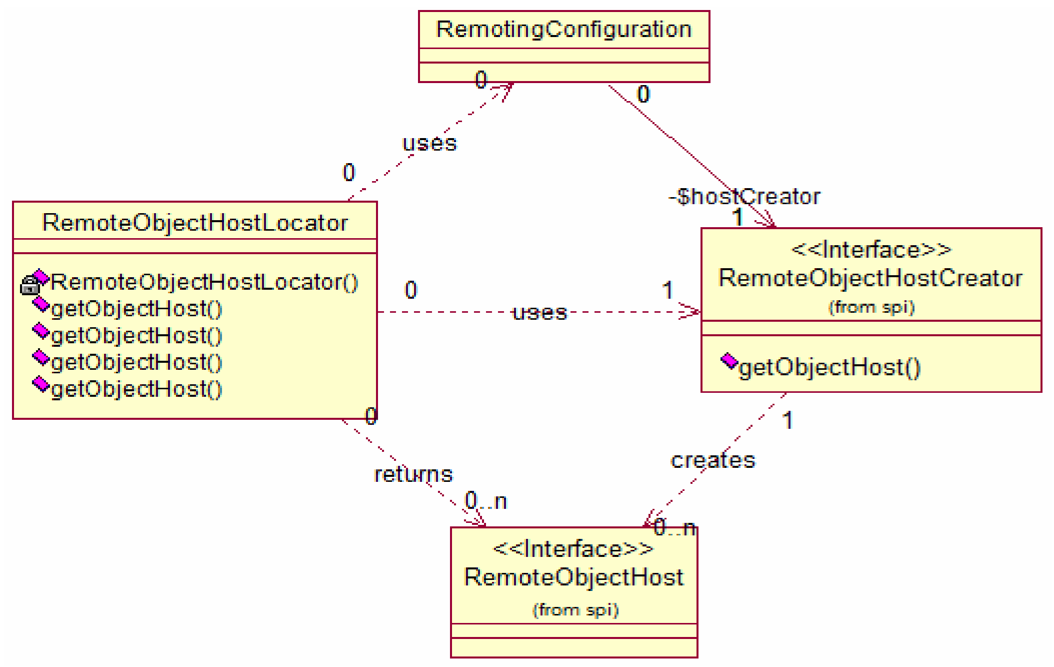

Figure 4.6 Class diagram of creation of `RemoteObjectHost`'s

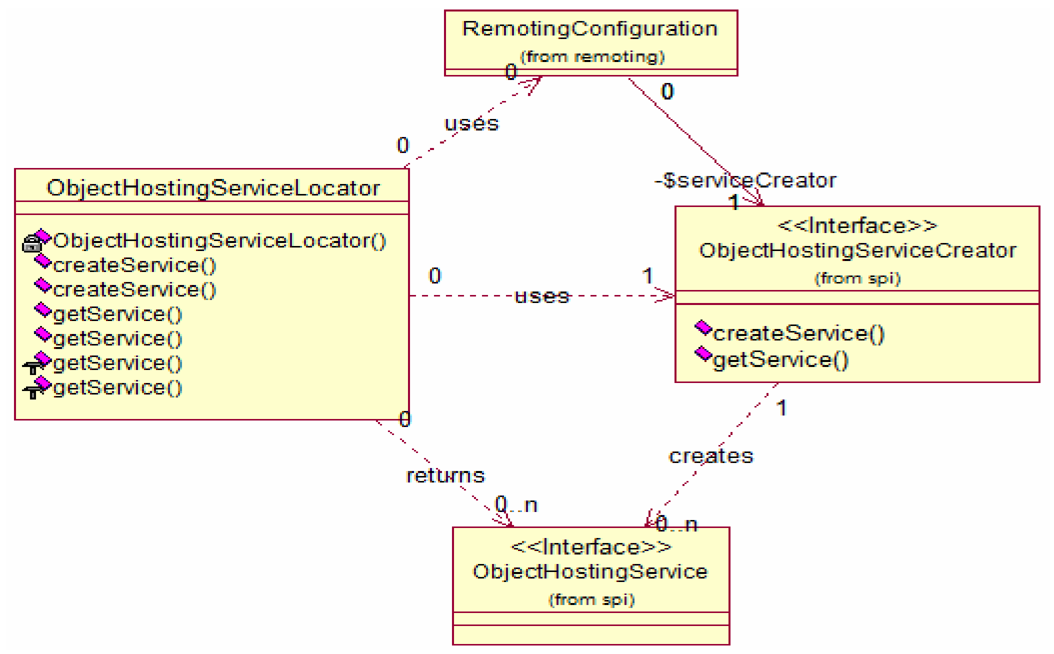

Figure 4.7 Class diagram for creation of `ObjectHostingService`'s

If the underlying Network Communication layer implementation needs to be replaced by some other implementation based on some other network transport technology, only the configuration file of the Distributed-Objects layer needs to be updated. Distributed-Objects layer implementation itself and the upper layers that depend on it do not need be changed in any way. So Distributed-Objects layer and upper layers are abstracted away from the underlying network communication technology and mechanisms in a pluggable way.

The Safran implementation that we provided with this work uses a Network Communication layer implementation based on Java RMI technology. Although the Safran implementation we provided runs on Java RMI (due to the Communication SPI) Distributed-Object layer, the layers above it and also the end-user applications do not have any static dependency to Java RMI. So, some other implementation (based on direct network sockets for example) of the Communication Services layer can be developed and plugged into our implementation of Safran to replace our Java RMI based Communication Services layer without affecting other parts of the Safran and applications built on it.

4.4 Parallel Application Services

Parallel Application Services layer mainly consists of two parts:

- A software infrastructure and a set of executable software that form and support the Safran system architecture which is explained in **Section 3.3**.
- A public software library that users use and program against to develop parallel and distributed application on Safran using the computational and programming model explained in **Section 3.2.2**.

Figure 4.8 is the class diagram showing the major classes that form the infrastructure of the Parallel Application Services layer.

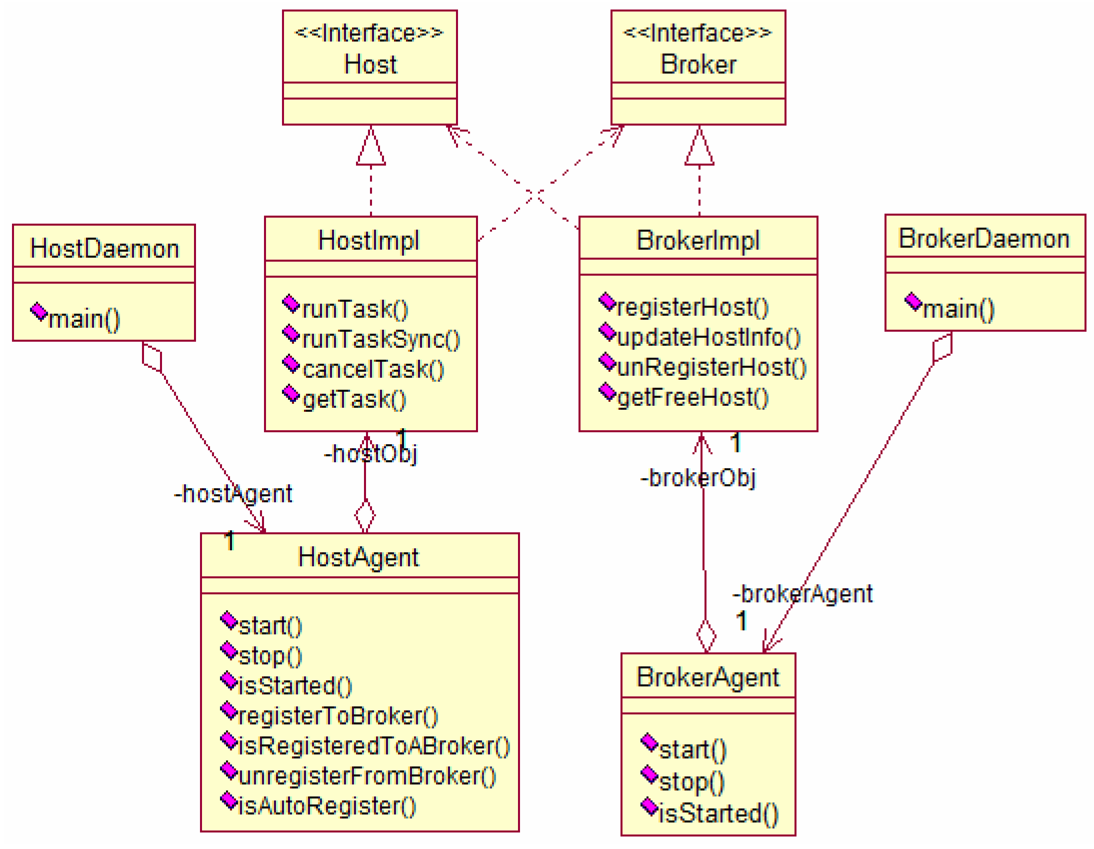

Figure 4.8 Class diagram for the infrastructure of Parallel Application Services

`HostDaemon` and `BrokerDaemon` are classes that implement the executable daemon processes (Hosts and Brokers). They simply create an instance of their corresponding agent classes (`HostAgent`, `BrokerAgent`), which actually provides almost all of the functionality of Hosts and Brokers. `HostDaemon` and `BrokerDaemon` are very simple executable classes that simply instantiate the functionality that is actually implemented in the `HostAgent` and `BrokerAgent` classes. `HostAgent` and `BrokerAgent` classes are publicly exposed classes, so users can actually implement their own daemon process instead of using `HostDaemon` and `BrokerDaemon` due to reasons like providing a better user interface and better integration with their specific platform's capabilities such as operating system's service or daemon process running capabilities.

`HostAgent` and `BrokerAgent` classes are the actual provider of the Broker and Host functionalities explained in **Section 3.3** and its sub-sections.

`HostAgent` and `BrokerAgent` communicate through Distributed-Objects layer as depicted in **Figure 3.2**. So Parallel Application Services layer is built on the Distributed-Object layer as another framework. `BrokerAgent` and `HostAgent` interact with each other by exposing remotely accessible objects to one other with the help of Distributed-Object layer. The class of the object exported by `BrokerAgent` is `BrokerImpl` which is accessed by remote `HostAgent`s (and Applications) through the `Broker` interface it implements. In the same way, the class of the object exported by `HostAgent` is `HostImpl` which is accessed by remote `BrokerAgent`s (and Applications) through the `Host` interface it implements. A `HostAgent` (or an Application) accesses the services of `BrokerAgent` by getting a `Broker` reference to the `BrokerImpl` object exported by the `BrokerAgent` to the Distributed-Object layer at a well-known port and with a well-known name. Similarly, a `BrokerAgent` (or an Application) accesses the services of HostAgent by getting a `Host` reference to the `HostImpl` object exported by the `HostAgent` to the Distributed-Object layer with a well-known name.

4.4.1 Broker and Host Relations

As explained in **Section 3.3** and its sub-sections, there is a registration relation between Hosts and Brokers: A group of Hosts register to a Broker to form a Safran cluster.

The `BrokerImpl` object which is contained and managed by `BrokerAgent` keeps a list of information about `Host`s registered to it. `BrokerImpl` assumes that the network is unreliable (network failures are possible) and Hosts that registered to it might fail without any notification. With this assumption of unreliable network and possibly failing `Host`s, `BrokerImpl` tries to keep the list of `Host`s registered to it as up-to-date as possible through the use of a protocol between itself and `Host`s. The main points of this protocol are as follows:

- Whenever a `Host` registers to a `BrokerImpl`, `Host` provides initial information about itself to the `BrokerImpl` during registration.
- During registration `Host` agrees to provide periodic notifications to the `BrokerImpl` so that `BrokerImpl` knows that the Host is still alive and it has up-to-date information about the `Host`. The period of notification is determined by the `Host` (constrained with a configurable minimum value) and provided to the `BrokerImpl` during registration.
- Other than periodic notifications, `Host` sends up-to-date information to the `BrokerImpl` that it has registered to whenever its state (capacity, availability etc.) changes.
- When a `Host` is shutting-down normally, it notifies the `BrokerImpl` it has registered to.
- `BrokerImpl` periodically checks the list of `Host`s registered to it. If a `Host` did not provided a notification for longer than the period it agreed, it is considered inaccessible and dead and it is removed from the list of registered `Host`s.

4.4.2 Locating Brokers and Automatic Registration

Sections 3.3 and **4.4.1** described the relations between entities like Brokers and Hosts but did not provided information on how these entities locate each other. Here, we explain the infrastructure used by different entities to automatically locate each other.

As explained earlier, Hosts need to register to a Broker to become part of a Safran Cluster. Also Applications need to contact a Broker to request Hosts to run Tasks on them. Hosts and Applications can be controlled and maintained manually to use a specific Broker running at a specific network address. Manual setup, control and maintenance become difficult as the number of the entities increases.

To enable easy and automatic setup and maintenance of a Safran Cluster, Parallel Application Services layer provides the infrastructure to enable Hosts and Applications to find and use Brokers automatically. When a Host starts-up it automatically locates (if configured to find and register automatically) a Broker and registers to it. In the same way, an Application can be configured to automatically find the Broker to use.

This sub-system of automatic discovery of Brokers is based on the TCP/IP multicasting technology. TCP/IP multicasting enables a group of hosts to join a multicast group and broadcast datagram packets to every member of the group, allowing group communication.

A class named `MessageBroadcaster` and its helper classes (`BroadcastMessage` and `MessageProcessor`) form general purpose multicast message broadcasting sub-system. This sub-system generally allows processes running on the same network send broadcast messages to each other without knowing each others network address. Message broadcasting sub-system is used by a singleton class named `BrokerLocator` to send and receive broadcast messages about the availability of the Brokers such that:

- When a `BrokerAgent` is started it registers itself to the `BrokerLocator` and `BrokerLocator` starts to send periodic broadcast messages to the network about the availability of the Broker as long as the Broker runs.
- When the `BrokerAgent` is stopped it un-registers itself from the `BrokerLocator` and `BrokeLocator` sends a message to the network about unavailability of the Broker.
- Hosts that are configured to find and register automatically, listen broadcast messages about availability of Brokers with the help of `BrokerLocator` object in their own process. `BrokeLocator` notifies the `HostAgent` whenever it determines that a new Broker has started or an existing one aborted so that

`HostAgent` automatically registers to a Broker or un-registers when the Broker it has registered is stopped.

- Whenever a `HostAgent` starts, it broadcasts a message to the network announcing that it has started and needs to know the Brokers currently running on the network.
- `BrokerLocator`s reply to the messages of newly started Hosts by broadcasting immediately the list of Brokers that registered to them.

The following figure is the UML class diagram of the classes of that support the message broadcasting and Broker locating sub-system of the Parallel Application Services layer.

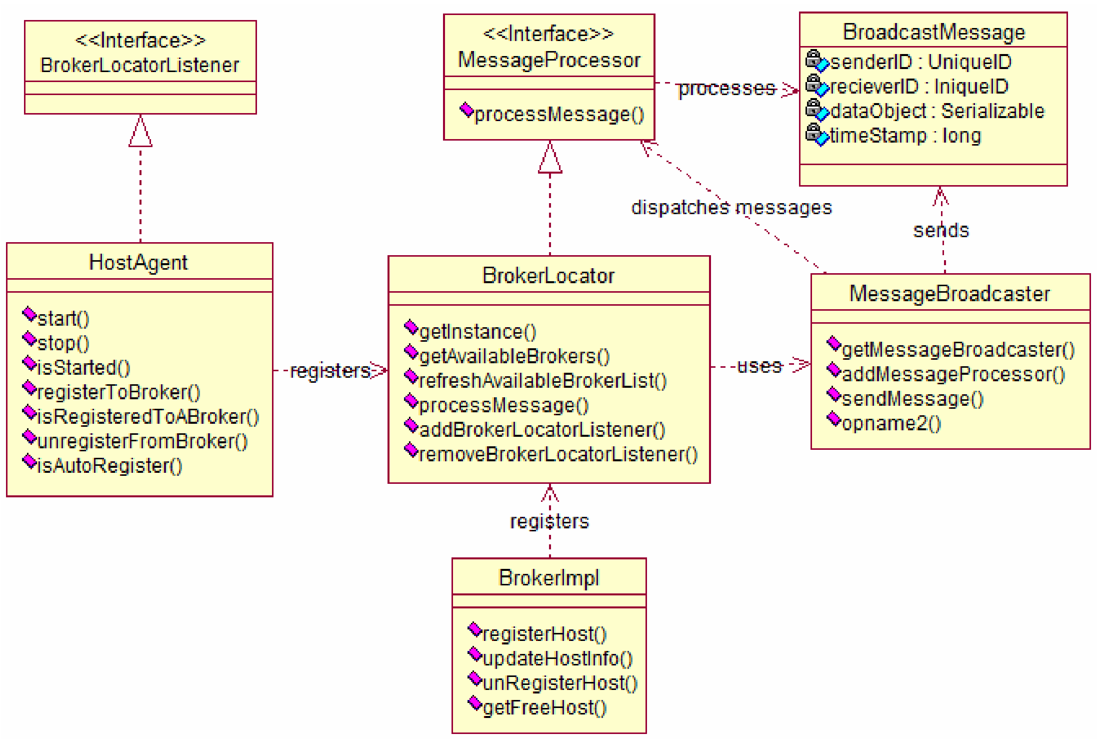

Figure 4.9 Message broadcasting and Broker locating sub-system class diagram

CHAPTER 5

TESTING AND EVALUATON OF SAFRAN

To test and evaluate Safran's design and implementation, we conducted an extensive set of experiments and we did extensive analysis on the result of these experiments. This chapter presents the experiments we conducted, the results we obtained, and our analysis and discussions on these results.

The objectives of these experiments are:

- End-to-end testing and demonstration of our implementation of Safran with a complete application.
- To determine performance characteristics (speedup, efficiency, overheads) of Safran for different sets of applications on different system configurations.
- To determine the level of convenience of Safran for different sets of applications.
- To find out design and implementation problems and deficiencies.

5.1 Experiments

To test Safran and collect data about its performance characteristics, we designed and implemented two different test applications. These two applications were run for a wide variety of system and application configurations using a test framework. The test framework automatically executed both applications several times for different configurations, collected performance data for each configuration and saved them to files for later analysis. The main data collected is the application completion times for each different configuration. Both of the applications are executed serially with no relation to Safran to determine serial execution times. Later these execution time data for serial executions and parallel executions for different configurations are analyzed to derive different performance information and reach general conclusions about performance characteristics of Safran.

5.1.1 Test Applications

Two different versions of the same problem are used for experiments and data collection: matrix multiplication and distributed data matrix multiplication.

Real Matrix Multiplication Application (RMMA): This application gets two randomly generated integer matrices *A* and *B* as input and calculates the multiplication of these matrices. When the application is run, to create *T* sub-tasks, the applications horizontally partitions the first matrix (A) into *T* sub-matrices each having *N/T* rows (where *N* is the row number of the first matrix). For each pair of matrices consisting of a sub-matrix of *A* and the whole matrix *B*, a Task object is created and submitted to Safran's application services. When executed, each Task calculates the matrix multiplication of a sub-matrix of *A* and the second matrix *B*. Safran dispatches all *T* Tasks to available remote Hosts, remotely executes them and get the results back. Note that this partitioning of input matrices for parallel computation is by no means a good way and much better algorithms for parallel matrix multiplication are available. Our purpose here is not to show the best possible way of parallel matrix multiplication on Safran. Instead, we tried to create an application that has the characteristics of many other applications, so that we can use the results of our experiments for this application to reach general conclusions for the set of applications that have similar characteristics. This application is characterized of being both computation (CPU) intensive and communication (I/O) intensive. The algorithm used by Tasks to multiply two matrices is naïve (of complexity $O(n^3)$). The partitioning method used to create the Tasks requires the transmission of second matrix (*B*) as a whole to remote Hosts which makes this version extensively I/O intensive. The communication requirement of the application for a *T* task run is of complexity $O(Tn^2)$. So, this application is a good representative of both CPU and I/O intensive applications.

Distributed-Data Matrix Multiplication Application (DDMMA): This version is designed as a representative of pure CPU intensive applications. The version simulates parallel matrix multiplication operation by using the exact same matrix multiplication algorithm that is used by the Real Matrix Multiplication Application. The difference is that, the matrix data on which the algorithm operates on is not transmitted over the network to the remote Hosts that execute the Tasks. Instead, when a Task is dispatched to a remote Host (without matrix data) the matrix multiplication algorithm operates on locally generated random matrix data. This way, we avoid the communication and network I/O overhead of the Real Matrix Multiplication Application but have an application that has exactly the same computational characteristics.

The only difference between Real Matrix Multiplication Application and Distributed Data Matrix Multiplication Application is that, one has extensive network I/O overhead and the other does not. We could have used some different application representing pure CPU bound applications instead of Distributed Data Matrix Multiplication Application. But we choose to simulate the matrix multiplication algorithm because both Real Matrix Multiplication Application and Distributed Data Matrix Multiplication Application have the exact same computational properties which enabled us to do meaningful comparison between the results of these two applications and clearly see the effects of I/O overhead.

5.1.2 Experimental Setup

All the experiments are performed on identically configured Sun Blade 2000 workstations on a 100 Mbps, general purpose, corporate office LAN. There were about 40 of these workstations but at most 10 of them were used during our experiments. The workstations (and the network) were not exclusively used for our experiments as they are owned by different people and used for different purposes. They were running different applications and services as the experiments are performed but they were lightly loaded because all the experiments are performed after office hours (i.e. they were not being used heavily). So, the available capacity and load of the workstations used during the experiments were slightly differing from each other. Details of the configuration of each of these machines are:

- Running Sun Solaris OS Release 5.8
- 1200 MHz Sun UltraSPARC-III+ CPU
- 1024 MB physical memory
- 1341 MB virtual (swap) memory on a UFS mount.

For all experiments Java 2 Runtime Environment (JRE) Standard Edition build 1.5.0_01-b8 of Sun Microsystems for Sun Solaris OS is used. JVMs are started in the client (Java HotSpot™ Client VM) mode with all other startup parameters left default. During all experiments each entity of Safran (Broker Agent, Application, and Host Agents) is run on a different machine exclusively.

5.1.3 Experiment Configurations

Each of the two applications is executed on Safran for a wide range of configurations to obtain a detailed understanding of performance characteristic of the platform for different kinds of system and application configurations and for different types of applications.

For all experiments involving Safran (parallel executions), only one Broker Agent is used. Each entity (Broker Agent, Application, and Host Agents) is executed on its own machine exclusively.

Serial execution experiments are run locally on a single machine with no involvement of Safran entities. Computation is done locally in a single process that does not use Safran in any way.

The following table show the matrix of different configurations for which experiment conducted.

	Serial Execution	**Parallel Execution**
Host Number	1. Executed locally on a single machine without involvement of Safran.	1, 2, 4, 8
Problem Size	256x256 384x384 512x512 640x640 768x768 896x896 1024x1024 1152x1152	256x256 384x384 512x512 640x640 768x768 896x896 1024x1024 1152x1152
Task Number	1 (i.e. the computation is done as a whole)	1, 2, 4, 8, 16, 32

For each different configuration (e.g. 2 Hosts, 8 Tasks, 896x896 Matrix) applications are executed 5 times repeatedly and average execution time is calculated. For every pair of Host number and Task number result are written to files. For example, the contents of the result file of a test with Real Matrix Multiplication Application on 4 Hosts and for 8 Tasks are as follows:

```
Test Name:   Real Matrix Multiplication
Host No:    4
Task No:    8

Host No     Task No     Matrix Size      Execution Time (ms)
4           8           128              412
4           8           256              1153
4           8           384              1927
4           8           512              3338
4           8           640              5969
4           8           768              10022
4           8           896              14647
4           8           1024             21750
4           8           1152             31976
```

The results of all experiment are transferred from results files to spreadsheets and analyzed, whose results are presented in **Section 5.2**.

5.2 Experiment Results

In this section, we present the results we obtained, the analysis we made on the results and conclusions we are led by our analysis.

During experiment, the only performance metric collected is the application completion time which is the difference between the time the application is submitted to the system and the time the system returns back the application as completed. Note that this time measure includes both application's computation time as well as the overhead time incurred by system and network communication. We analyzed this raw data for various configurations to derive more comprehensible performance metrics such as speedup, system efficiency and scalability measures.

Speedup is the measure of the improvement (i.e. reduction) in an application's execution time when run on more than one machine using parallelization of the computation. To be able to calculate speedup values we run both of our applications serially on one machine using the same exact algorithms. Then, speedup values are calculated as:

$$S_N = \frac{T_{serial}}{T_N}$$

where T_{serial} is the serial execution time of the application and T_N is the execution time of the same application on a Safran cluster of *N* Hosts.

Another analysis we did is system efficiency analysis which we define as the usage percentage of the system's theoretical computational capacity. For example, on a 4 Host Safran system, if we are achieving a 3.5 speedup value, then the efficiency of the system usage is $(3.5/4)\cdot 100 = 87.5\%$. So efficient values are calculated as:

$$E_N = (S_{N-\exp erimental} / S_{N-theoritical}) \cdot 100$$

Theoretical speedup values are the theoretically maximum speedup values achievable on the system and are equal to the number of Host on the system.

Lastly, we made some theoretical scalability analysis on our results. Scalability is the measure of the change in the effectiveness of the system when the size of the problem or the size of the system (Host number) is increased. This analysis enabled us to deduce whether or not Safran is suitable for much larger applications and for configurations for which we were not able to conduct experiments.

5.2.1 Real Matrix Multiplication Application Tests

We run the real matrix multiplication application serially and for 1, 2, 4, and 8 Host and for 1, 2, 4, 8, 16, and 32 Task Safran configurations. In the following, we present the discussion only for the 1 Host and 8 Host Safran configurations. The analysis for the 2 Host and 4 Host configurations will be presented together with the other cases in the overall summary section.

Safran Configuration of 1 Host

The execution time vs. the problem size graph for the 1 Host and different Task numbers case is given **Figure 5.1**.

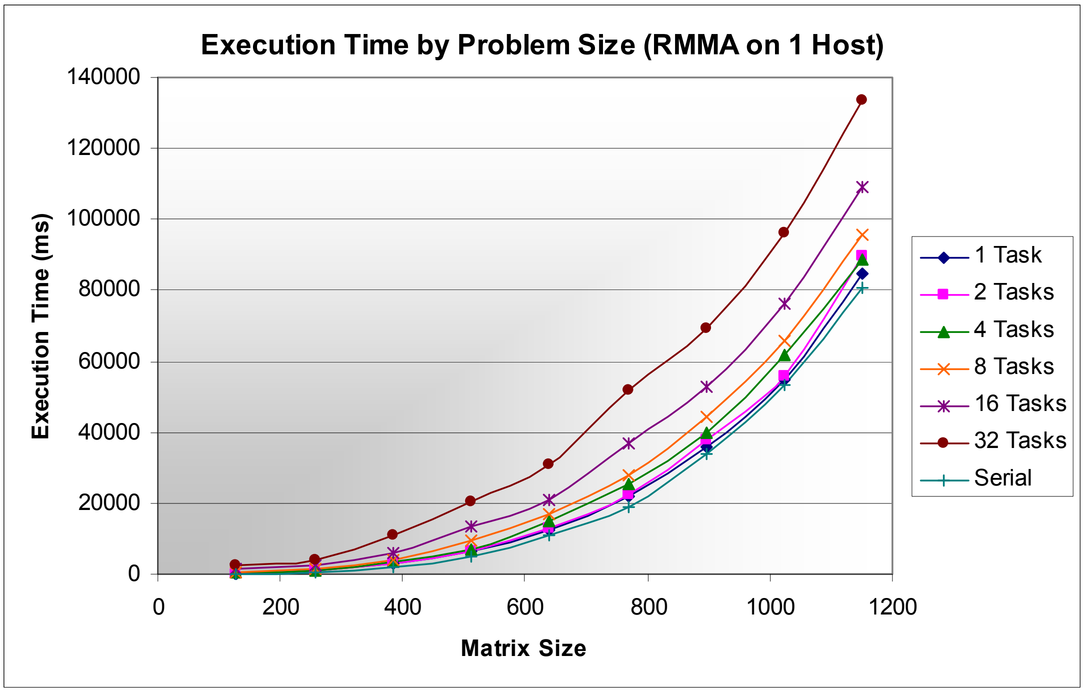

Figure 5.1 Problem size vs. execution time graph for 1 Host RMMA tests

The polynomial increase in the execution time is expected and is due to the matrix multiplication algorithm used, which is an $O(n^3)$ complexity algorithm. Execution time also increases as the Task number is increased and for all configuration of Task number, it is larger than the serial execution case. This is due to the fact that we do not have any parallelization gain (as we have only one Host) but have overheads of the Safran system. The graph suggests that the overhead increases by the Task number. The graph in **Figure 5.2** more clearly shows the affect of Task number on the execution time and overheads, for the same configuration.

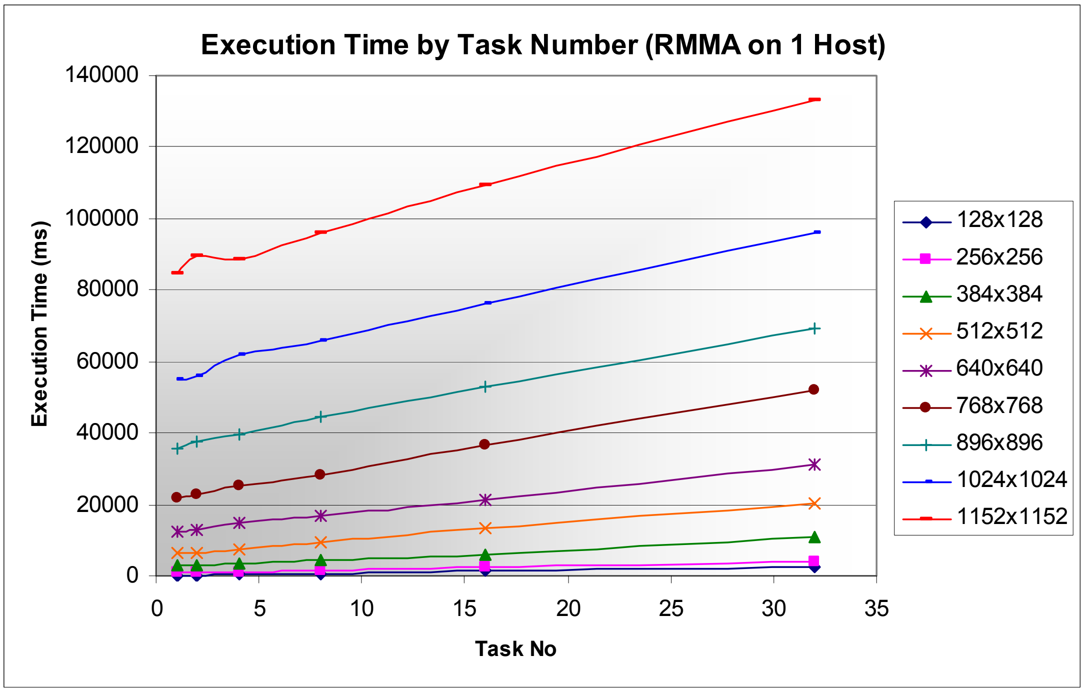

Figure 5.2 Task number vs. execution time graph for 1 Host RMMA tests

It is clearly seen that the execution time for all matrix sizes increases linearly by the Task number. Also note that the rate of increase (slope of the line) in execution time gets larger as the matrix size increases. For example, for the 11152x1152 matrix case, difference between 1 Task execution time and 32 Task execution time is about 52 seconds which is the overhead incurred by increasing Task number. We suggest that almost all of this overhead is due to network communication. The matrix partitioning method we used requires the transmission of whole second matrix with each Task over the network. For example, the size of the data transferred for the 1152x1152 matrix reaches about 334 Mbytes during the execution of the application.

The following speedup graphs (**Figure 5.3** and **Figure 5.4**) suggest that the matrix size (problem size) affects the speedup positively and task number (for 1 Host case) affects speedup negatively.

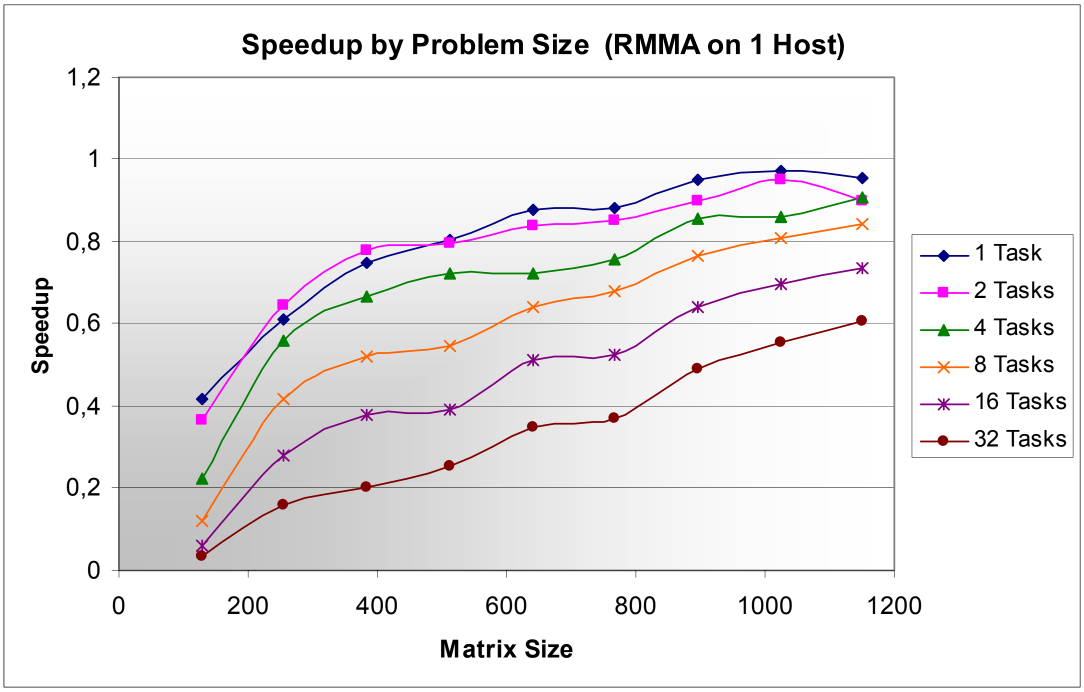

Figure 5.3 Problem size vs. speedup graph for 1 Host RMMA tests

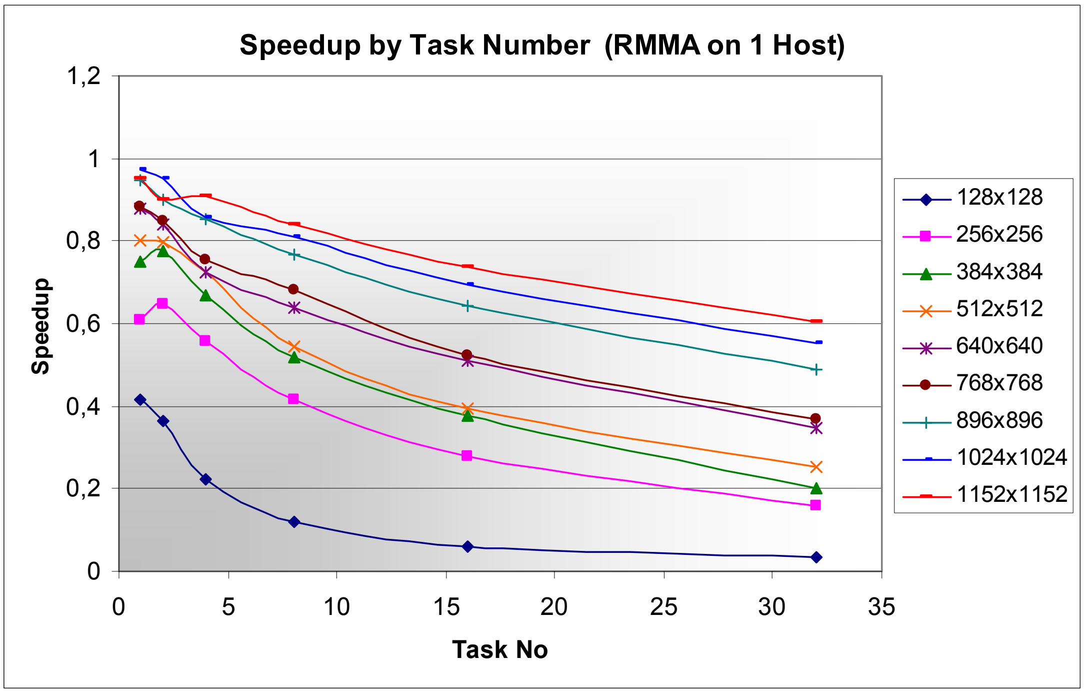

Figure 5.4 Task number vs. speedup graph for 1 Host RMMA tests

Larger matrix sizes produce better speedups because as the problem size increases, overhead time gets relatively smaller compared to the execution time (because relative increase in the overhead is smaller than the relative increase in the execution time). As explained before, the computational complexity of the application is $O(n^3)$ and communication requirement complexity is $O(Tn^2)$ (the T is Task number). So, as the problem size increases the computation time becomes much dominant in the total execution time compared to the overhead time incurred by network communication.

The following graph (**Figure 5.5**) clearly shows the relation of total overhead to the Task number.

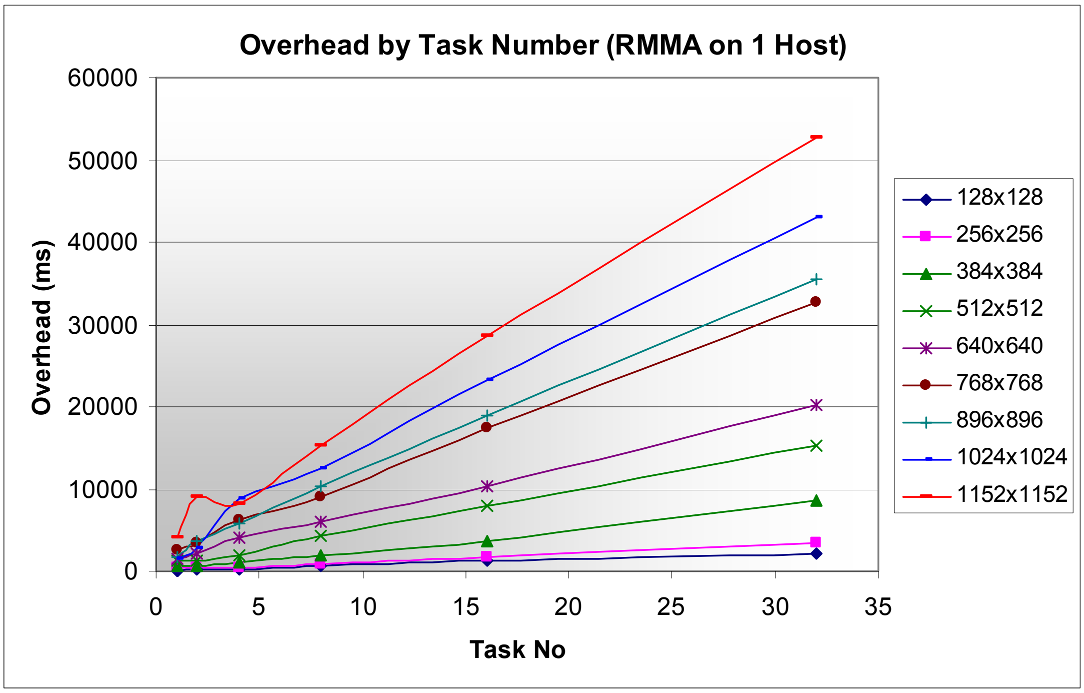

Figure 5.5 Task number vs. overhead time graph for 1 Host RMMA tests

Overhead times are calculated as the difference between the execution time of 1 Host Safran configuration and the serial execution. As expected, the overhead time increases linearly by the Task number. We obtain these clear linear graphs because system overheads other than the communication overhead is negligibly smaller compared to the communication overheads incurred by increasing Task number.

Safran Configuration of 8 Hosts

The following two graphs (**Figure 5.6** and **Figure 5.7**) show the variation of execution time of RMMA on an 8 Host Safran configuration for different problem sizes and Task numbers.

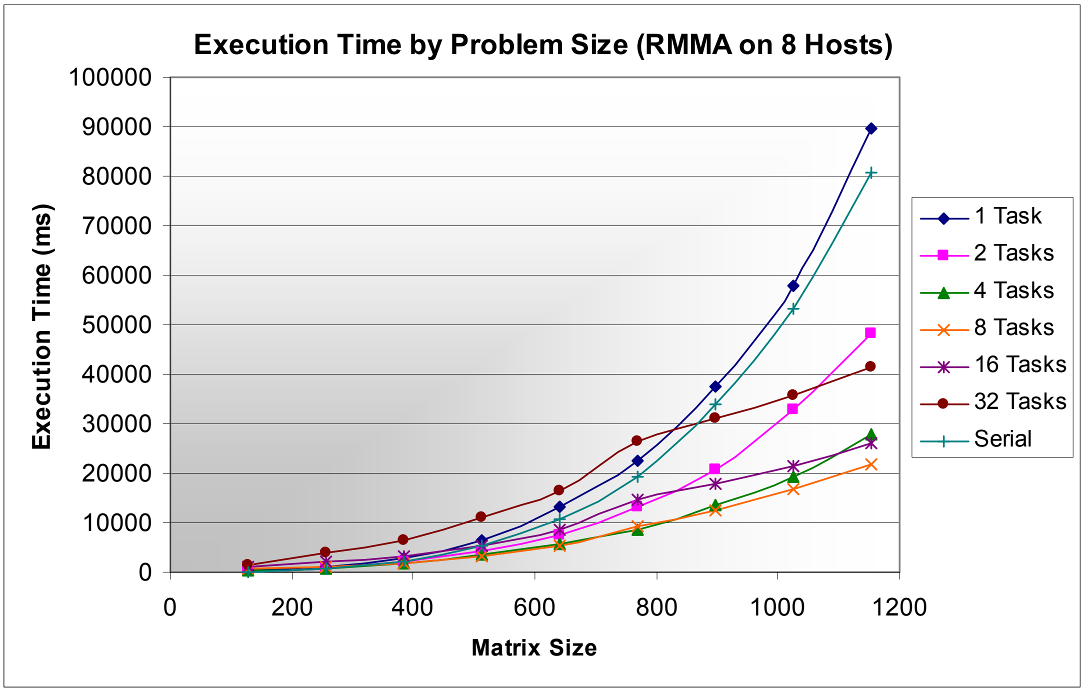

Figure 5.6 Problem size vs. execution time graph for 8 Host RMMA tests

It is seen that for Safran configurations of 8 Hosts the optimum value of Task number (minimum execution time) for almost all problem sizes is 8, for which the whole problem is divided into number of sub-problems that is equal to the machine number. Execution time decreases as the Task number increased from 1 to 8, but begins to increase after this point (16 and 32 Tasks). This is because dividing the problem into more sub-problems does not help further in parallelizing the execution as we have limited number of machines (8 in this case). Instead it adds unnecessary overhead which increases the total execution time.

The following graph (**Figure 5.7**) shows the effect of Task number on the execution time.

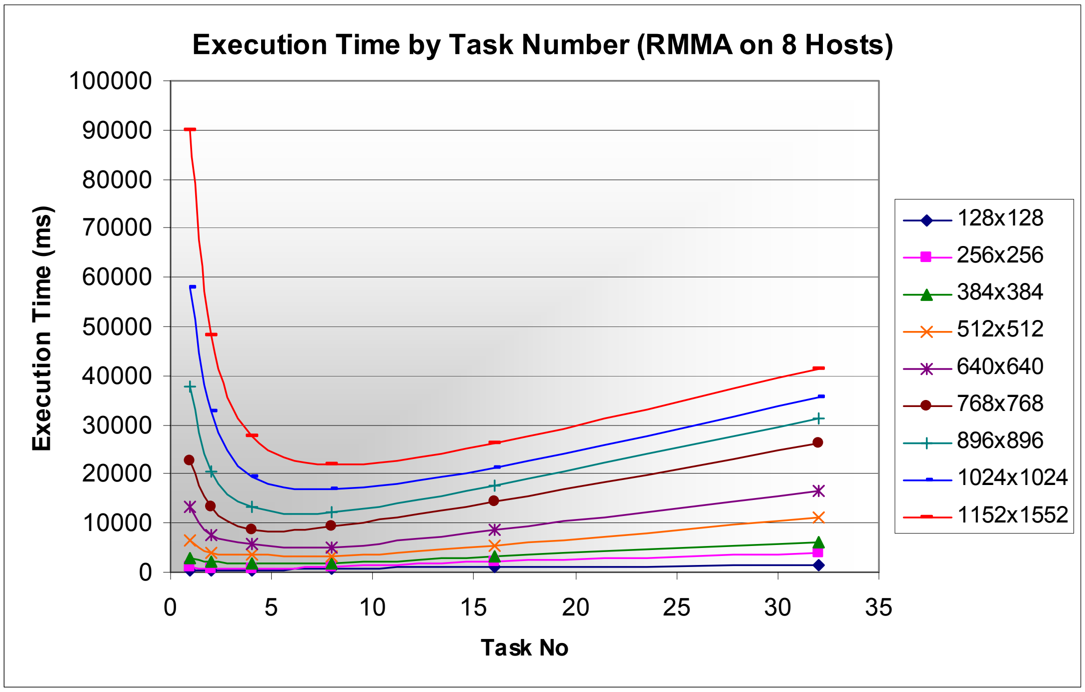

Figure 5.7 Task number vs. execution time graph for 8 Host RMMA tests

The minimum execution time for all sufficiently large problem sizes is achieved with the task number equal to the number of Hosts (i.e. 8). After this threshold the execution time begins to increase linearly by the Task number because increasing Task number does not help in parallelizing the execution but instead adds huge network communication overhead.

The following two graphs (**Figure 5.8** and **Figure 5.9**) show the variation of speedup achieved for real matrix multiplication application on an 8 Host Safran configuration for different problem sizes and Task numbers.

Figure 5.8 suggests that the best speedup is achieved when the Task number is equal to the Host number. Speedup improves as the Task number is increased from 1 to 8, but begins the decrease for future increase in Task number. Generally, as the size of the problem increases the speedup increases and for sufficiently large problem sizes it begins to converge to a value. We see this convergence characteristic for 1, 2 and 4 Task configurations (**Figure 5.8**) but it is not visible for the 8-Task configuration still for the largest problem size (1152x1152) we used. This means that we can expect some further improvement in the speedup for 8-Task configuration and for the problem sizes larger than 1152x1152.

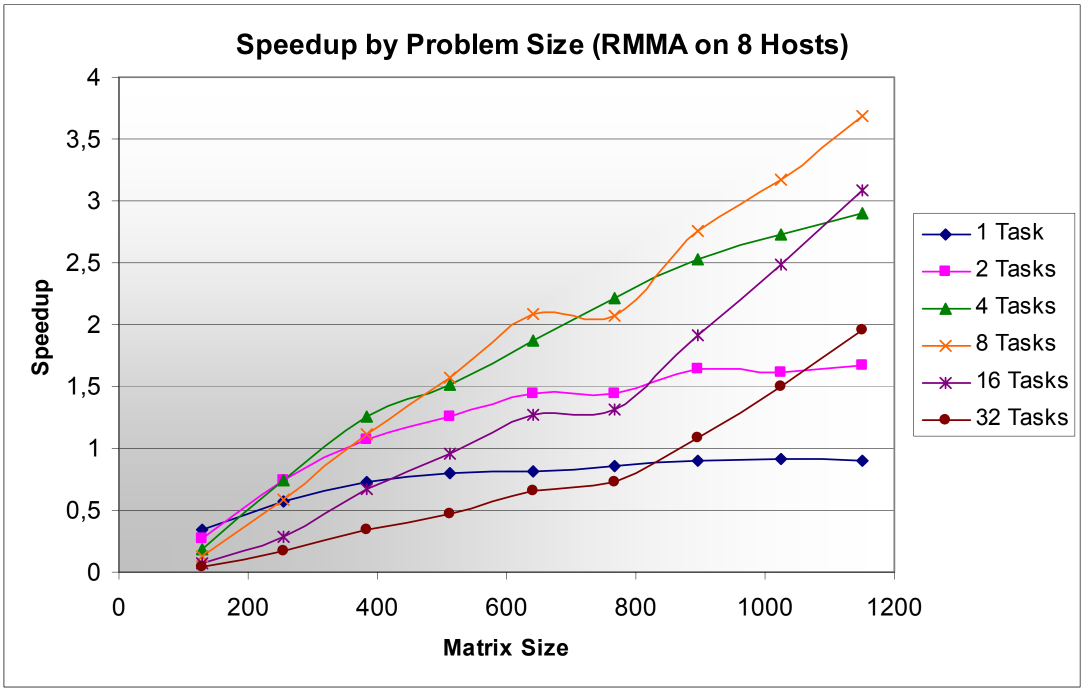

Figure 5.8 Problem size vs. speedup graph for 8 Host RMMA tests

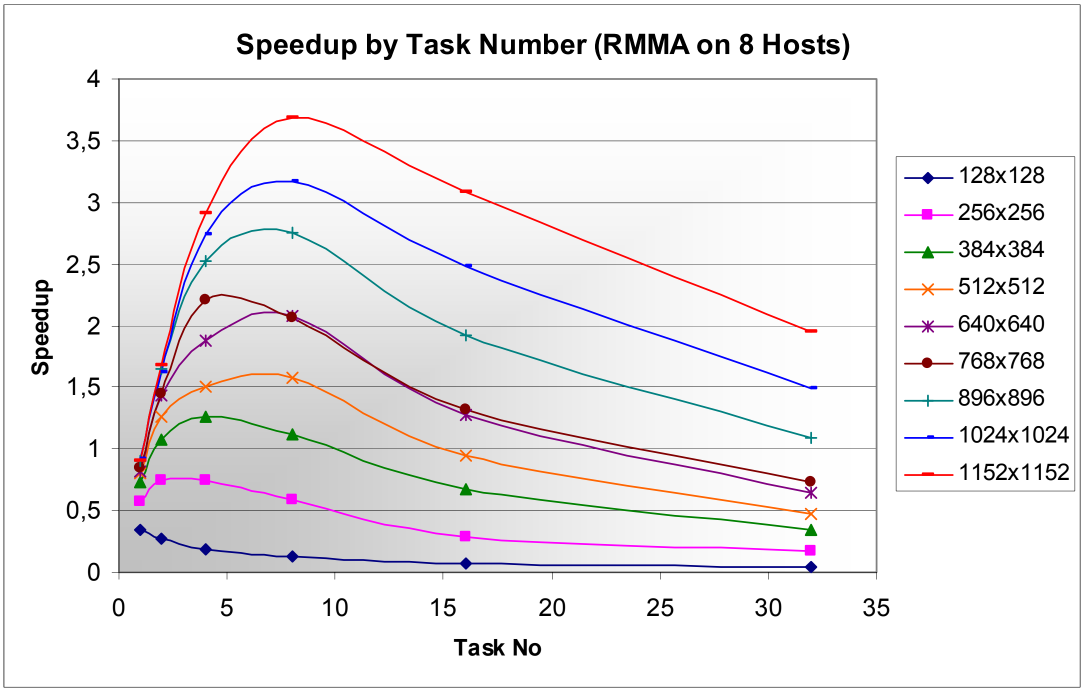

Figure 5.9 Task number vs. speedup graph for 8 Host RMMA tests

Figure 5.9 shows how speedup changes by Task number. It's clearly seen that best speedup is achieved at the optimum task number of 8. For larger Task numbers the speedup degrades rapidly because more Tasks add huge communication overhead. Note that, the rate of degradation in the speedup is bigger for larger problem sizes because the cause of this degradation is communication overheads, which totally depends on the size of the second matrix (*B*).

Summary of the RMMA Tests

Here we present the overall analysis for all (1, 2, 4, 8 Host) configurations of Safran for real matrix multiplication application.

The **Figure 5.10** and **Figure 5.11** show how maximum speedups achieved for 1, 2, 4, and 8 Host configurations change depending on the Task number and the problem size respectively.

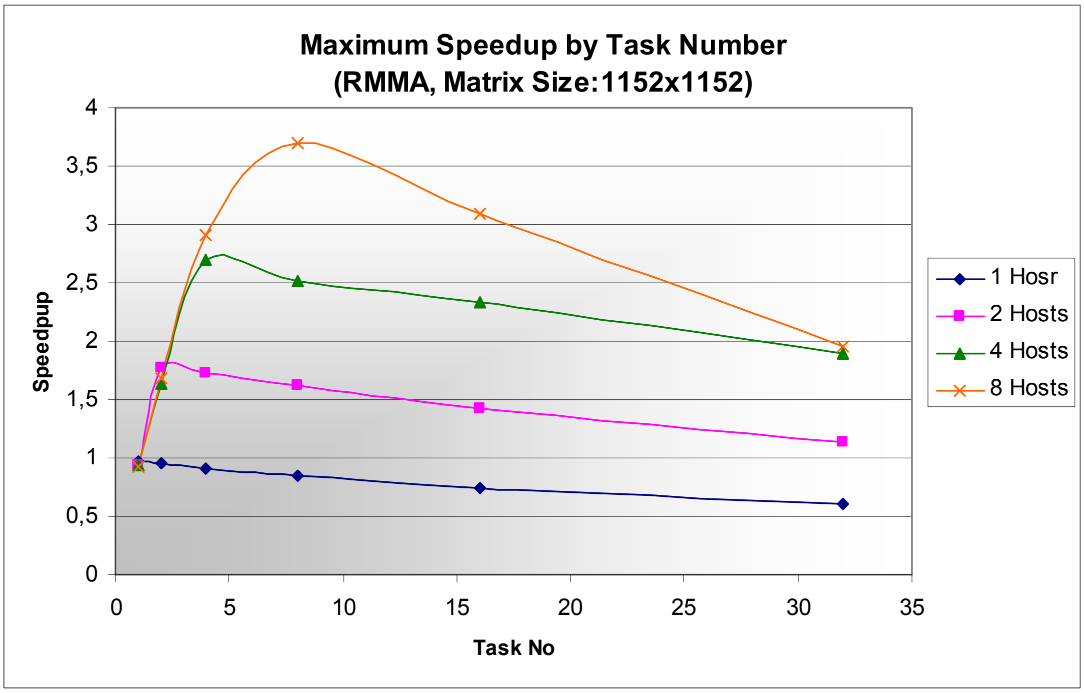

Figure 5.10 Task number vs. maximum speedups achieved for all RMMA tests

Figure 5.10 suggests that for each configuration maximum speedup is achieved for the Task number that is equal to the Host number. Larger Task numbers cause degradation in the speedup which is proportional to the problem size. **Figure 5.11** suggests that speedup generally improves by the problem size but converges to a value for sufficiently larger problems.

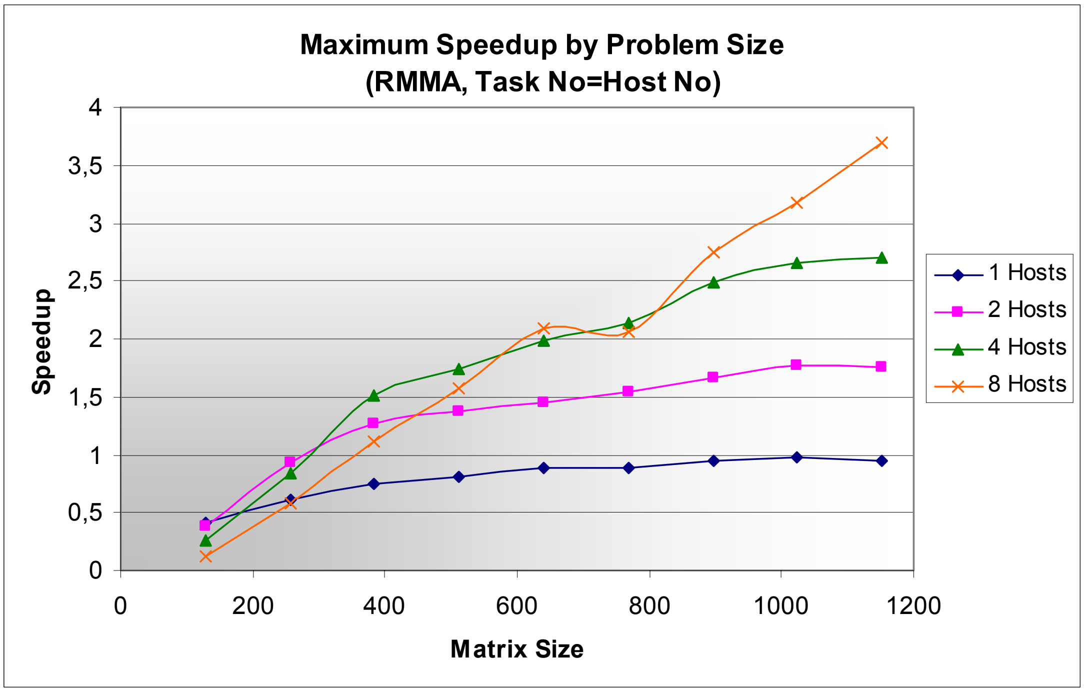

Figure 5.11 Problem size vs. maximum speedups achieved for all RMMA tests

Figure 5.12 shows how maximum speedups achieved during all experiment change by Host number. It seems that the maximum speedup achievable on a Safran system is logarithmical proportional to the Host number. **Figure 5.12** suggests that our system does not scale very well for the set of applications represented by the real matrix multiplication application (i.e. for applications that require massive network I/O and communication) because increasing Host number does not produce proportional speedup gains. For example, for the 8 Host configuration, the maximum speedup achieved is only about 3.69 and (using the experiment results) we approximately estimate that for 64 Hosts Safran can archive a speedup value of about only 6.4.

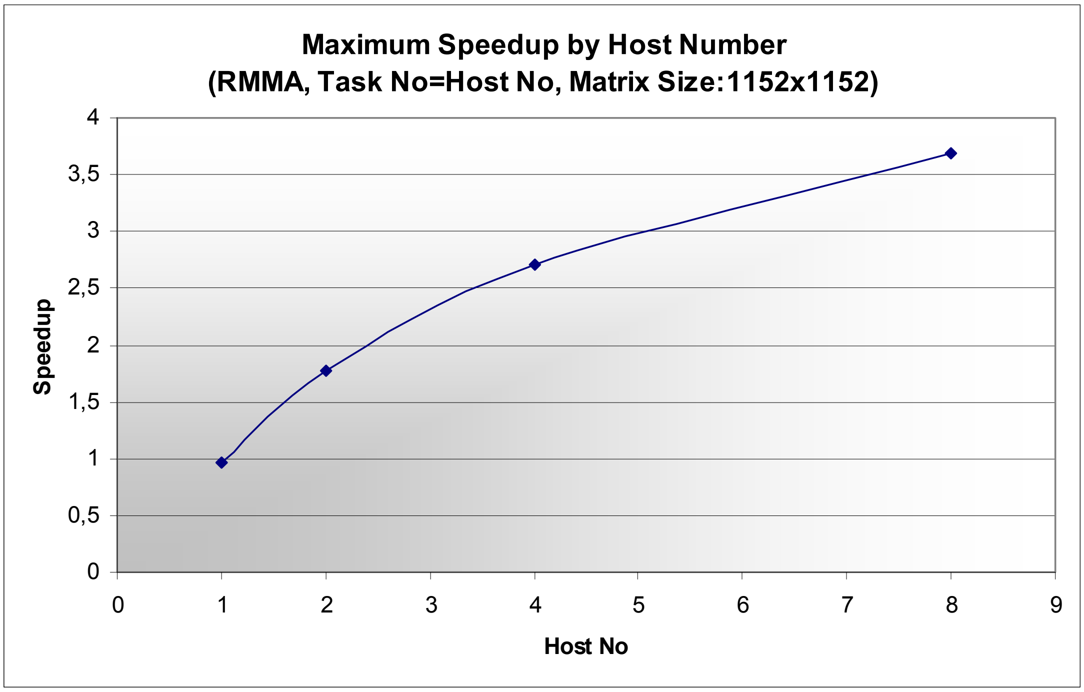

Figure 5.12 Host number vs. maximum speedups achieved for all RMMA tests

Figure 5.13 further clarifies the situation. As seen, efficiency of the system drops drastically as the number of Hosts increases.

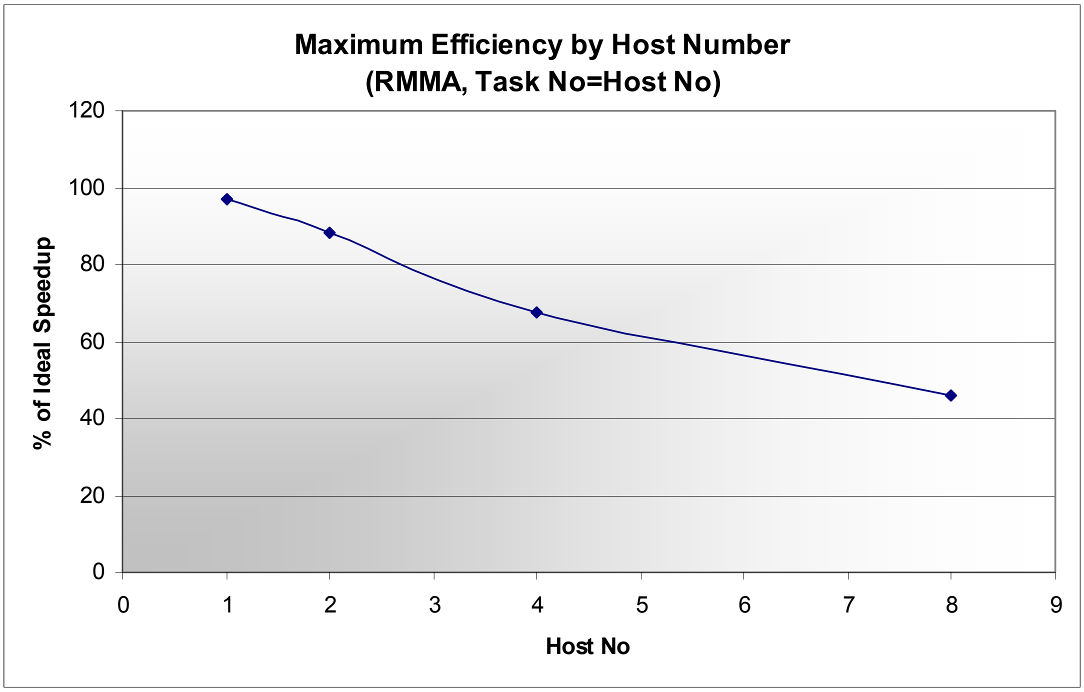

Figure 5.13 Host number vs. efficiency of Safran for RMMA

To sum up, from our analyses we can conclude that Safran is not much suitable for applications that have high data transfer and network I/O requirements.

5.2.2 Distributed Data Matrix Multiplication Application Tests

We run the distributed data matrix multiplication application serially and for 1, 2, 4, and 8 Host and for 1, 2, 4, 8, 16, and 32 Task Safran configurations. Here we present the discussion only for the 1 Host and 8 Host configurations as an example of our analysis approach. The analysis for the 2 Host and 4 Host configurations will be presented together with the other cases in the overall summary section.

Safran Configuration of 1 Host

The execution time vs. the problem size graph for the 1 Host and different Task numbers is given **Figure 5.14**.

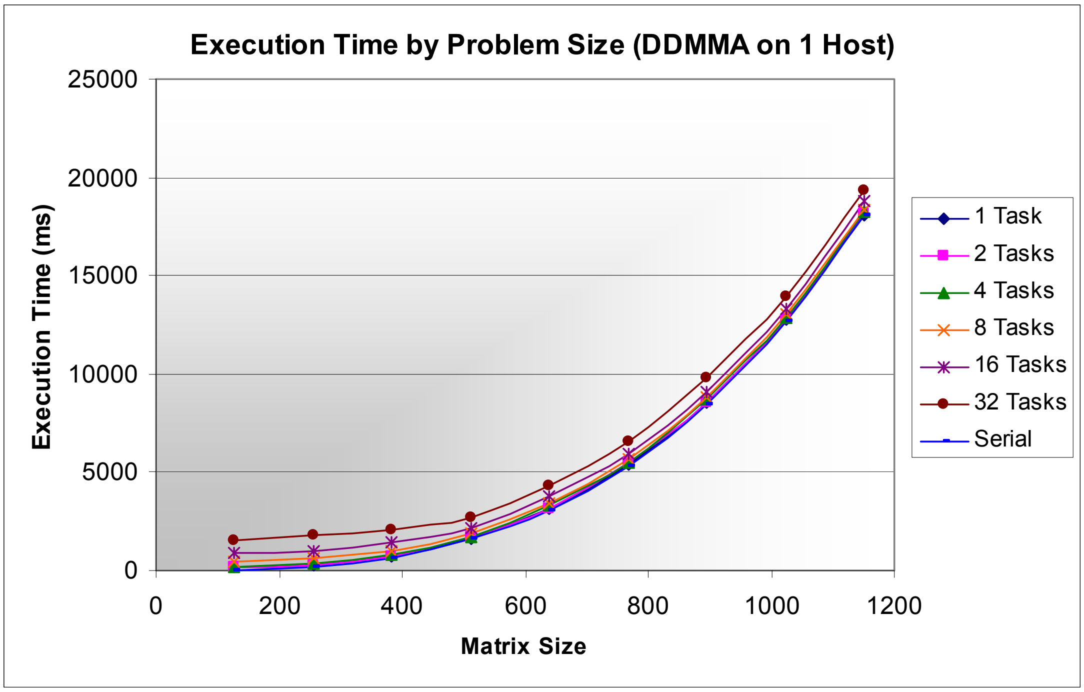

Figure 5.14 Problem size vs. execution time of graph for 1 Host DDMMA tests

As seen from **Figure 5.14**, for sufficiently large problem sizes the overhead of the system is quite low even for the configuration of 32 Tasks. Unlike the RMMA, the overhead is mainly due to Safran's bookkeeping activities. The overhead for the configuration of 32 Tasks and the largest problem size is 1267 ms which is about the %6.5 of the whole execution time. As the **Figure 5.15** shows, the execution time (therefore the overhead) is increasing linearly by Task number. The average overhead incurred by each Task is about 35 ms which is about %5.7 of the average execution time of a Task (604 ms for the 32 Task and 1152x1152 matrix size case). As the overhead created by each Task is generally constant, for much larger problems (i.e. high granularity Tasks) we can expect this percentage of overhead to decrease substantially. For example, for matrix of size 2048x2048, we estimate that this percentage of overhead to fall down to about %1.2 of the total execution time..

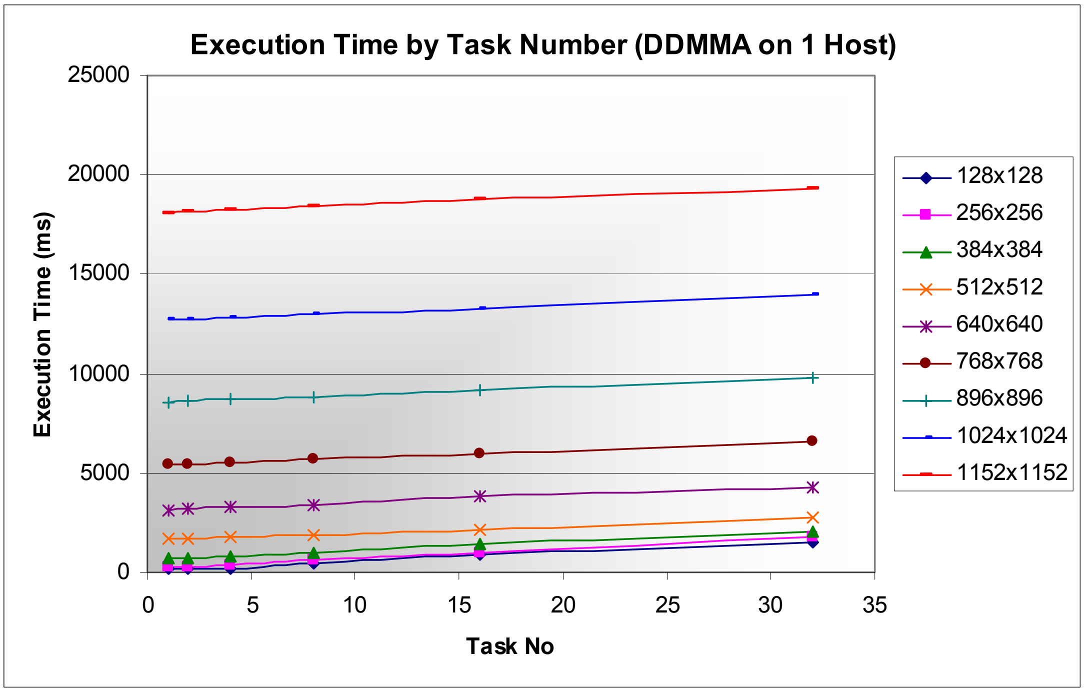

Figure 5.15 Task number vs. execution time graph for 1 Host DDMMA tests

The situation is more clearly demonstrated by **Figure 5.16**. As seen, the overhead time is generally increasing linearly by the Task number and rate of increase is mostly independent of the problem size.

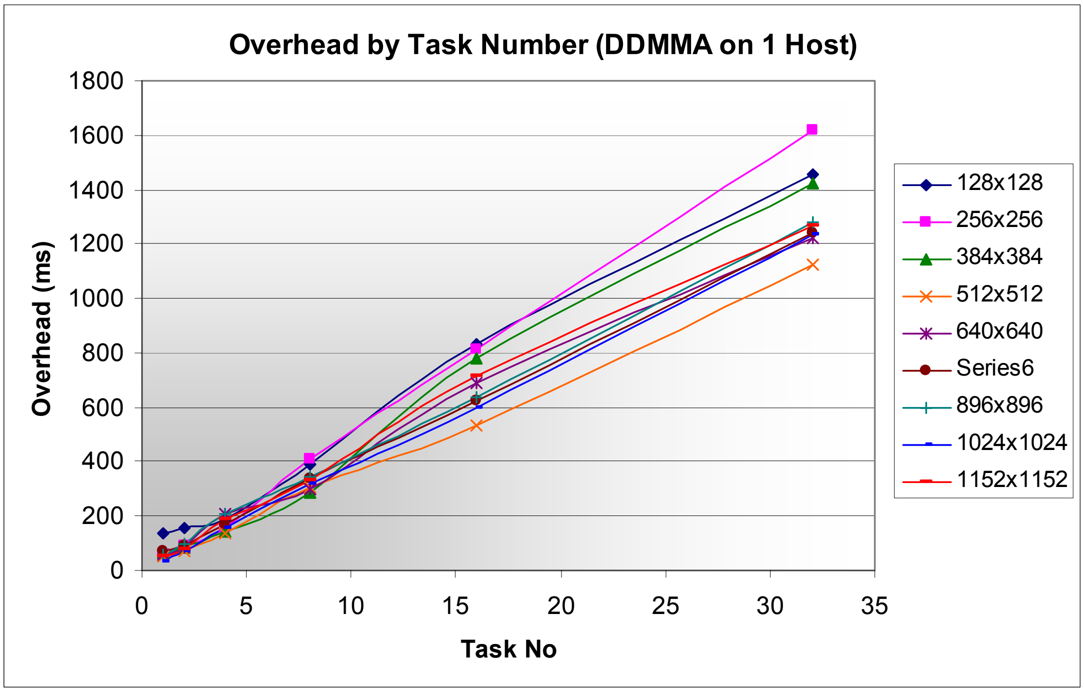

Figure 5.16 Task number vs. execution overhead graph for 1 Host DDMMA tests

These results and our analysis suggest that the overhead of Safran for pure CPU intensive applications is low and quite reasonable.

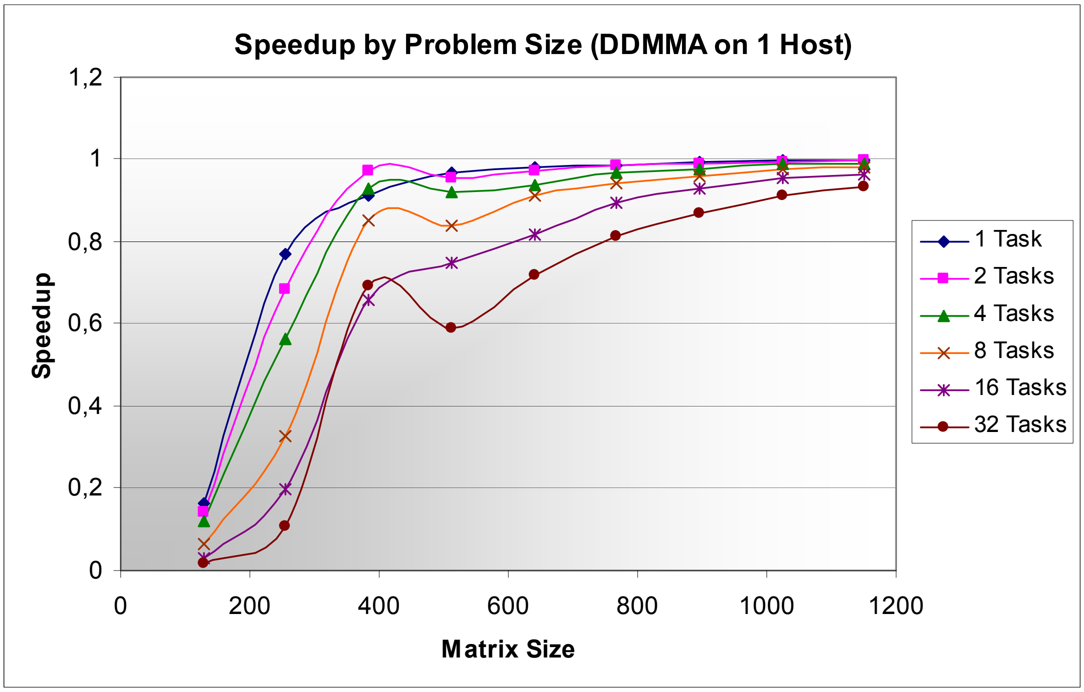

Figure 5.17 Task number vs. execution overhead graph for 1 Host DDMMA tests

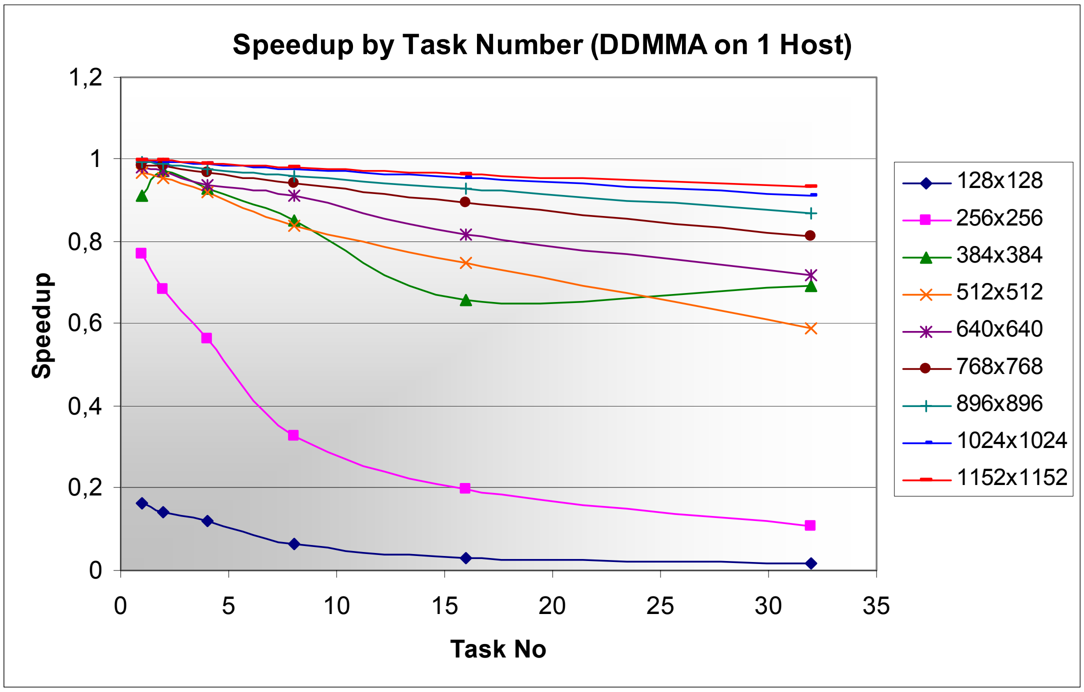

Figure 5.18 Task number vs. execution overhead graph for 1 Host DDMMA tests

Figure 5.17 and **Figure 5.18** show how speedup changes by the problem size and Task number respectively. **Figure 5.17** suggests that despite the overhead of the system, for larger problems quite good speedups can be achieved. As seen in **Figure 5.18**, for sufficiently large problem sizes (512x512 and larger) increasing Task number does not degrade the speedup very much as each Task adds only a small constant overhead.

Safran Configuration of 8 Hosts

The following two graphs (**Figure 5.19** and **Figure 5.20**) show the variation of execution time of distributed-data matrix multiplication application on an 8 Host Safran configuration for different problem sizes and Task numbers.

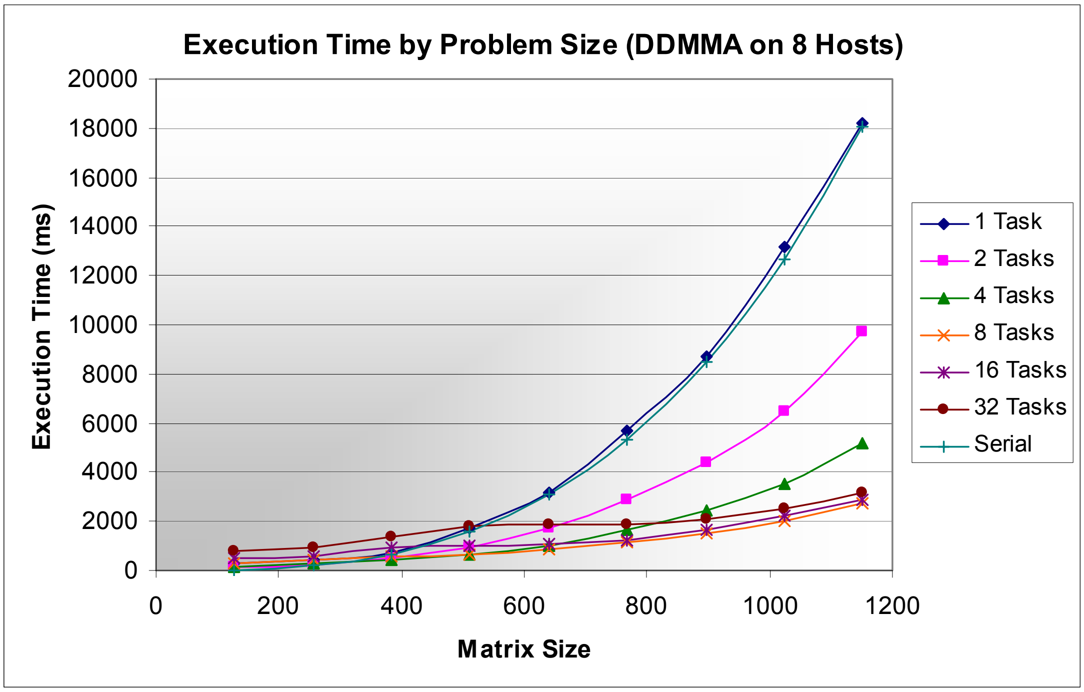

Figure 5.19 Problem size vs. execution time graph for 8 Host DDMMA tests

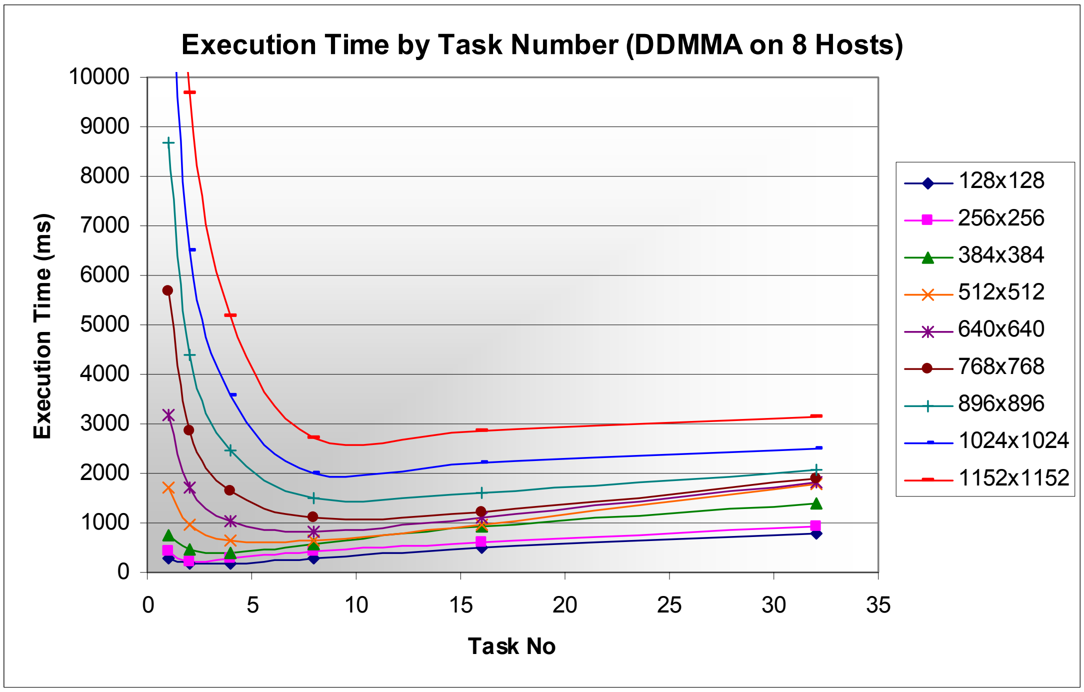

Figure 5.20 Task number vs. execution time graph for 8 Host DDMMA tests

As expected, it’s seen that (**Figure 5.19**) for a configuration of 8 Hosts, execution time decreases rapidly as the number of Tasks is increased from 1 to 8 and reaches the optimum (minimum) value for 8 Hosts for almost all problem sizes. Further increase in the Task number increases the execution time slightly which shows that overhead of increased Task number is not very prohibitive.

Figure 5.20 more clearly shows the change of execution time by Task number. As seen, for sufficiently large problem sizes minimum execution times are achieved when the Task number is 8. Larger Task numbers increases the execution time, but not much excessively.

The following graphs (**Figure 5.21** and **Figure 5.22**) show how the speedup achieved changes by the problem size and the Task number.

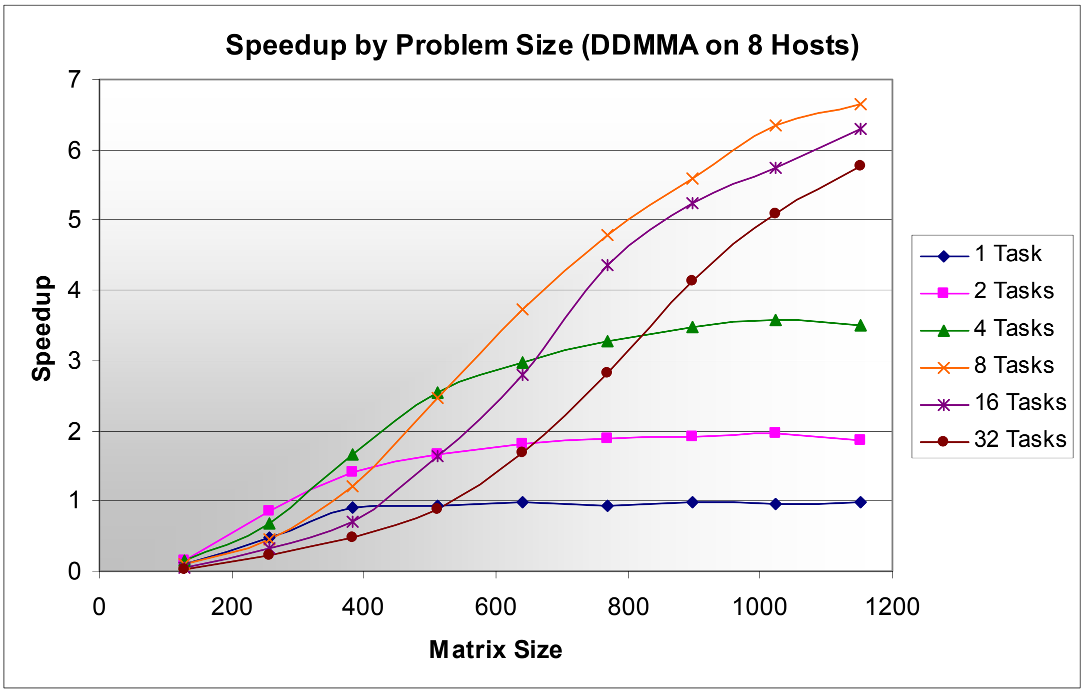

Figure 5.21 Problem size vs. speedup achieved graph for 8 Host DDMMA tests

Figure 5.21 suggest that speedup generally improves as the problem size (consequently the Task granularity) increases and gradually converges to a value (1, 2, and 4 Task cases). A speedup value of 6.65 is achieved with 8 Tasks for the maximum matrix size but convergence behavior is still not visible for this matrix size. So, for the configuration of 8 Hosts we expect some more improvement in the speedup for much larger and high granularity Tasks for which we were not able to conduct experiment. **Figure 5.22** suggests that the decrease in the speedup due to Task numbers larger than the optimum value (8) gets smaller as the application size increases.

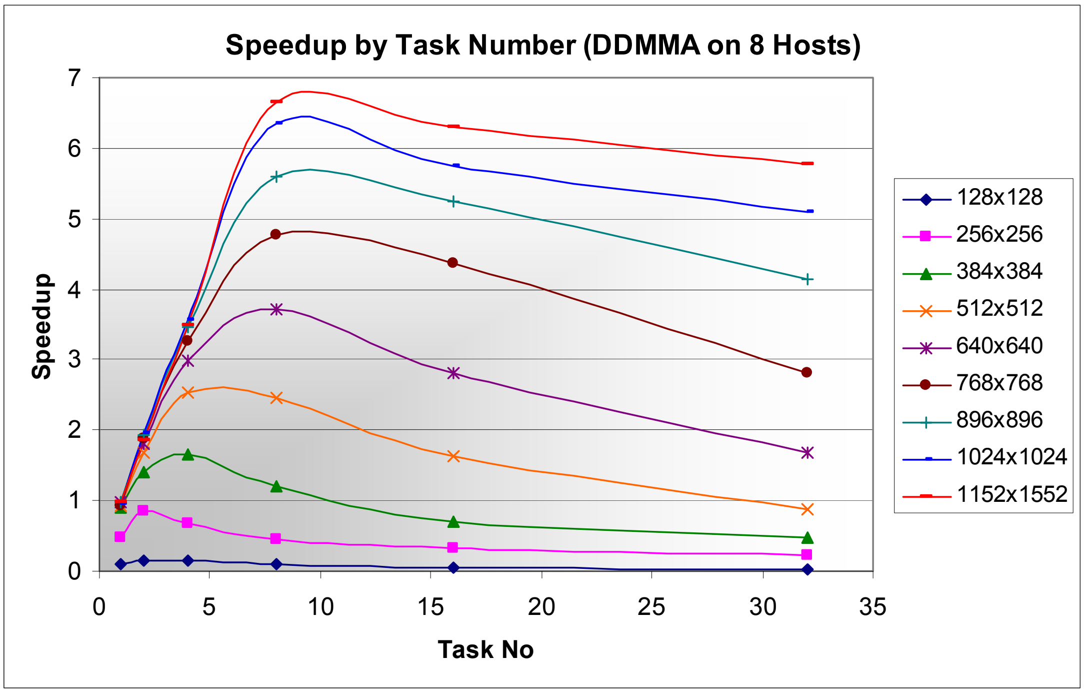

Figure 5.22 Task number vs. speedup achieved graph for 8 Host DDMMA tests

Summary of the DDMMA Tests

Here we present the overall analysis for all (1, 2, 4, 8 Host) configurations of Safran for distributed-data matrix multiplication application.

Figure 5.23 suggests that for each configuration, maximum speedup is achieved for the Task number that is equal to the Host number. Larger Task numbers cause degradation in the speedup but this degradation is not too much.

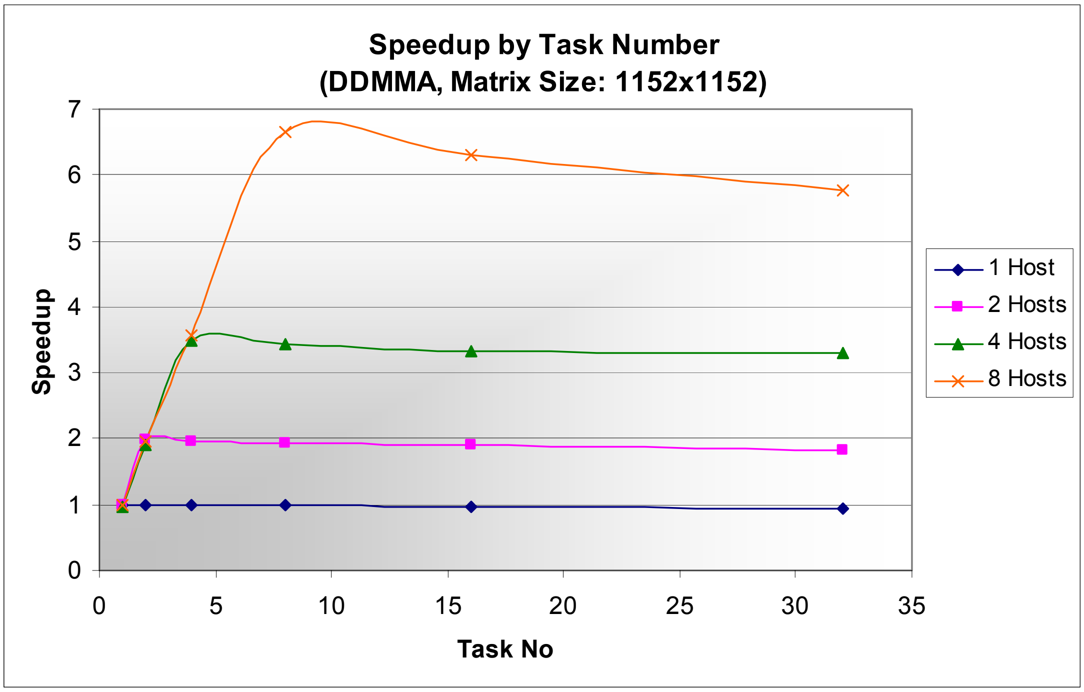

Figure 5.23 Task number vs. speedup achieved for all DDMMA tests

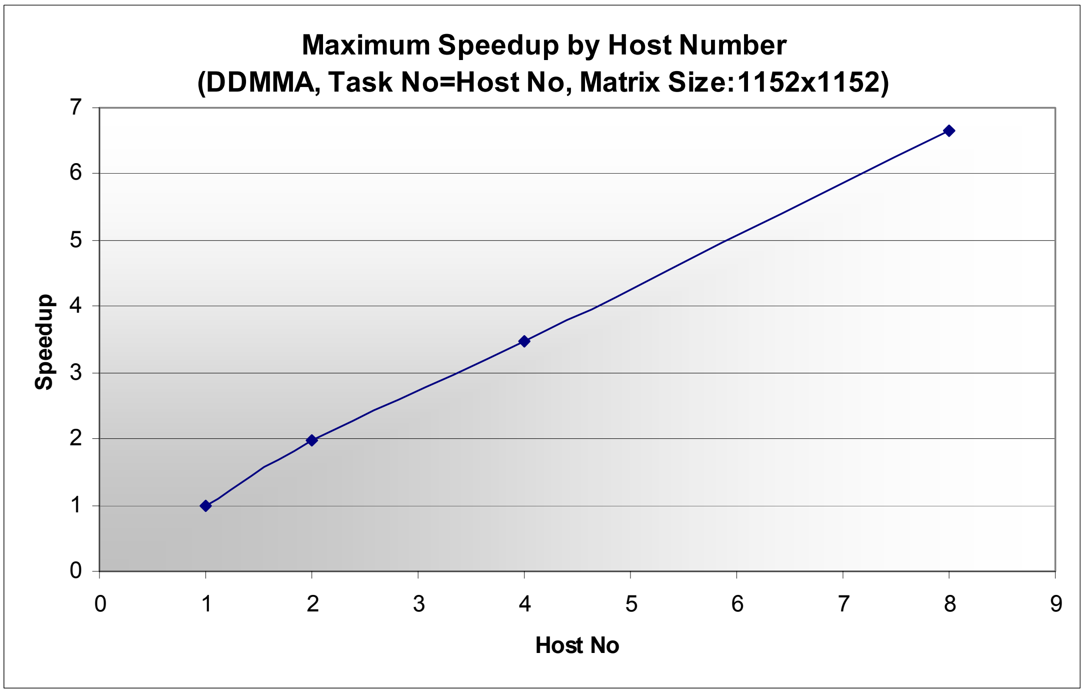

Figure 5.24 Host Number vs. speedup achieved for all DDMMA tests

Figure 5.24 shows the variation of maximum speedups by the Host number achieved during all experiments. It is seen that the maximum speedup achievable on a Safran system is linearly proportional to the Host number which means that Safran is quite scaleable for this kind of applications. Assuming that the system maintains its scalability characteristics, we roughly estimate (using the experiment results) that for 64 Hosts Safran can achieve a speedup value of about 51.

To sum up, from our analyses we can conclude that, Safran is quite suitable for applications that do not have much network I/O requirement but that are extensively computation bound.

5.2.3 Comparison

In this section we present the general comparison of the performance characteristics of Safran for the two different applications for which we conducted experiments.

The following figures (**Figure 5.25** and **Figure 5.26**) generally show how speedup changes by Host number and Task number for both applications.

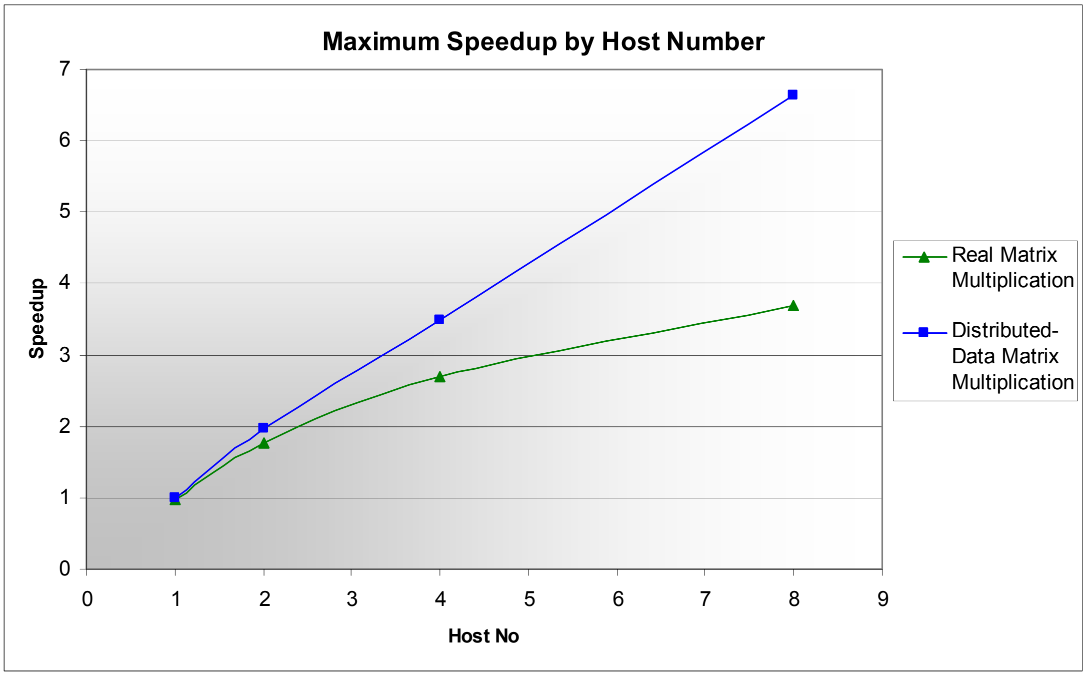

Figure 5.25 Host number vs. maximum speedup achieved for both applications.

Figure 5.25 clearly shows that Safran scales perfectly for pure CPU bound applications represented by the distributed-data matrix multiplication application. But for application that have excessive network I/O requirement (represented by real matrix multiplication) it is not much suitable because it does not achieve good speedup values and it does not scale at all as the number of Hosts is increased.

In **Figure 5.26** the effect of over-dividing the application (large Task number) for the two applications is seen. Larger Task number causes rapid drop in the speedup for the real matrix multiplication application because each Task adds large network I/O overhead to the execution time. For the distributed-data matrix multiplication application each Task adds only a small amount of constant overhead independent of the application.

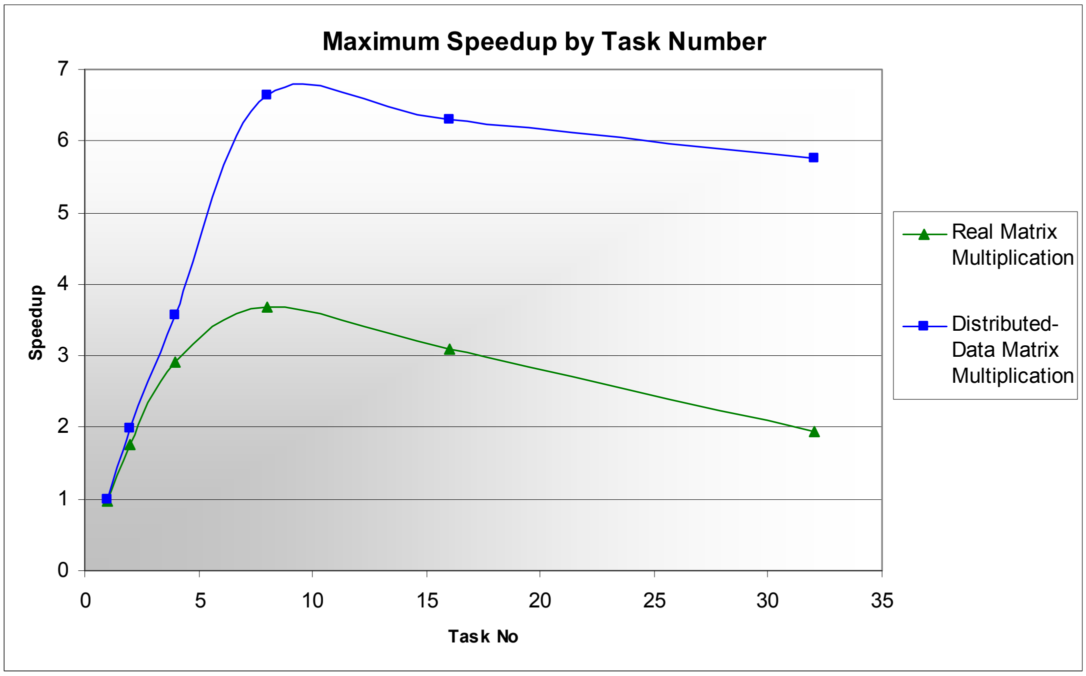

Figure 5.26 Task number vs. maximum speedup achieved for both applications.

Figure 5.27 shows how efficiency of the system changes by Host number. As seen, the efficiency of the system decreases rapidly for the I/O bound application as the number of Host of the system increases. For the pure CPU bound application efficiency also drops as the system extends but with a much smaller rate.

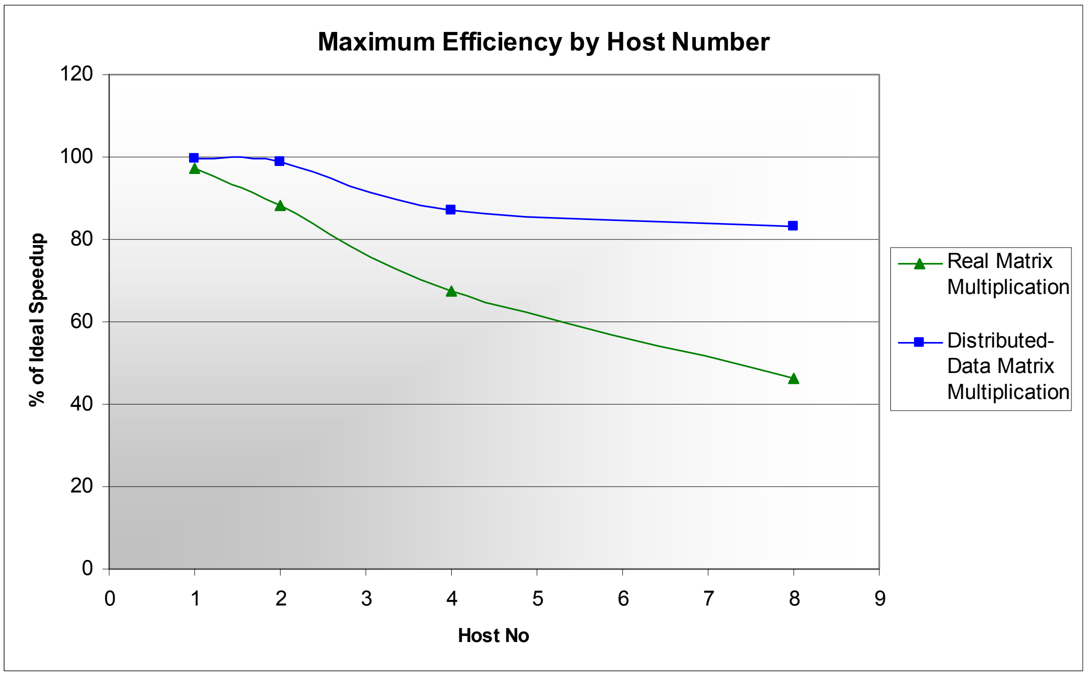

Figure 5.27 Host number vs. maximum efficiency achieved for both applications.

5.3 Summary of Test Results

From our analysis of all the test results we can reach the following conclusions about the performance characteristics of Safran:

- Safran is more appropriate for applications that require extensive computational capacity. The larger the applications' computation requirement (and the granularity of the sub-computations i.e. the Tasks) the better the speedups and systems efficiency achieved.
- Applications that require too much network I/O are not much suitable for running on Safran. For such application Safran does not scale very well i.e. adding more Hosts does not produce much better speedups.
- Applications that are pure CPU bound produce quite good speedup and efficiency values, so they are quite suitable for running on Safran. For such applications Safran scales perfectly i.e. adding more Hosts produce proportionally better speedups (i.e. smaller execution time).

- To achieve the optimum speedup, the application should be divided into the number of Task that is at least equal to the number of Hosts in the system. Larger Task numbers generally decreases the speedup and efficiency values but the rate of decrease is quite low for pure CPU bound applications. So, larger Task numbers is not prohibitive for this kind of applications. To help the application developer in deciding the optimum number of Tasks to use, Safran provides APIs (`ApplicationServices.getAvailableHostNo` method) that enable the application developer to dynamically determine the best-effort estimate of the number of available Hosts in the system. Application developers can use this API to determine the Task number to use.

CHAPTER 6

CONCLUSION

Safran is designed as a framework to support easy development of a wide variety of distributed and parallel applications on networks of heterogeneous workstations using the Java programming language. The main objective of Safran is to support platform independence and heterogeneity and also minimize the overheads related to setup and maintenance of the platform as well as making it easy to develop applications on the platform.

Safran supports platform independence and heterogeneity by using the Java Virtual Machine abstraction, which already solves these problems as a platform independent development and runtime environment. Although we conducted our experiments and collected data on a network of Sun workstations, we also the run our test applications on a small number of Windows machines just to verify that our system works on heterogeneous systems.

Safran makes it easy to setup and maintain the system by supporting some level of automatic setup of the system. Entities named Brokers, Hosts and Applications can discover and use each other automatically if required. Compared to other similar works, setup and maintenance of Safran is easier. A Host that is started up somewhere on the network can automatically find and join a Safran cluster. So, Safran keeps the overhead minimum associated with installation, configuration, and management of the system.

Most of the other frameworks we examined support either a low-level programming model based on distributed objects or a high-level of programming model for parallel applications. Safran provides support for the both programming and computational models. So, Safran enables development of both general distributed applications (through its low-level programming model) and parallel applications (through its high-level programming model).

Safran's high-level programming model provides a virtualization of a network of heterogeneous workstations so that applications are written for a virtual, single computer that has parallel processing capability. As the applications have a virtual single machine view of the network, new machines can transparently join the system or existing machines can leave the system dynamically at runtime. The application does not have to deal with the dynamically changing properties (machine failures for example) of the network.

Safran's high level programming model for parallel application development is specifically designed for compute intensive, CPU-bound applications with high granularity of sub-computations. Our experiments verified that for this kind of applications Safran achieves reasonable speedups and scales reasonably as the system gets larger.

CHAPTER 7

FUTURE WORK

This section contains discussions about possible future extensions to Safran that are not included in the current design and implementation but might extend the capabilities of systems in important ways. Current design is flexible enough so that future extension can be built on it without extensive modifications to it.

7.1 High-level Computational and Programming Model

As mentioned in **Section 3.2.2**, currently Safran only supports piecework computations type of high-level computational model because it does not allow communication of distributed processes. This limits the type of problems that can be solved on Safran. To get rid of this limitation and increase the number of problems solvable on Safran infrastructure, support for different computational models can be added to the system. For example to allow communication of distributed processes a shared memory abstraction (possibly based on JavaSpaces[30]) can be provided. This solution might eliminate the distributed snapshot problem associated with direct point-to-point communication of distributed processes, which is explained in detail in **Section 3.2.2**.

7.2 Low-level Computational and Programming Model

The low-level programming model based on the concepts of distributed-objects is made easy to use by the help of Dynamic Proxies layer. But some functionality of the layer such as making asynchronous and one-way method calls are provided through an API that is difficult to use as it's mainly designed for Safran's own internal use.

The API that provides this kind of functionality can be re-designed to make it more easy to use by end-user applications.

7.3 Security

Safran's current design is based on the requirements of a single administrative institution. So, security infrastructure is mainly designed based on the assumption that all entities are controlled by the same administrative body and they all can trust each other.

Safran's infrastructure is mainly ready to be extended to support more than one, distributed organizations. If Safran would be extended to out of administrative boundaries of a single institution, security infrastructure must be re-designed such that Broker, Hosts and Applications must be authenticated and authorized by each other for different operations. This way clusters formed by totally different organizations can be combined to work together without security concerns.

7.4 Network Communication Service Implementation

Current implementation of Safran is based on Java RMI technology. RMI requires some tight relation between processes and it is difficult to get it work through firewalls and proxy servers. Network Communication Service Layer implementation can be replaced with a new implementation based on low level network sockets to make Safran capable of working through firewalls without difficulty.

7.5 Scheduling and Load Balancing

The implementation of Safran we provide makes some too strong assumptions about the type of applications that are expected to be run on Safran. The design and implementation of the Task scheduling and load balancing infrastructure are mainly based on these assumptions. For example, it's assumed that applications that will run on Safran will be mostly computation bound and also Tasks running on Hosts will use almost all computation capacity of the Hosts. So, only one Task is allowed to run on a Host exclusively and non-preemptively. While this design produces good results for applications and cases for which the assumptions they are based on holds, for some other type of applications and scenarios it might not produce good speedup and throughput values. In short, the scheduling and load balancing sub-system is quite simple and is not designed for wide variety of applications and runtime scenarios.

The scheduling and load balancing sub-system can be redesigned and implemented for different type of applications and for more complex and advanced runtime scenarios and requirements.

REFERENCES

[1] R. Bisani and A. Forin, Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. Computers*, Vol. 37, No. 8: pp. 930-945, Aug 1988.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman, Super Web: Towards a Global Web-based Parallel Computing Infrastructure. *11th International Parallel Processing Symposium*, pp. 100-106, April 1997.

[3] B. Christiansen, P. Cappello, M.F. Ionescu, M.O. Neary, K. Schanuser. And D. Wu. Javelin: Internet Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, Vol. 9, No. 11, pp. 1139-1160, November 1997.

[4] P. Cappello, Janet's Abstract Distributed Service Component. In *Proceedings of 15th IASTED Int. Conf. on Parallel and Distributed Computing and Systems*, pp. 751 - 756, Marina del Rey, California, November 2003.

[5] Baratloo, M. Karaul, Z. Kedem, P. Wijckoff, Charlotte: Metacomputing on the Web. *Future Generation Computer Systems, Vol. 15*, pp. 559-570, 1999.

[6] A. Baratloo, M. Karaul, H. Karl, and Z. Kedem. KnittingFactory: An infrastructure for distributed Web applications. TR1997-748, New York University, 1997.

[7] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an environment for parallel, distributed and mobile Java applications. *ACM 1999 Java Grande Conference*, pp. 15–24, 1999.

[8] M. Philippsen, M. Zenger. JavaParty: Transparent Remote Objects in Java. *Concurrency: Practice and Experience,* Vol. 9 No. 11, pp. 1225-1242, 1997.

[9] T. Brecht, H. Sandhu, J. Talbott, and M. Shan. ParaWeb: Towards World-Wide Supercomputing. In *Proceedings of the 7th ACM SIGOPS European Workshop*, pp. 181-188, September 1996.

[10] T. Fahringer. JavaSymphony: A system for development of locality-oriented distributed and parallel Java applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, pp. 145-152, 2000.

[11] A. J. Ferrari, JPVM: Network Parallel Computing in Java. *Technical Report CS-97-29, Dept. of Comp. Sci., University of Virginia*, Charlottesville, VA 22903, USA.

[12] G. K. Thiruvathukal, P. M. Dickens and S. Bhatti, Java on Networks of Workstations (JavaNOW): A Parallel Computing Framework Inspired by Linda and the Message Passing Interface (MPI). *DePaul University, JHPC Research Laboratory, School of CTI.*

[13] L. F. Lau, A. L. Ananda, G. Tan, W. F. Wong, JAVM: Internet-based Parallel Computing Using Java. *School of Computing, National University of Singapore.*

[14] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer, ATLAS: An Infrastructure for Global Computing. In *Proceedings of the 7th ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, pp. 165-172, 1996.

[15] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: A Migratable Parallel Objects Framework using Java. *In Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, pp. 151-159, February 1998.

[16] C. H. Cap and V. Stumpen, Efficient parallel computing in distributed workstation environments. *Parallel Computing* Vol. 19, pp. 1221-1234, 1993.

[17] G. Cabillic and I. Puaut, Stardust: An Environment for Parallel Programming on Networks of Heterogeneous Workstations. *Journal of Parallel and Distributed Computing*, Vol. 40, pp. 65-80, 1997.

[18] Michal Cierniak, M. Javed Zaki and Wei Li, Compile Time Scheduling Algorithms for a Heterogeneous Network of Workstations. *The Computer Journal*, Vol. 40, No. 6, pp. 356-372, July 1997.

[19] A. Weinrib and S. Shenker, Adaptive Load Sharing in Large Heterogeneous Systems. In *Proceedings of the IEEE INFO COM*, pp. 986-994. 1988.

[20] Yan Gu, B.S. Lee, Wentong Cai, Evaluation of Java Thread Performance on Two Different Multithreaded Kernels. *Operating Systems Review*, Vol. 33, No. 1, pp. 34-46, 1999.

[21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.1*, Research Report, June 1995.

[22] Message Passing Interface Forum. *MPI-2: Extension to the Message-Passing Interface Standard*, Research Report, July 1997.

[23] V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing. *Journal of Concurrency: Practice and Experience*, Vol. 2, No. 4, pp. 315-339, December 1990.

[24] M. Hamdi, Y. Pan, B. Hamidzadeh, and F.M. Lim, Potentials and Limitations of Parallel Computing on a Cluster of Workstations. *Proceedings 1997 International Conference on Parallel & Dist. Systems* pp. 572-577, 1997.

[25] O. Larson, M. Feig and L. Johnsson, Sommet Computing Experiences For Scientific Applications. *Parallel Processing Letters*, Vol.9 No.2, pp. 243-252, 1999.

[26] Ahuja, Sudhir, Carriero, and Gelernter, Linda and Friends. *IEEE Computer*, Vol.19, No.8, pp. 26-34 August 1986.

[27] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel Processing On Networks of Workstations: A Fault-Tolerant High Performance Approach. In *Proceedings of 15th IEEE International Conf. on Distributed Computing Systems*, pp. 467-474, 1995.

[28] V. Sunderam, J. Dongarra, A. Geist, and R. Manchek, The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, Vol.20, pp.293-311, June 1992.

[29] W. Grosso, *Java RMI*, O'Reilly, October 2001

[30] Sun Microsystems, Javaspaces Specification. http://www.javasoft.com/products/javaspaces/specs/index.html (1998).

[31] J. Gosling, B. Joy, and G. Steel, *The Java Language Specification*. Addison Wesley Developers Press, Sunsoft Java Series, 1996.

[32] J. Gosling and H. McGilton, The Java Language Environment. Technical Report, Sun Microsystems, http://java.sun.com, October1995.

[33] T. Linholm and F. Yellin. *The Java Virtual Machine Specification*, Addison Wesley, 1999.

[34] M. Stang and S. Whinston. Enterprise Computing with Jini Technology. *In Issue of IT Professional, IEEE Computer Society*, Vol.3, No.1, pp. 33-38, February 2001.

[35] Sun Microsystems Inc., Palo Alto, CA, *Java Platform API Specification Version 1.5*, 2004.

[36] Sun Microsystems Inc., Palo Alto, CA, *Java Remote Method Invocation Specification JDK 1.2*, 1998.

APPENDIX A: INTERFACES PROVIDED BY SAFRAN'S LAYERS

Here we list the programming interface (method signatures and brief documentation) provided by layers of Safran to each other and to application developers.

A.1 Communication SPI Layer

Communication SPI basically consists of four Java interface definitions. Underlying communication service provider implementation must implement these interfaces.

```
/* Provides the interface for concrete ObjectHostingService Creator
 * classes. Together with RemoteObjectHostCreator this interface
 * provides an extensibility point for Safran's remoting layer's implementation.
 */
public interface ObjectHostingServiceCreator {
        /**
         * Creates and returns a new <code>ObjectHostingService</code> that will
         * service on the local machine and on the given port once started.
         */
        public ObjectHostingService createService(int port)
                throws ServiceCreationException;

        /**
         * Gets a new <code>ObjectHostingService</code> referring to the service
         * running on the given machine and port.
         */
        public ObjectHostingService getService(String hostName, int port)
                throws ServiceNotFoundException;
}
```

```
/**
 * Represents a remotely accessible service (which might be running on the local
 * machine or some remote machine) that hosts remotely accessible objects.
 */
public interface ObjectHostingService {
        /**
         * Starts the object hosting service.
         */
        public void start() throws RemotingException;

        /**
         * Stops the remote object hosting service. All hosted objects are deleted
         * and the service is not accessible to RemoteObjectHost's anymore.
         */
        public void stop() throws RemotingException;

        /**
         * Returns the status of this ObjectHostingService.
         */
        public boolean isStarted() throws RemotingException;

        /**
```

```
         * Gets the IP address or DNS resolvable host name of the machine on
         * which this ObjectHostingService is running.
         */
        public String getHostName() throws RemotingException;

        /**
         * Gets the port number on which this ObjectHostingService is running.
         */
        public int getPort() throws RemotingException;

        /**
         * Gets whether this ObjectHostingService is running in the same
         * JVM process with the caller of this method.
         */
        public boolean isLocalToProcess() throws RemotingException;

        /**
         * Gets whether this ObjectHostingService is running on the same
         * host (machine) as the caller's JVM process is running.
         */
        public boolean isLocalToHost() throws RemotingException;
}
```

```
/**
 * Provides the interface for concrete RemoteObjectHost Creator classes.
 * Together with ObjectHostingServiceCreator interface provides
 * an extensibility point for Safran's remoting layer's implementation.
 */
public interface RemoteObjectHostCreator {
        /**
         * Creates and returns an object implementing RemoteObjectHost
         * interface referring to the ObjectHostingService which is
         * running on the given host and port number.
         */
        public RemoteObjectHost getObjectHost(String hostName, int port)
                throws ServiceNotFoundException;
}
```

```
/**
 * Allows access to services of an object hosting service i.e. ObjectHostingService.
 */
public interface RemoteObjectHost {
        /**
         * Creates an new instance of a class on a remote host and gets back the
         * UUID of the created object. A new instance of the class clazz on the
         * remote ObjectHostingService referred by this is created using the
         * constructor of the class that can accept the given array of parameter
         * types.
         */
        public UUID createObject(Class clazz, Object[] ctorArgs, Class[] paramTypes)
                throws RemotingException;

        /**
         * Puts the given object itself (not a copy of it) to the this
         * RemoteObjectHost and exposes it to remote access with the
         * given well-known name.
         */
        public UUID exportObject(Object obj, String objectName, UUID exporterID)
                throws RemotingException;

        /**
         * Un-exports and removes the remotely accessible object which is previously
         * exported with the a well-known name to this RemoteObjectHost.
         */
        public void unexportObject(UUID objectID, UUID callerID)
                throws RemotingException;
```

```
    /**
     * Gets the UUID of the hosted object which is exported with the given
     * well-known name.
     */
    public UUID getExportedObject(String objectName) throws RemotingException;

    /**
     * Get the well-known name with which the hosted object whose UUID is given.
     */
    public String getExportedObjectName(UUID objectID) throws RemotingException;

    /**
     * Puts a copy of the local object to the remote ObjectHostingService
     * referred by this and returns a UUID of the remote object.
     */
    public UUID putObjectCopy(Object obj) throws RemotingException;

    /**
     * Puts the given object itself (not a copy or clone of it) to the
     * ObjectHostingService referred by this and returns a UUID
     * for the object. For this to be possible ObjectHostingService
     * referred by this must be running in the same JVM as the caller. Otherwise
     * this method should throw an OperationUnsupportedException.
     */
    public UUID putObject(Object obj) throws RemotingException;

    /**
     * Gets a cloned copy of the remote object with the given
     * UUID hosted by the ObjectHostingService referred by this.
     */
    public Object getObjectCopy(UUID objectID)
           throws ObjectNotFoundException, RemotingException;

    /**
     * Deletes the remote object with the given UUID hosted by
     * the ObjectHostingService referred by this.
     */
    public void deleteObject(UUID objectID)
           throws ObjectNotFoundException, RemotingException;

    /**
     * Gets the class of the hosted object whose <code>UUID</code> is given.
     */
    public Class getObjectType(UUID objectID)
           throws ObjectNotFoundException, RemotingException;

    /**
     * Returns true if the connected object hosting service
     * (ObjectHostingService) is hosting a valid object with the given ObjectID.
     */
    public boolean objectExists(UUID objectID)throws RemotingException;

    /**
     * Calls the method with the given name on the remote object whose unique
     * identifier is given.
     */
    public Object invokeMethod(UUID objectID, String method, Object[] args,
           Class[] paramTypes) throws ObjectNotFoundException, RemotingException;

    /**
     * Returns the DNS resolvable host name on which the connected remote object
     * hosting service (ObjectHostingService) is running.
     */
    public String getHostName() throws RemotingException;

    /**
     * Returns the port number on which the connected remote object hosting
     * service (ObjectHostingService) is running.
     */
    public int getPort() throws RemotingException;
```

```
        /**
         * Returns <code>true</code> if there is valid connection to the refered
         * remote object hosting service (ObjectHostingService).
         */
        public boolean isValid();

        /**
         * Gets whether the connected remote object hosting service
         * (ObjectHostingService) is running in the same JVM process with
         * the caller of this method.
         */
        public boolean isLocalToProcess() throws RemotingException;

        /**
         * Gets whether the connected remote object hosting service
         * (ObjectHostingService) is running on the same host (machine)
         * as the caller's JVM process is running.
         */
        public boolean isLocalToHost() throws RemotingException;
}
```

A.2 Distributed-Objects Layer

```
/**
 * Enables creation of ObjectHostingService objects i.e. provides
 * static methods that return ObjectHostingService objects.
 */
public final class ObjectHostingServiceLocator {
        /**
         * Creates and returns a new ObjectHostingService that will service
         * on the local machine and on default port once started.
         */
        public static ObjectHostingService createService() throws
                ServiceCreationException;

        /**
         * Creates and returns a new ObjectHostingService that will service
         * on the local machine and on the given port once started.
         */
        public static ObjectHostingService createService(int port)
                throws ServiceCreationException;

        /**
         * Returns the ObjectHostingService running on the local machine
         * and on default port.
         */
        public static ObjectHostingService getService()
                throws ServiceNotFoundException, RemotingException;

        /**
         * Returns the ObjectHostingService running on the local machine
         * and on the given port.
         */
        public static ObjectHostingService getService(int port)
                throws ServiceNotFoundException, RemotingException;
}
```

```
/**
 * Provides means for initializing and configuring remoting subsystem.
 * Users of the remoting layer must use static methods of this class
 * for initializing (and configuring if necessary) remoting subsystem
 * before being able to doing anything with remoting layer services.
 */
public final class RemotingConfiguration {
        /**
         * Returns whether the remoting subsystem is initialized with a call to
         * RemotingServices.initialize method.
         */
        public static boolean isInitalized();

        /**
         * Configures the remoting subsystem according to the given configuration
         * file. If ever this method is called it must be called before a call to
         * RemotingServices.initialize i.e. once the remoting subsystem is
         * initialized it cannot be reconfigured.
         */
        public static void configure(String configFile) throws IOException;

        /**
         * Configures the remoting subsystem according to the configuration data from
         * the given InputStream. If ever this method is called it must be called
         * before a call to RemotingServices.initialize i.e. once the remoting
         * subsystem is initialized it cannot be reconfigured.
         */
        public static void configure(InputStream inputStream) throws IOException;

        /**
         * Returns the currently configured RemoteObjectHostCreator implementation
         * object. This method cannot be called before the remoting subsystem is
         * initialized with a call to RemotingServices.initialize method.
         */
        public static int getDefaultRemotingPort();
}
```

```
/**
 * This class provides client side services for remoting sub-system. Clients can
 * create, put, delete, and move objects on and between remote hosts.
 */
public final class RemotingServices {
        /**
         * Initializes the remoting subsystem. Users of remoting layer must call this
         * method before doing anything with the remoting layer. If required remoting
         * subsystem must be configured using the RemotingConfiguration.configure
         * method before being initialized. Once the remoting subsystem initialized
         * it is not possible to configure or reconfigure it. Attempts to do so will
         * result in an exception being thrown.
         */
        public static void initialize() throws RemotingException;

        /**
         * Creates an new instance of a class on a remote host and gets back a proxy
         * to the created object. A new instance of the class clazz on the remote
         * ObjectHostingService running on the given host and default
         * remoting port is created using the constructor of the class that can
         * accept the given array of objects as the arguments.
         */
        public static Object createObject(String hostName,
               Class clazz, Object[] ctorArgs) throws RemotingException;

        /**
         * Creates an new instance of a class on a remote host and gets back a proxy
         * to the created object. A new instance of the class clazz on the remote
         * ObjectHostingService running on the given host port is
         * created using the constructor of the class that can accept the given array
         * of objects as the arguments.
         */
        public static Object createObject(String hostName, int port,
               Class clazz, Object[] ctorArgs) throws RemotingException;

        /**
         * Creates an new instance of a class on a remote host and gets back a proxy
         * to the created object. A new instance of the class clazz on the remote
         * ObjectHostingService running on the given host and default
         * remoting port using the constructor of the class that can accept the given
         * array of parameter types.
         */
        public static Object createObject(String hostName, Class clazz,
               Class[] paramTypes, Object[] ctorArgs) throws RemotingException;

        /**
         * Creates an new instance of a class on a remote host and gets back a proxy
         * to the created object. A new instance of the class clazz on the remote
         * ObjectHostingService running on the given host and port using
         * the constructor of the class that can accept the given array of parameter
         * types.
         */
        public static Object createObject(String hostName, int port, Class clazz,
               Class[] paramTypes, Object[] ctorArgs) throws RemotingException;

        /**
         * Puts the given object itself (not a copy of it) to the
         * ObjectHostingService running in this JVM and at the default
         * port and exposes it to remote access with the given well-known name.
         * Remote clients can access and make method calls on the given object itself
         * by using RemotingServices.getExportedObject method.
         */
        public static void exportObject(Object obj, String objectName)
               throws NameInUseException, ServiceNotFoundException,RemotingException;

        /**
         * Puts the given object itself (not a copy of it) to the
         * ObjectHostingService running in this JVM and at the default
         * port and exposes it to remote access with the given well-known name.
         * Remote clients can access and make method calls on the given object itself
         * by using RemotingServices.getExportedObject method.
         */
        public static void exportObject(Object obj, String objectName, int port)
               throws NameInUseException, ServiceNotFoundException,RemotingException;

        /**
         * Un-exports and removes the remotely accessible object which is previously
         * exported with the given well-known name to the ObjectHostingService
         * running in the local JVM and on the default port.
         * After call to this method the object that is previously exported with the
         * given well-known name is no longer accessible to remote clients even if
         * they acquired a remote reference to it before it is un-exported.
         */
        public static void unexportObject(String objectName)throws RemotingException;

        /**
         * Un-exports and removes the remotely accessible object which is previously
         * exported with the given well-known name to the ObjectHostingService
         * running in the local JVM and on the given port.
         * After call to this method the object that is previously exported with the
         * given well-known name is no longer accessible to remote clients even if
         * they acquired a remote reference to it before it is un-exported.
         */
        public static void unexportObject(String objectName, int port)
               throws RemotingException;

        /**
         * Returns a proxy object that can be used to access the remote object
         * which is hosted by the remote ObjectHostingService running on
         * the given host and default port and that is exported with the given
         * well-known name.
         */
        public static Object getExportedObject(String objectName, String hostName)
               throws RemotingException;

        /**
         * Returns a proxy object that can be used to access the remote object
         * which is hosted by the remote ObjectHostingService running on
         * the given host and given port and that is exported with the given
         * well-known name.
         */
```

```
    public static Object getExportedObject(String objectName, String hostName,
           int port) throws RemotingException;

    /**
     * Creates and returns a proxy object that can be used to remotely call
     * methods on the given object itself (not it's copy). The returned proxy
     * object supports the same interfaces that are implemented by the type
     * of the given object. Method calls on the returned object are dispatched
     * to the original given object.
     */
    public static Object createCallbackHandle(Object obj)
           throws RemotingException;

    /**
     * Puts a copy of the given object to the ObjectHostingService
     * running on the given host and default port.
     * An object that allows access to the remotely hosted copy of the
     * given object is returned. The returned proxy object supports the
     * same interfaces that are implemented by the type of the given object.
     */
    public static Object putObjectCopyTo(Object obj, String hostName)
           throws RemotingException;

    /**
     * Puts a copy of the given object to the ObjectHostingService
     * running on the given host and port. An object that allows access to the
     * remotely hosted copy of the given object is returned. The returned proxy
     * object supports the same interfaces that are implemented by the type of
     * the given object.
     */
    public static Object putObjectCopyTo(Object obj, String hostName, int port)
           throws RemotingException;

    /**
     * Gets a local copy of the remotely hosted object referred by the parameter
     * remoteObj.
     */
    public static Object getObjectCopy(Object remoteObj)throws RemotingException;

    /**
     * Moves the remotely hosted object referred by the parameter remoteObj from
     * its current host to the remote ObjectHostingService running on the given
     * host and default remoting port. After the remote object is moved all local
     * references to the remote object is transparently updated so that users of
     * local references do not need to do anything related to the movement of the
     * object.
     */
    public static void moveObjectTo(Object remoteObj, String hostName)
           throws RemotingException;

    /**
     * Moves the remotely hosted object referred by the parameter
     * remoteObj from its current host to the remote ObjectHostingService running
     * on the given host and port. After the remote object is moved all local
     * references to the remote object is transparently updated so that users of
     * local references do not need to do anything related to the movement of the
     * object.
     */
    public static void moveObjectTo(Object remoteObj, String hostName, int port)
           throws RemotingException;

    /**
     * Deletes and removes the remotely hosted object referred by the parameter
     * remoteObj.
     */
    public static void deleteObject(Object remoteObj)throws RemotingException;

    /**
     * Determines whether the given object refers to a valid/accessible remote
     * object.
```

```
         */
        public static boolean isValidRemoteObject(Object remoteObj);

        /**
         * Returns the RemoteObjectRef that enables access to remote
         * object given as the parameter. If the given parameter is itself an
         * RemoteObjectRef object, returns it. If else the given parameter
         * is a proxy object to a remote object, return the RemoteObjectRef
         * that backs that proxy object. Other wise an exception is thrown.
         */
        public static RemoteObjectRef getRemoteObjectRef(Object remoteObj);
}
```

```
/**
 * Represents a reference to an object hosted by a remote object hosting service
 * (ObjectHostingService). In a single VM there is only one
 * RemoteObjectRef object referring to the same remotely hosted object.
 */
public final class RemoteObjectRef implements Serializable {
        /**
         * Invokes the method of the remote object referred by this RemoteObjectRef
         * with the given name by passing the array of object as the arguments.
         */
        public MethodInvocationResult invokeMethod(String methodName, Object[] args);

        /**
         * Invokes the method of the remote object referred by this RemoteObjectRef
         * with the given name by passing the array of object as the arguments. The
         * method of the object that has the given name and that has a of list
         * parameter types exactly matching paramTypes array is called.
         */
        public MethodInvocationResult invokeMethod(String methodName,
                Class[] paramTypes, Object[] args);

        /**
         * Invokes asynchronously the method of the remote object referred by this
         * RemoteObjectRef with the given name by passing the array of
         * object as the arguments. This method immediately returns after setting up
         * the asynchronous method call without waiting its completion. The status
         * and result of the asynchronous method call can be queried and accessed
         * through the returned AyncMethodInvokationResult object.
         * As the method call takes place asynchronously the parameter validations
         * are also made asynchronously. So the same exceptions thrown by
         * invokeMethod method is only thrown when the result of
         * the method call is accessed asynchronously through the returned
         * AyncMethodInvokationResult object.
         */
        public AyncMethodInvocationResult invokeMethodAsync(
                String methodName, Object[] args);

        /**
         * Invokes asynchronously the method of the remote object referred by this
         * RemoteObjectRef with the given name by passing the array of
         * object as the arguments. The method of the object that has the given name
         * and that has a list parameters types exactly matching
         * paramTypes array is called. This method immediately returns
         * after setting up the asynchronous method call without waiting its
         * completion. The status and result of the asynchronous method call can be
         * queried and accessed through the returned AyncMethodInvokationResult
         * object. As the method call takes place asynchronously the parameter
         * validations are also made asynchronously. So the same exceptions thrown by
         * invokeMethod(String, Class[], Object[]) method is only thrown when the
         * result of the method call is accessed asynchronously through the returned
         * AyncMethodInvokationResult object.
         */
        public AyncMethodInvocationResult invokeMethodAsync(String methodName,
                Class[] paramTypes, Object[] args);

        /**
```

```
     * Invokes asynchronously the method of the remote object referred by this
     * RemoteObjectRef with the given name by passing the array of
     * object as the arguments with a "fire-and-forget" kind of semantics. This
     * method immediately returns after setting up the asynchronous method call
     * without waiting its completion. Nothing (return value, any kind of
     * exceptions thrown even due to invalid parameter) is returned related to
     * the result of the method call. Even if the arguments are invalid for the
     * method call no exception is thrown i.e. the caller will not be able to
     * know whether the method call completed successfully or even dispatched to
     * the remote object.
     */
    public void invokeMethodOneWay(String methodName, Object[] args);

    /**
     * Invokes asynchronously the method of the remote object referred by this
     * RemoteObjectRef with the given name by passing the array of
     * object as the arguments with a "fire-and-forget" kind of semantics. The
     * method of the object that has the given name and that has a list
     * parameters types exactly matching paramTypes array is called. This method
     * immediately returns after setting up the asynchronous method call without
     * waiting its completion. Nothing (return value, any kind of exceptions
     * thrown even due to invalid parameter) is returned related to the result
     * of the method call. Even if the arguments are invalid for the method call
     * no exception is thrown i.e. the caller will not be able to know whether
     * the method call completed successfully or even dispatched to the remote
     * object.
     */
    public void invokeMethodOneWay(String methodName, Class[] paramTypes,
           Object[] args);

    /**
     * Gets a local copy of the remotely hosted object referred by this
     * RemoteObjectRef.
     */
    public Object getObjectCopy() throws RemotingException;

    /**
     * Moves the remotely hosted object referred by this RemoteObjectRef from its
     * current host to the remote ObjectHostingService running on the given host
     * and port. After the remote object is moved all local references (including
     * this one) to the remote object is transparently updated so that users of
     * local references (RemoteObjectRef objects and transparent proxies)
     * do not need to do anything related to the movement of the object.
     */
    public void moveTo(String hostName, int port) throws RemotingException;

    /**
     * Deletes and removes the remotely hosted object referred by this. After a
     * call to this method any method call on this object results with
     * ObjectDeletedException being thrown.
     */
    public void delete() throws RemotingException;

    /**
     * Determines whether this RemoteObjRef refers to a valid and accessible
     * remotely hosted object.
     */
    public boolean isValid() throws RemotingException;

    /**
     * Determines whether this RemoteObjRef refers to a remotely hosted object
     * that is exported with a well known name.
     */
    public boolean isExportedObject() throws RemotingException;

    /**
     * Returns the <code>Class</code> object representing the type of the remote
     * object referred by this RemoteObjectRef.
     * If the class of the remote object is not loaded yet the class is
     * downloaded from the remote ObjectHostingService and loaded
```

```
         * into the current class loader.
         */
        public Class getType() throws RemotingException;
}
```

```
/**
 * Allows access to the result of a method call that is made on a remotely hosted
 * object. On remote objects method calls are made using invokeMethod methods of the
 * RemoteObjectRef objects which return MethodInvocationResult objects.
 */
public class MethodInvocationResult {
        /**
         * Returns the return value of the remote method call that returned this
         * object.
         */
        public Object getValue() throws RemotingException;

        /**
         * Returns whether an exception is thrown during the remote method call that
         * returned this object.
         */
        public boolean exceptionThrown();

        /**
         * Returns the exception that is thrown during the remote method call that
         * returned this object.
         */
        public Exception getException();
}
```

```
/**
 * Allows access to the completion status and result of a method call that is made
 * asynchronous on a remotely hosted object. On remote objects asynchronous method
 * calls are made using invokeMethodAsync methods of the RemoteObjectRef
 * objects which return AyncMethodInvocationResult objects.
 */
public class AyncMethodInvocationResult extends MethodInvocationResult {
        /**
         * Returns the completion status of the asynchronous remote method call
         * that returned this object.
         */
        public boolean isCompleted();

        /**
         * Returns the return value of the asynchronous remote method call that
         * returned this object. This method blocks until the asynchronous method
         * call is either completed successfully or resulted with an exception being
         * thrown.
         */
        public Object getValue() throws RemotingException;

        /**
         * Returns whether an exception is thrown during the asynchronous remote
         * method call that returned this object.
         * This method blocks until the asynchronous method call is either completed
         * successfully or resulted with an exception being thrown.
         */
        public boolean exceptionThrown();

        /**
         * Returns the exception that is thrown during the asynchronous remote method
         * call that returned this object.
         * This method blocks until the asynchronous method call is either completed
         * successfully or resulted with an exception being thrown.
         */
        public Exception getException();
        /**
         * This method blocks the caller until the asynchronous method call that
         * returned this object is either completed successfully or resulted with
         * an exception being thrown.
         */
        public void waitForCompletion();
}
```

A.3 Dynamic Proxies

```
/**
 * Helper class that provides services related to dynamic proxy classes and objects.
 * Proxy object provides the same interfaces as the remotely hosted object so
 * once they are obtained they can be cast down to any interface of the remote
 * object for which they act as the proxy. Client can make remote method calls
 * on remotely hosted object through the proxy instances as if the method call is
 * a local (in VM) method call. So the proxy objects provides access, location and
 * migration transparency for remotely hosted objects.
 * /
public final class ProxyHelper {
        /**
         * Generates a transparent dynamic proxy object to the remotely hosted object
         * referred by the given RemoteObjectRef. The returned object implements the
         * same interfaces implemented by the remote object referred by the given
         * RemoteObjectRef. So the returned object can be cast down to any interface
         * supported by the remote object.
         */
        public static Object generateProxy(RemoteObjectRef remoteObjRef)
                throws RemotingException;

        /**
         * Determines whether the given object is a dynamic proxy instance for a
         * RemoteObjectRef object.
         */
        public static boolean isProxy(Object obj);

        /**
         * Returns the RemoteObjectRef for which the given object acts
         * as a proxy. Clients can use this method to get a reference to a
         * RemoteObjectRef for low-level operations.
         */
        public static RemoteObjectRef getRemoteObjectRef(Object proxy);
}
```

A.4 Parallel Application Services Layer

```
/**
 * Provides the entry point of the Application Services as static methods for
 * initializing the sub-system, running applications, and doing some configuration.
 */
public final class ApplicationServices {
        /**
         * Initialized the Application Services system. Before doing anything with
         * the Application Services system the user application should call this
         * method.
         */
        public static void initialize();

        /**
         * Submits the given Application to the system for asynchronous execution.
         * The method does NOT block until all the Tasks of the given Application
         * is executed. The method returns when the <code>run</code> methods of the
         * given Application finishes.
         */
        public static void runApplication(Application app);

        /**
```

```
     * This method blocks and does not return until all the tasks of the given
     * application are completed (either successfully or with error).
     */
    public static void waitForCompletion(Application app);

    /**
     * Cancel all the Task of the given application that are submitted to the
     * system but that are not returned back to the Application yet.
     */
    public static void cancelApplication(Application app);

    /**
     * Gets the list of Brokers that Application Services is configured to use
     * to find Hosts to run Tasks on. This list contains Brokers that are:
     * 1. Configured to be used using Application Services configuration
     *    mechanism.
     * 2. Automatically discovered broker if configured to use them.
     * 3. Programmatically configured to be used using addBroker and setBroker
     * methods.
     */
    public static String[] getBrokers();

    /**
     * Gets the list of host names of Brokers that are automatically located by
     * Application Services. Note that this list of broker may or may not being
     * used by Application Services depending on the configuration of Application
     * Services (See methods enableUseOfDiscoveredBrokers and
     * isUsingDiscoveredBrokers).
     */
    public static String[] getDiscoveredBrokers();

    /**
     * Adds the given Broker to the end of the list of Brokers that will be used
     * to find Hosts to run Tasks on.
     */
    public static boolean addBroker(String brokerHostName);

    /**
     * Adds the given Broker to the list of Brokers (at the given position) that
     * will be used to find Hosts to run Tasks on.
     */
    public static void addBroker(int i, String brokerHostName);

    /**
     * Removes the given Broker from the list of Brokers that will be used to
     * find Hosts to run Tasks on.
     */
    public static boolean removeBroker(String brokerHostName);

    /**
     * Configures Application Services to use or not to use automatically
     * discovered Brokers.
     */
    public static void enableUseOfDiscoveredBrokers(boolean useDiscoveredBrokes);

    /**
     * Returns true if Application Services is configured to use automatically
     * discovered Brokers. Otherwise returns false.
     */
    public static boolean isUsingDiscoveredBrokers();

    /**
     * Returns the total number of Hosts that are believed (by the Brokers they
     * are registered to) to be available for running Task. Note that this number
     * is the not exact and is a best estimate.
     */
    public static int getAvailableHostNo();

    /**
     * Returns the total number of Hosts that are believed (by the Brokers they
         * are registered to) to be available for running Task. Note that this number
         * is the not exact and is a best estimate.
         */
        public static int getTotalHostNo();
}
```

```
/**
 * Abstract base class for all user applications. Applications that will be run
 * on Application Services must inherit this class and must at least provide the
 * implementation of the abstract method <code>run</code>.
 */
public abstract class Application {
        /**
         * Application implementers sub-classes this class and implement this
         * method in the minimum. Generally, in this method the user creates
         * instances of Tasks of its application and submit them to the system using
         * one of runTask, runTasks, or runTaskSynch methods of this class.
         */
        public abstract void run();


        /**
         * This method does nothing by default. It should be overridden and
         * implemented by Application implementers that want to receive notification
         * when a Task of their application completes successfully.
         */
        public void onTaskCompleted(Task task);

        /**
         * This method does nothing by default. It should be overridden and
         * implemented by Application implementers that want to receive notification
         * when a Task of their application stops with an error (exception).
         */
        public void onTaskStopped(Task task);

        /**
         * This method does nothing by default. It should be overridden and
         * implemented by Application implementers that want to receive notification
         * when a Task of their application reports progress.
         */
        public void onTaskProgressed(Task task);

        /**
         * Submits the given Task to the Safran system for asynchronous execution.
         */
        protected final void runTask(Task task);

        /**
         * Submits the given Tasks to the Safran system for asynchronous execution.
         */
        protected final void runTasks(Task[] tasks);

        /**
         * Submits the given Tasks to the Safran system for synchronous execution.
         * The method blocks until the Task get executed at some Host and its result
         * (either successful or error) retrieved.
         * Application's taskCompleted, taskProgressed, and taskStopped are NOT
         * called by the system for the given Task.
         */
        protected final void runTaskSynch(Task task);

        /**
         * Cancels the Tasks (which must belong to this application) that is
         * submitted to the system but whose result is not returned to the
         * Application yet.
         */
        protected final void cancelTask(Task task);

        /**
        * Cancels all the Tasks of this application that are submitted to the system
        * but whose results are not returned to the Application yet.
        */
       protected final void cancelAllTasks();
}
```

```
/**
 * Abstract base class for all user Task objects. User writing applications that
 * will be run on Application Services must provide at least one class inheriting
 * from this class and must at least provide the implementation of the abstract
 * methods run and copyFrom.
 */
public abstract class Task implements Serializable {
       /**
        * Application developers must implement the application specific Task
        * execution logic in this method. When the Task is dispatched to a remote
        * Host for remote execution, this method is called by the remote Host.
        */
       public abstract void run(TaskExecutionContext context);

       **
        * Task implementers must provide an implementation of this method. The
        * method should set the internal, application specific state (members) of
        * the Task object to the state of the given Task object. Functioning of the
        * framework depends on the correct implementation of this method so
        * applications developer must take extra care in implementing this method.
        */
       public abstract void copyFrom(Task task);

       /**
        * Marks this Task as being completed successfully.
        */
       public final void setCompletedWithSuccess();

       /**
        * Marks this Task of being completed with the given exception being thrown.
        */
       public final void setCompletedWithError(Throwable t);

       /**
        * Gets whether the execution of this Task completed on a remote Host
        * (either successfully or with error).
        */
       public final boolean isCompleted();

       /**
        * Gets whether the execution of this Task completed on a remote Host with
        * error.
        */
       public final boolean isCompletedWithError();
}
```

```
/**
 * Enables interaction of Task with their execution context while they are running
 * on remote Hosts.
 */
public interface TaskExecutionContext {
       /**
        * While executing, Task can call this method on safe points of its execution
        * to request from the system to checkpoint its state for systems fault
        * tolerance and recovery operation.
        */
       void checkpointTask();
}
```

APPENDIX B: SOURCE CODE OF THE TEST APPLICATION

Here we present the source code of the real matrix multiplication application that we used for our experiments (explained in **Section 5.1.1**) as example of parallel application built on Safran.

Matrix.java

```
package safran.tests.apps.realmatrix;

import java.util.Random;

public class Matrix implements java.io.Serializable {
        private static Random random = new Random(System.currentTimeMillis());

        int rowNo, colNo;
        int matrixData[][];


        public Matrix(int rowNo, int colNo) {
                this.rowNo = rowNo;
                this.colNo = colNo;
                matrixData = new int[rowNo][colNo];
        }

        public void initializeRandom() {
                for (int r=0; r<rowNo; r++) {
                        for(int c=0; c<colNo; c++) {
                                matrixData[r][c] = random.nextInt(10);
                        }
                }
        }

        public Matrix[] getSubMatrices(int count) {
                if (count<=0 || count>rowNo || rowNo%count!=0) {
                        throw new IllegalArgumentException("This matrix
                        cannot be divided in "+count+" sub-matrices.");
                }

                Matrix[] results = new Matrix[count];
                int rowPerSubmatrix =
                          (int)Math.ceil((double)rowNo/count);
                for (int i=0; i<count; i++) {
                        Matrix tmpMatrix = new Matrix(rowPerSubmatrix,
                                                              colNo);
                        for (int j=0; j<rowPerSubmatrix; j++) {
                                tmpMatrix.matrixData[j]=
                                this.matrixData[i*rowPerSubmatrix+j];
                        }
                        results[i] = tmpMatrix;
                }

                return results;
        }

        public static Matrix multiply(Matrix m1, Matrix m2) {
                if (m1.colNo!=m2.rowNo) {
                        throw new IllegalArgumentException("Row number of
                              the seconds matrix must be equal to the
                              column number of the first matrix.");
                }

                Matrix result = new Matrix(m1.rowNo, m2.colNo);
                for (int r=0; r<m1.rowNo; r++) {
                        for (int c=0; c<m2.colNo; c++) {
                                int tmpVal = 0;
                                for (int i=0; i<m1.colNo; i++) {
                                        tmpVal +=
                                m1.matrixData[r][i]*m2.matrixData[i][c];
                                }
                                result.matrixData[r][c] = tmpVal;
                        }
                }

                return result;
        }
}
```

MMApp.java

```
package safran.tests.apps.realmatrix;

import safran.applications.*;

public class MMApp extends Application {
        int taskCount;
        Matrix m1, m2;

        public MMApp(Matrix m1, Matrix m2, int taskCount) {
                this.m1 = m1;
                this.m2 = m2;
                this.taskCount = taskCount;
        }

        public void run() {
                Matrix[] m1SubMatrices = m1.getSubMatrices(taskCount);
                for (int i=0; i<taskCount; i++) {
                        MMTask task = new MMTask("RealMTask#"+i,
                                m1SubMatrices[i], m2);
                        this.runTask(task);
                }
        }

        public void taskCompleted(Task task) {
                System.out.println("TASK COMPLETED Task: "+task);
        }

        public void taskProgressed(Task task) {
                System.out.println("TASK PROGRESSED Task: "+task);
        }

        public void taskStopped(Task task) {
                System.out.println("TASK STOPPED Task: "+task);
        }
}
```

MMTask.java

```
package safran.tests.apps.realmatrix;

import safran.applications.*;

public class MMTask extends Task {
        String taskName;
        Matrix m1, m2, result;
```

```
	public MMTask(String name, Matrix m1, Matrix m2) {
		if (m1.colNo!=m2.rowNo) {
			throw new IllegalArgumentException("Given matrices
				cannot be multiplied.");
		}

		this.taskName = name;
		this.m1 = m1;
		this.m2 = m2;

		result = new Matrix(m1.rowNo, m2.colNo);
	}

	public void copyFrom(Task task) {
		MMTask remoteTask = (MMTask)task;
		this.result = remoteTask.result;
	}

	public void run(TaskExecutionContext context) {
		result=  Matrix.multiply(m1, m2);
	}

	public Matrix getResult() {
		return result;
	}

	public String toString() {
		return taskName;
	}
}
```

TestRunner.java

```
package safran.tests;

import safran.applications.ApplicationServices;
import safran.tests.apps.realmatrix.*;
import safran.tests.apps.datalessmatrix.*;

import java.util.Date;
import java.text.SimpleDateFormat;
import java.io.FileWriter;
import java.io.IOException;
import java.io.File;
import java.util.logging.LogManager;
import java.io.ByteArrayInputStream;


/**
 * The executable entry point of test code. Runs all tests.
 */
public class TestRunner {
	private final int REPEAT_NO = 5;

	long start, end;
	String fileSep, newLine, currentDir;
	SimpleDateFormat dateFormat;

	public TestRunner() throws IOException {
		start = 0;
		end = 0;
		fileSep = System.getProperty("file.separator");
		newLine = System.getProperty("line.separator");
		currentDir = System.getProperty("user.dir");
		dateFormat = new SimpleDateFormat("yyyyMMdd_HHmmss");
	}

	public static void main(String args[]) throws Exception {
		TestRunner tests = new TestRunner();
		tests.runTests();
	}

	public void runTests() throws Exception {
		runLocalRealMatrixTests();
		ApplicationServices.initialize();
		ApplicationServices.enableUseOfDiscoveredBrokers(true);
		runRealMatrixTests();
	}

	public void runLocalRealMatrixTests() throws IOException {
		int taskNo = 1;
		int hostNo = 0;
		String dirName = fileSep+"LRMT"+fileSep+"H#"+hostNo;
		String fileName = "T#"+taskNo;
		FileWriter resultFile = createFile(dirName, fileName);

		resultFile.write("Test Name:\tLocal Real Matrix Test"+newLine);
		resultFile.write("Host No:\t"+hostNo+newLine);
		resultFile.write("Task No:\t"+taskNo+newLine+newLine);
		resultFile.write("Task No\tMatrix Size (nxn)\tLocal Time"+newLine);

		final int minMatrixSize = 128;
		final int maxMatrixSize = 1152;
		final int sizeIncrement = 128;
		for (int matrixSize=minMatrixSize;
			matrixSize<=maxMatrixSize;
			matrixSize+=sizeIncrement) {
			System.out.println("Running test for Size: "+matrixSize);
			Matrix m1 = new Matrix(matrixSize, matrixSize);
			Matrix m2 = new Matrix(matrixSize, matrixSize);
			m1.initializeRandom();
			m2.initializeRandom();

			long localTime = 0;
			for (int repeat=0; repeat<REPEAT_NO; repeat++) {
				System.gc();
				start = System.currentTimeMillis();
				Matrix result=Matrix.multiply(m1, m2);
				end = System.currentTimeMillis();
				localTime += (end-start);
			}
			localTime /= REPEAT_NO;
			resultFile.write(taskNo + "\t" + matrixSize + "\t" + localTime
+ "\t" + newLine);
			resultFile.flush();
		}

		resultFile.write(newLine);
		resultFile.close();
	}

	public void runRealMatrixTests() throws IOException {
		int totalHostNo = ApplicationServices.getTotalHostNo();
		while (totalHostNo<=0) {
			totalHostNo = ApplicationServices.getTotalHostNo();
		}

		final int minTaskNo = 32;
		final int maxTaskNo = 32;
		final int taskNoMultiply = 2;
		for (int taskNo=minTaskNo;
			taskNo<=maxTaskNo; taskNo*=taskNoMultiply) {
			totalHostNo = ApplicationServices.getTotalHostNo();

			String dirName = fileSep+"RRMT"+fileSep+"H#"+totalHostNo;
			String fileName = "T#"+taskNo;
			FileWriter resultFile = createFile(dirName, fileName);
			resultFile.write("Test Name:\t Remote Real Matrix Test" +
newLine);
                        resultFile.write("Host No:\t"+totalHostNo+newLine);
                        resultFile.write("Task No:\t"+taskNo+newLine+newLine);
                        resultFile.write("Host No\tTask No\tMatrix Size (nxn)\tRemote
Time (ms)" + newLine);

                        final int minMatrixSize = 128;
                        final int maxMatrixSize = 1152;
                        final int sizeIncrement = 128;
                        for (int matrixSize=minMatrixSize;
                                matrixSize<=maxMatrixSize;
                                matrixSize+=sizeIncrement) {
                                Matrix m1 = new Matrix(matrixSize, matrixSize);
                                Matrix m2 = new Matrix(matrixSize, matrixSize);
                                m1.initializeRandom();
                                m2.initializeRandom();

                                long remoteTime = 0;
                                for (int repeat=0; repeat<REPEAT_NO;
                                        repeat++) {
                                        MMApp theApp = new MMApp(m1, m2, taskNo);
                                        System.gc();
                                        start = System.currentTimeMillis();

                        ApplicationServices.runApplication(theApp);
                        ApplicationServices.waitForCompletion(theApp);
                                        end = System.currentTimeMillis();
                                        remoteTime += (end-start);
                                }
                                remoteTime /= REPEAT_NO;


                        resultFile.write(totalHostNo + "\t" + taskNo + "\t" +
matrixSize+"\t"+ remoteTime+"\t"+newLine);
                                resultFile.flush();
                        }
                        resultFile.close();
                }
        }

        private FileWriter createFile(String subDir, String prefix)
                throws IOException {
                String dirName = "Data" + subDir;
                File dir  = new File(currentDir, dirName);
                dir.mkdirs();

                String fileName = dateFormat.format(new Date());
                fileName = prefix + "_" + fileName + ".txt";

                File file  = new File(dir, fileName);
                FileWriter writer = new FileWriter(file, false);
                return writer;
        }
}
```