ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

ITU-PRP: PARALLEL RUNNING PLATFORM
A PARALLEL PROGRAMMING FRAMEWORK FOR JAVA

M.Sc. THESIS

Enis SPAHI

Department of Computer Engineering

Computer Engineering Programme

NOVEMBER 2014

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

ITU-PRP: PARALLEL RUNNING PLATFORM
A PARALLEL PROGRAMMING FRAMEWORK FOR JAVA

M.Sc. THESIS

Enis SPAHI
(504091531)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Assoc. Prof. Dr. D. Turgay ALTILAR

NOVEMBER 2014

<u>**İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ**</u>

ITU-PRP : PARALEL İŞLEM PLATFORMU JAVA İÇİN PARALEL PROGRAMLAMA ARACI

YÜKSEK LİSANS TEZİ

Enis SPAHI
(504091531)

Bilgisayar Mühendisliği Anabilim Dalı Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Doç. Dr. D. Turgay ALTILAR

KASIM 2014

Enis SPAHI, a **M.Sc.** student of ITU **Graduate School of Science** student ID 504091531, successfully defended the **thesis** entitled "**ITU-PRP: Parallel Running Platform A Parallel Programming Framework for Java**", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc.Prof. Dr. D. Turgay ALTILAR** İstanbul Technical University	....
Jury Members :	**Prof. Dr. Nadia ERDOĞAN** İstanbul Technical University	....
	Assist.Prof. Dr. Yunus Emre SELÇUK Yıldız Technical University	....

Date of Submission : 9 October 2014
Date of Defense : 13 November 2014

FOREWORD

This thesis was written for my Master degree in Computer Engineering Department at Istanbul Technical University.

I would like to take this opportunity to thank following people, who helped and supported me during the writing process of this thesis. First I would like to express my gratitude to my supervisor Assoc. Prof. Dr. D. Turgay Altılar, for his patience and encouragement during different phases of this process.

Finally, I am grateful to my family and my girlfriend, without whose support this thesis would not be possible.

November 2014 Enis SPAHI

TABLE OF CONTENTS

ABBREVIATIONS

PRP	**:** Parallel Running Platform
TP	**:** Task Plan
NAT	**:** Network Address Translation
ITU	**:** Istanbul Technical University

LIST OF TABLES

Page

LIST OF FIGURES

Page

ITU-PRP: PARALLEL RUNNING PLATFOM, A PARALLEL PROGRAMMING FRAMEWORK FOR JAVA DEVELOPERS

SUMMARY

During past decades, developments on Web technologies have brought various new concepts on Distributed and Parallel Computing systems. Increased Internet usage along with evolved capabilities of high level programming technologies, leaded to new concepts such as Multi-Host parallel computing, task distribution, peer-to-peer programming, etc. Especially, Java as the leading programming environment used on Internet basis systems, attracted the attention of Parallel Programming Studies, which resulted with the invention of many Parallel Programming Frameworks. The lack of native Parallel Programming Frameworks to provide High-Scale Parallel systems, oriented parallel programmers to develop various solutions. Also, multi-host parallel systems have become reasonable alternative solutions over multi-core parallel systems.

The System built in these study aims providing a Parallel Programming Framework for Java Developers on which they can adapt their sequential application code to operate on a heterogeneous multi-host parallel environment. Developers would implement parallel models, by the help of an API Library provided under framework. Produced parallel applications would be submitted to a middleware called Parallel Running Platform (PRP), on which parallel resources for parallel processing are being organized and performed. The middleware creates Task Plans (TP) according to application's parallel model, assigns best available resource hosts, in order to perform fast parallel processing. Task Plans will be created dynamically in real time according to resources actual utilization status or availability, instead of predefined/preconfigured task plans. ITU-PRP achieves better efficiency on parallel processing over big data sets and distributes divided base data to multiple hosts to be operated by Coarse-Grained parallelism. According to this model distributed parallel tasks would operate independently with minimal interaction until processing ends.

ITU-PRP: PARALEL İŞLEM PLATFORMU, JAVA İÇİN PARALEL PROGRAMLAMA ARACI

ÖZET

Son yıllarda, Web teknolojilerinin kaydettiği gelişim ile birlikte Dağıtık ve Paralel İşleme sistemlerinde yeni kavramlar ortaya çıkmıştır. Artan İnternet kullanımı ve gelişen üst seviye programlama dilleri sayesinde, Mutli-Host Parallel Computing, Task Distribution, Peer-to-peer programlama gibi yeni kavramlar ortaya çıkmıştır. Özellikle İnternet tabanlı sistemlerin geliştirilmesinde öncü geliştrme ortamlarından biri olan Java, Paralel Programlama ile ilgili çalışmalarda yerini almaya başlayıp, çok sayıda Paralel Programlama arayüzünün ortaya çıkmasında rol oynamıştır. İlkel paralel programlama dillerinin kapsam genişletme ile ilgili yaşanan eksiklikler, paralel programlama uzmanlarının değişik çözümler üzerinde çalışmalarına yol açmıştır. Ayrıca, Multi-Host paralel sistemler Multi-Core (çok çekirdekli) paralel sistemler karşısında alternatif çözüm olarak değerlendirilmeye başlamıştır.

Tez kapsaminda tasarlanan sistem, Java Geliştiricilerine uygulamalarını heterojen ve Multi-Host çalışan bir paralel paralel platforma taşıyabilecekleri bir framework sunmayi amaçlamaktadır. Bu sisteme göre geliştirilen uygulamalar Multi-Host bir ortamda paralel çalışarak performans iyileştirme sağlanacaktır. Framework kapsamında sunulan bir API kütüphanesi, paralel modellerin gerçeklenmesini sağlayacaktır. Üretilen paralel uygulamalar Parallel Running Platform (PRP) olarak adlandırılan bir ara katmana yüklenerek kayıt altına alınacaktır. İlgili ara katman paralel işlemlerin gerçeklenebilmesi adına kaynak yönetimi ve tahsis etme süreçlerini gerçekleştirmektedir. Bu ara katman, uygulamanın paralel modeline göre hızlı paralel işlem gerçekleştirebilme adına, eldeki en müsait durumdaki kaynak Host'lardan oluşan bir Görev Planı (Task Plan) oluşturmaktadır. Görev Planı önceden belirlenmiş bir plandan ziyade, kaynakların anlık durum ve müsaitliğine dikkate alınarak gerçek zamanda dinamik olarak oluşturulacaktır. ITU-PRP sistemi esas performans artışını büyük data kümelerini bölerek çoklu hostlara dağıtıp işleyerek elde etmektedir. Coarse-Grained Parallelism olarak adlandırılan bu modele göre dağıtılmış paralel görevler birbirinden bağımsız bir şekilde işlem sonuçlanana kadar işlem yapacaklardır.

ITU-PRP paralel uygulama geliştiricilerine hem paralel çalışan uygulama geliştirebilecekleri hem de uygulamaların paralel çalışmasını sağlayan ortak bir çözüm sunmacaktır. ITU-PRP paralel yazılım geliştirme sürecinin basit, kolayca gerçeklenebileceği bir kütüphane sağlamayı amaçlamaktadır. İlgili kütüphane kullanıcılara bir JAR paketi halinde sunulmaktadır. İlgili kütüphane, implementasyon için gerekli arayüzlerden oluşmaktadır, ki bu arayüzler sıralı kodlamaya benzeyen özerk paralel görevler yazmaya imkan tanıyacaktır. Ek olarak, ITU-PRP paralel işleme için gerekli kaynakları hazırlayan bir ara katman sağlamaktadır. Task Execution Middleware olarak isimlendirilen bu ara katman sistemdeki değişken şartları göz önüne alarak dinamik kaynaklar sağlayacaktır.

Paralel Programlama Framework'ü paralel işlemleri soyutlayarak kullanıcıdan gizlemektedir. Multi-Host paralel işlemler de yine kullanıcılara yansıtılmayan paralel işlemlerdir. Kullanıcı parallel görevlerin dağıdımı, çalıştırılması, paralel görevlerin birleştirilmesi, sonuç toplam, senkronizasyon ve bağlantı konularıula uğraşmayacaktır. Sadece uygulamaya ve gerçeklenen implementasyona ait bazı parametre girişlerini yapacaktır. Kullanıcılar geliştirdikleri uygulamaları sisteme yüklemektedirler. Sonrasında çalıştırma işlemlerini gelecekte yapacaklardır. Sisteme yüklenen uygulamalar Task olarak adlandırılıp görev olarak değerlendirileceklerdir.

Kullanıcılar ITU-PRP sistemine kullanıcı bilgileriyle web tabanlı bir arayüz üzerinden giriş yapmaktadırlar. Bu web uygulama kullanıcılara Task Execution Middleware katmanının hizmetlerini sağlayacaktır. Yetkili kullanıcılar görev işlemlerini başlatmakta, ayrıca sonuçları ilgili ekranlardan görüntüleyebilmektedirler.

ITU-PRP sistemi web tabanlı bir sistem olarak tasarlanmıştır. İlgili web sayfalarında Java Applet teknolojisi kullanılarak Java eklentisi ve prosesi çalıştırılmaktadır ve paralel işlemler bu proses içinde başlatılmaktadırlar. Sisteme giriş sonrası, yaratılan applet prosesleri kullanıcıya ait lokal verileri toplayarak Task Execution Middleware katmanına iletirler ve kayıt altında tutulurlar. İlgili Proses ve Thread'ler boşta oldukları sürece kaynak Host olarak davranmaktadırlar ve görev ataması için hazırda beklemektedirler. Kullanıcı görev talebinde bulunduğunda ise ilgili proses Client olarak davranacaktır ve gerekli işlemleri yapacaktır.

ITU-PRP paralel çalıştırma kaynaklarının sağlanabilmesi için kullanıcılardan katkı beklemektedir. ITU-PRP sistemine bağlı bütün kullanıcılar potansiyel kaynak durumundadırlar. Sisteme bağlı Client'lar Host olarak kayıtlıdırlar, diğer kullanıcılara görev işlem talepleri için yardımcı görevini yürütmektedirler. Kullanımda olmadıkları süre içinde potansiyel kaynak durumunda kalacaklardır. Birçok bilgisayarın çoğunlukla boşta olduğu, tüm kaynaklarını kullanmadığı düşünüldüğünde, bu yaklaşım yüksek performans paralel uygulama gerçeklenmesi için bir yöntem olarak düşünülmüştür.

Proje kapsamındaki Paralel Processing yöntemi Multi-Threaded task dağıtım modeline göre gerçeklenmektedir. ITU-PRP tasarımı, bir ana görevin çok sayıdaki alt görevi paralel bir döngü ile Host'lara dağıttığı, nesne yönelimli paralel modellerden oluşmaktadır. Bu tasarıma göre alt görevler paralel işlenmektedir. Sistemin nesne yönelimli olması, diğer ilkel paralel modellere göre bir artı olma özelliği taşımaktadır.

ITU-PRP'nın veri paralelleştirme yaklaşımı kullanıcıya özelleştirilebilir bir yapı sunma yönündedir. Veri dağıtımı işleminin alt sınıfa Object Serialization yapılarak uygulanması kullanıcının veri paralelleştirme üzerinde kontrolünü sağlamaktadır. Veri dağıtımın nesne bazlı olması, mesaj bazlı yapıya kıyasla kullanıcıya alt görev veri tiplerini belirlemesi açısından esneklik sağlamaktadır.

ITU-PRP için tasaralanan peer-to-peer protokolü proje kapsamını belirleyen önemli bir unsur olmuştur. İnternet üzerinde ağ erişim kısıtlamaları göz önüne alındığında farklı peer'lerin birbirlerine bağlanması zorlayıcı bir çalışma olmuştur ve bu konuda derin literatür araştırması yapılmıştır. 3 farklı NAT traversal tekniği olarak bilinen Relaying, Connection Reversal ve UDP Hole Punch Teknikleri kombine edilip internet üerindeki farklı Host'ların birbirine bağlanması sağlanmıştır. Yapılan tasarıma göre, bütün Host'lar birbirine bağlanacaktır ve bağlantılarını sisteme bağlı oldukları sürece aktif tutacaklardır. Ek olarak, Peer gruplama kavramı bölgesel host gruplarının oluşturmayı veya belirli görev gruplarının oluşturulabilmeyi amçlamaktadır. Ancak, bu uygulama sistemde kullanıcı sayısının artmasıyla daha anlamlı olacaktır, gelecekte yapılmak üzere şimdilik kapsam dışında bırakılmıştır.

Deney sonuçları minimal sayıda kaynak Host'la yapılan Paralel Uygulama çalıştırmanın bir performans artışı sağlamadığını gstermiştir. Ancak, kaynak Host sayısının artmasıyla birlikte deney sonuçlarının istenilen performans artışını sağladığı gözlemlenmiştir. Ek olarak, kaynak sayısı attıkça ağ iletişim gecikmelerinin etkisinin azaldığını göstermiştir. Ayrıca, Sequential çalıştırılma süreleri nispeten yüksek olan uygulamaların ITU-PRP'ye uyarlanmasının daha olumlu performans iyileştirme sağlayacağı görülmüştür.

Deneyler sırasında en yüksek Client-to-Host bağlantı gecikmesi 360ms olarak hesaplanmıştır. Ancak, gerçek Dünya'daki ağ iletişim sorunları göz önüne alındığında bu değerin daha yüksek olabileceği düşünülmektedir. Prensip olarak 2 saniye üzerinde sequential çalıştırma süresi olan uygulamaların ITU-PRP'nin paralel kalıplarına uyarlanması önerilmektedir. Çalıştırma süresi arttıkça, ITU-PRP'ye üzerinde başarım artışı daha fazla beklenmektedir. Her ne kadar ITU-PRP bir paralel programlama aracı olsa da, performans için diğer paralel programlama araçlarıyla yarışmamaktadır. ITU-PRP'nin asıl amacı global bir ortamda yüksek başarımlı uygulamaların çalıştırılabileceği Multi-Host ve heterojen bir sistem sağlamaktır. Ayrıca, kullanıcılar için bireysel işemci kaynaklarının global işlemci kaynağına dönüştürülmesi amaçlanmaktadır.

Farklı Paralel programalama framework'lerinin performans karşılaştırmalarının yapılması için kullanılan Benchmark araçlarının ITU-PRP'nin performans ölçümleri için kullanılabilmesi sözkonusu olamamaktadır. Bunun nedeni heterojen ve Multi-Host çalışan ve global peer-to-peer protokolüyle çalışan farklı karakterli bir sistem olmasıdır. Ayrıca, üst kısımda da bahsedildiği üzere, ITU-PRP'nin amacı başka ürünlerin performansları ile yarışmak değil, farklı karakteristikleri olan özgün bir sistem sağlayarak, maliyetsiz, yüksek başarımlı ve global bir sistem sağlamaktır.

Daha etkin bir sistem ancak yeterli sayıda kullanıcının katılımıyla sağlanabilecektir. Bu amaçla bir kullanıcı katılım ödüllendirme sistemi kurulacaktır. Kullanıcı sayılarının artmasıyla birlikte daha gerçekçi performans ölçümleri yapılım geleceğe yönelik sonuçlar incelenecektir ve ona göre sistemde iyileştirmeler yapılacaktır.

1. INTRODUCTION

ITU-PRP provides an all-in-one solution for Parallel Programmers, with a Parallel Programming Framework and a Task Execution Middleware within a single system. ITU-PRP intends a simple way for Parallel Application Development, which makes Parallel Code easy to implement through a Java Library released as JAR Package. The regarded library contains implementable interfaces, which would generate autonomous parallel tasks written as sequential code blocks. Parallel tasks are operated according to Loop Parallelism and Divide and Conquer parallel models [1]. Additionally, ITU-PRP's distributed middleware provides resources for parallel processing and ensures execution of tasks. Computing resources are assigned dynamically according to System's real time conditions.

Parallel Programming Framework mostly encapsulates parallel operations and provides abstraction to developer. Multi-Host parallel operations are handled by the encapsulated package. Developer will not deal with Parallel Task Distribution, Task Execution, Task Reunification, Result Collection, Synchronization and Connection issues. Only some initial parameters regarding to task execution are required to be set as configuration on the implemented code. User will configure his application on the regarded platform with parameters specified for parallel task execution. Any user submits its produced applications for future task execution, request for task execution and collect execution results. Submitted applications are treated as tasks in the system, so once an application is submitted to system, it will be named as Task.

Users with their accounts for ITU-PRP System will connect to system through a web based graphical user interface. This web application would serve users for their operations on ITU-PRP System, especially on Task Execution Middleware. Authenticated user initializes task execution and views the results of parallel processing through a specified screen.

ITU-PRP system is designed as a web based system, which mainly utilizes Java Based Applet technology and does parallel processing operations on user's Web Browser. Prior to system log on, initialized Applets processes gather user information and are registered to Task Execution Middleware. The Process and Threads created on user's process behave as hosts during their idle states and wait assignments of a task execution. If user requests for Task Execution, main process will behave as client and do operations accordingly.

ITU-PRP expects contribution in terms of execution resources from any user using the platform. Any user logged in to ITU-PRP will be considered as potential resource. Connected clients are registered as hosts as well, in order to make possible serving other Task Execution Requestor clients. Hosts will be available as potential computational resources during their idle times. Considering that many computers are mostly idle, the approach of this research has been utilization of non-used executional power in order to achieve high performance parallel applications.

ITU-PRP System provides Parallel Programming Framework and Task Execution Middleware. Figure 1.1 illustrates operations within these two main entities.

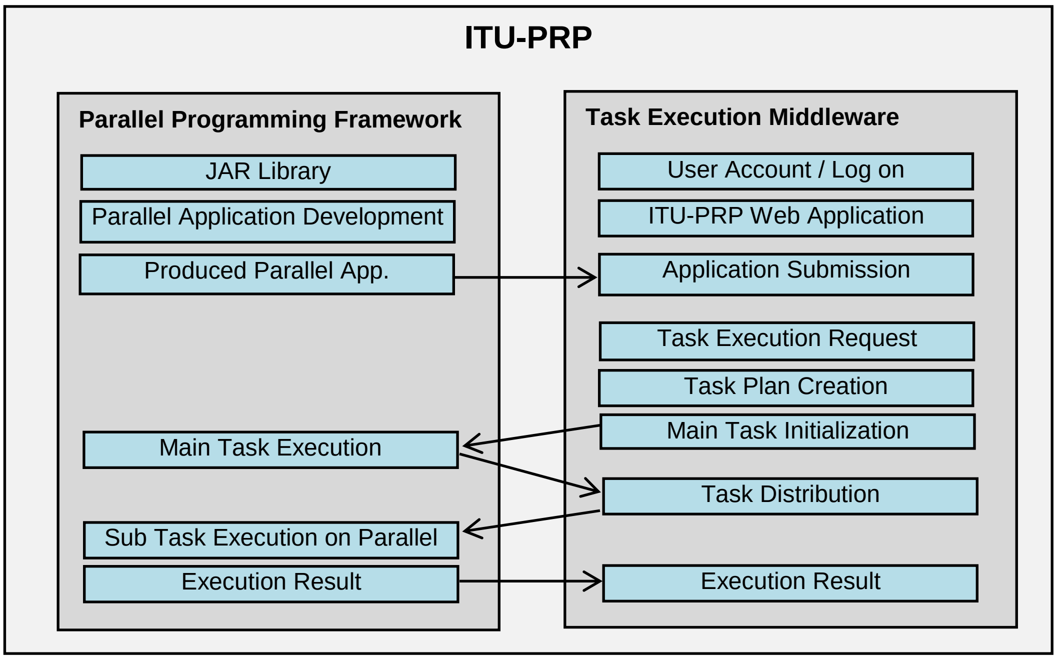


Figure 1.1 : ITU-PRP framework services.

Code parallelization for clients is achieved via a JAR Library provided to Java developers. This library would function as a framework implementable on development tools like Eclipse and etc.

1.1 Parallel Programming Framework

Parallel Programming Framework basically provides parallel code blocks for application developers. User adapts his application code to patterns specified by this framework. Provided JAR Package is named as Parallel Programming Library, which is implementable by the user according to specifications on Implementation Guidelines. The produced application will be uploaded to ITU-PRP Web Application, which is a unit of Task Execution Middleware. Besides the implementation, execution of Main Task, Parallelized Sub Task Execution and result generation are background operations hidden from the user.

The essential concerns of Application Framework can be listed as follows:

- **Easy Implementation:** Provide an easy to implement Parallel Programming Library to users. User's application is developed as sequential code blocks, but operate on parallel.
- **Simplicity:** Framework aims simplicity for parallel developers not being too much familiar with parallel programming.
- **Scalability:** Task executions running parallel on 2 hosts or on 10 hosts would not make any difference on implementation. This would be a configuration issue on Task Execution Middleware.
- **Performance:** Even performance is much more a concern of Task Execution Middleware, Application Framework also has some optimizations for getting better results during executions. Especially data sizes during task distribution are kept in small pieces. Caching mechanism of Java during applet execution also makes application execution faster after first time execution. Peer-to-peer communication instead of Centralized communication also optimizes the performance.

1.2 Task Execution Middleware

Applications developed on Parallel Programming Framework are uploaded to Task Execution Middleware. Once an Application is submitted to Parallel Running Framework, it becomes a registered application on the repository. Authorized users which have credentials are able to request execution for their application. Task execution requests are processed by the Broker and responded to client with a Task Execution Plan with assigned hosts and peers providing multi-host parallel execution. As soon as the Task Execution Plan is prepared, Broker calls the main task of the application executed on client's Java Applet process. Client's main task will distribute parameter information to assigned host's by communicating on peer-to-peer protocol. This is the Task Distribution phase specified by Task Execution Middleware but operated by Parallel Programming Framework. After the finalization of task execution, the main task will respond with an execution result to the user.

The essential concerns of Execution Framework:

- **Security:** Task repository is controlled under a permission mechanism. The user is able to execute only its own applications, check the results of its own task execution requests. Other users serving as hosts during their idle times would notice activity on their Java Applet processes, but calculation data and results are hided, unless the owner of application has put some output logs during development of it.
- **Performance:** Broker as coordinator of Task Execution should prepare such Task Execution Plans that it should predict the behavior of hosts in terms of performance and network delays during task distributions. Task Repository has got complexity information about the application. Also information regarding to available hosts are available on Host Registry. IP Addresses, Country, City, Location Info, Response Time, intensity, CPU and Configuration information are registered and updated initially and periodically. Broker does consider both information regarding to Applications and hosts, in order to prepare the best available Task Execution Plan to the client.
- **System Learning:** It's mentioned that information kept on Host Registry and Task Registry will guide the Broker to prepare the best possible task plan. But on some cases the prepared Task Plan may work better or worse than expected. System will also keep classified statistical information like execution time, host based individual execution times, network latency times or host based cooperation times. Multiple occurrences of executions will more valuable information on Task Execution performances. After multiple

executions of a Task, System will prioritize statistical information over Host Registry and Task Registry during Task Execution Plan creation.

2. RELATED WORK AND MOTIVATION

Java with its capability in developing Client-Server, Internet Based and Peer-to-peer application systems has been widespread in past few years. Growth of Java attracted the attention of Parallel Programming community to develop Parallel Programming frameworks adaptable in Java. The tendency of making Java programs to do parallel operations, aimed taking the advantage of Java capabilities, create Internet Based Parallel Applications easily. Parallel API Interfaces like Message Passing Interface (MPI), Parallel Virtual Machine (PVM), OpenMP [4] implementable in native languages (C/C++, Fortran) were lacking on wide range distribution over internet. On the other hand, also Java community needing for capability of Parallel Computing, in order to benefit computational resources over internet.

Several parallel programming API interfaces have shown up during recent few years. Mainly, existing parallel programming API interfaces are categorized in 3 main groups:

- **API Interfaces Derived From Native Interfaces**
- **API Interfaces derived from Java native thread models**
- **Java Applet Based Parallel Systems**

2.1 API Interfaces Derived From Native Interfaces

This category involves API interfaces derived from native C/C++, Fortran interfaces like MPI [4], PVM. Wrappers over MPI, PVM implement directives, provide adapted implementations. jPVM, MPJ-Express [5], Java MPI [6] are some of the existing ones.

API Interfaces provided under this category may face some issues regarding to heterogeneous platforms. Due to implementations requiring the explicit use of C/C++ or Fortran native libraries, platform dependency may be an issue or custom configurations may be required to run the same implementations on different platforms. This argument is considered against Java’s platform independence feature. Also, error handling and security may be another issue due to lack of control over Native methods.

2.2 API Interfaces Derived From Java Jative Thread Models

This category involves interface models developed from Java native Threads and communication protocols provided by Java. JOMP and JaMP [7] API interfaces have their directives adopted from OpenMP. JADE (Java Agent Development Framework) [8] as another specific Framework implemented on Java, provides a framework for Parallel Processing. It provides implementation of multi-agent systems through a middleware specified by FIPA. FIPA is the organization of standards for agents and multi-agent systems. Users create Agents, which will be able to run on remote hosts. Agents are task entities able to travel between hosts, do autonomus operations and get the results back to the initiator hosts. This methodology is especially suitable for Coarse-Grained Parallelism.

2.3 Java Applet Based Parallel Systems

This category involves Internet Based technologies utilized to use distributed parallel resources. Java Applets are application structures running on Browser. Embedded Java Applets on Web based Applications, create processes running on client computers. Applets are downloaded from the regarded URL to browser’s cache during first initialization and will behave like applications installed on client’s computer. Systems like Javelin [9][10], JAVM [11] implement Java Applets in order to use computational resource on a wide range network over Internet.

The drawbacks of using Java Applet based Parallel Systems may be limitations and security constraints on client computers. However, scalability is considered the main benefit of utilization of Java Applets.

2.3.1 Javelin

Javelin as a Java-Based parallel Infrastructure, provides a Java Applet based architecture for parallel computing. Javelin has 3 main entities of design for parallel processing, clients, Hosts and Brokers. Client is a process seeking computing resources, Host is a process offering computing resources and the Broker coordinates computing resources for clients. Client opens a URL on the Web Browser, on which an applet is created on client's computer. Applet initializes listener for tasks incoming from the broker. Client and Host may transform to each other, depending on activity, in case when client is idle it transforms to a host waiting for incoming tasks from broker. During task assignments, broker sends URL of the Applet process to be executed on host computer. Check Figure 2.1 for steps involved during Applet execution cycle.

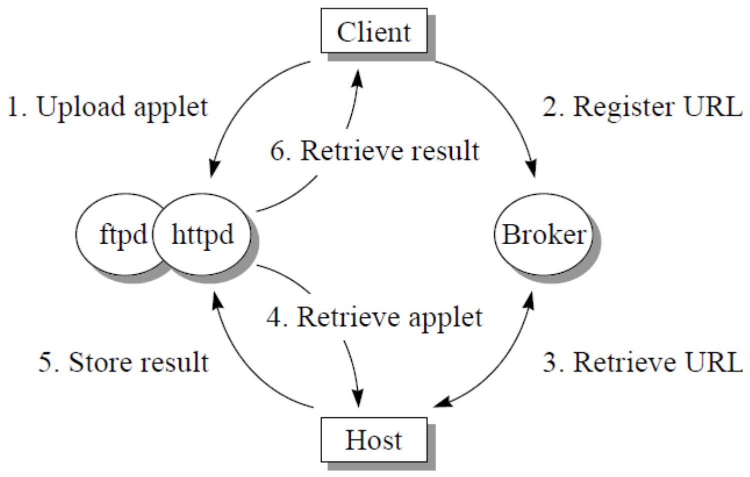


Figure 2.1 : Javelin – Steps involved for applet execution [9].

According to applet execution scenario on Figure 2.1, Javelin goes through this steps:

1. Client uploads the applet to HTTP Server.
2. Client registers the corresponding URL with Broker.
3. Host getting task notification retrieves the URL for execution.
4. Host downloads applet from server and makes the execution.
5. Host stores the result to server after execution.
6. Client gets the result of execution.

According to the opinion of authors in [9], server is required to function as a gateway for communicating client with hosts. This is a consequence of Applet security, of which its stated that applet cannot open a network connection to any computer other than the server where applet is downloaded. That's the reason why client does not communicate directly the listeners on hosts, instead Broker notifies Hosts for task assignment and sends the URL of the executable task in Applet form. All communication is routed through server, which is a potential single server a bottleneck. In order to minimize bottleneck in centralized server communication, applets communicate Broker based on HTTP connection to server. Applet connects to servlets implemented on the broker.

A newer version of Javelin, Javelin 2.0 [10] has some optimization compared with Javelin [9]. For instance multiple Brokers are utilized in order to achieve scalability and to prevent single server bottleneck. On the other hand host availability graph is sent to all hosts on response of the Broker. Hosts have information about other Brokers in cases when they ask for execution resource in case becoming clients. Mentioned graph consists state information on status of hosts and the status of task execution. The regarded graph is sent as multicast messages to hosts.

2.3.2 JAVM

JAVM (Java Astra Virtual Machine) [11] is an Internet-based parallel computing framework designed with pure Java implementation. JAVM is implementable on standalone Java applications. JAVM aims connecting users over Internet, which may be available as computational power. JAVM considers computational power as pool of machines, which are idle most of the time, offering more computational power than what a central parallel supercomputer can offer. JAVM involves 2 groups of users, the programmers who develop the parallel applications and the volunteers who contribute their machines for the computations.

Scenario of JAVM System interaction between entities is represented on Figure 2.2.

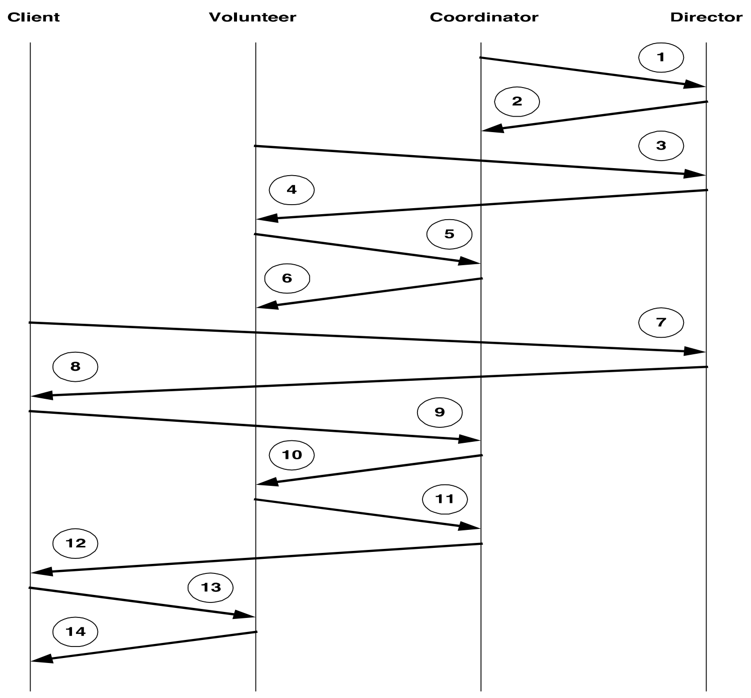


Figure 2.2 : Interaction among JAVM entities [11].

According to interaction scenario represented on Figure 2.3:

1. Coordinator entity registers itself to Director
2. Director responds to Coordinators registration request
3. Volunteer during initializing itself, asks director for active coordinators.
4. Director gives back information of active coordinates.
5. Volunteer selects one of the coordinators and provides its hardware information as computational resource information.
6. Coordinator assigns a Volunteer ID to the volunteer and informs the Volunteer.
7. Client asks director for available coordinators during start up.
8. Director provides active coordinators.
9. Client will select a coordinator and ask for Volunteers for computational purposes.
10. Coordinator will check available volunteers and will inform any assigned volunteers with the session information of client.
11. Volunteers will acknowledge the coordinator for assignment.
12. Coordinator will send the list of assigned volunteers to client.
13. Client will send task assignments to Volunteers in terms of application to run.
14. Volunteer will run the task and send the execution result to the client.

According to the opinion of the authors on [11], the design of the JAVM handles only master-slave style of parallelism. This implies that tasks executed on participating volunteers, would not be able to communicate with each other during computation. This means that coarse-grained parallel computing model is being implemented. However, the authors of [11] have mentioned on further enhanced to support peer-to-peer communications among volunteers and parallel applications based on tree computing model.

2.4 Motivation

As stated on the beginning of this section, Internet based parallel computing systems have become trending studies among the community. What arguments motivated us for developing ITU-PRP, in common with existing systems are as follows:

- Various Internet based Parallel Programming Frameworks implemented on Java, have the main idea of benefitting hosts as potential computational power during their idle states, providing computational power during their request for parallel execution. Considering that hosts would be idle on the most of their lifecycle, which is reasonable for using as computational power.
- Java applet is another useful model utilized for running tasks on distributed hosts. Applets are downloadable form of applications, which are initialized on web browsers of the user and run on the users computer just as described on section 2.2.
- Java as platform independent environment will also make possible make the parallel applications run on any platform with the philosophy of ”write once, run anywhere” [11].
- Another concern of such Frameworks is to provide opportunity of implementing parallel applications easily, with the concept of Automatic Parallelization. User should be able to parallelize its sequential code through Parallel Framework’s capability.

The arguments that motivated and made us excited about ITU-PRP study, differently from other existing systems are:

- ITU-PRP’s model of execution involves assigned computational hosts to communicate each other via peer-to-peer protocol during Parallel task execution, instead of a centralized Server coordinating communication traffic. Our design of peer-to-peer principal of communicating hosts with each other optimizes performance during Parallel Computing operations.
- Adaptability for heterogeneous and scalable platforms
- A low-cost global processing power
- Automatic parallelization is implemented via an object-oriented pattern prepared for parallelization.
- ITU-PRP offers implementations both for applet based and standalone applications.
- Java Applet security restrictions regarding to peer-to-peer communication may be overcome by doing policy configuration on Java.
- Broker as the coordinator entity makes real time decision for computational resource assignment by a special scoring system, which uses value factors such as host IP address locations, response times, declared computational power and retrieved computational power.
- Priority and special scoring for contributor clients named as Volunteer Reward System and additional scoring on Statistical Performance Records for recent executions.

3. ITU-PRP ARCHITECTURE AND DESIGN

ITU-PRP System is designed with the goal to provide users an all-in-one platform with separate self-functioning components integrated for a single purpose, realizing high performance parallel execution by easy implementation. ITU-PRP as an integrated Web Based System is implemented under layered architecture with several components. Figure 3.1 illustrates components and the technologies used for implementation within layered architecture.

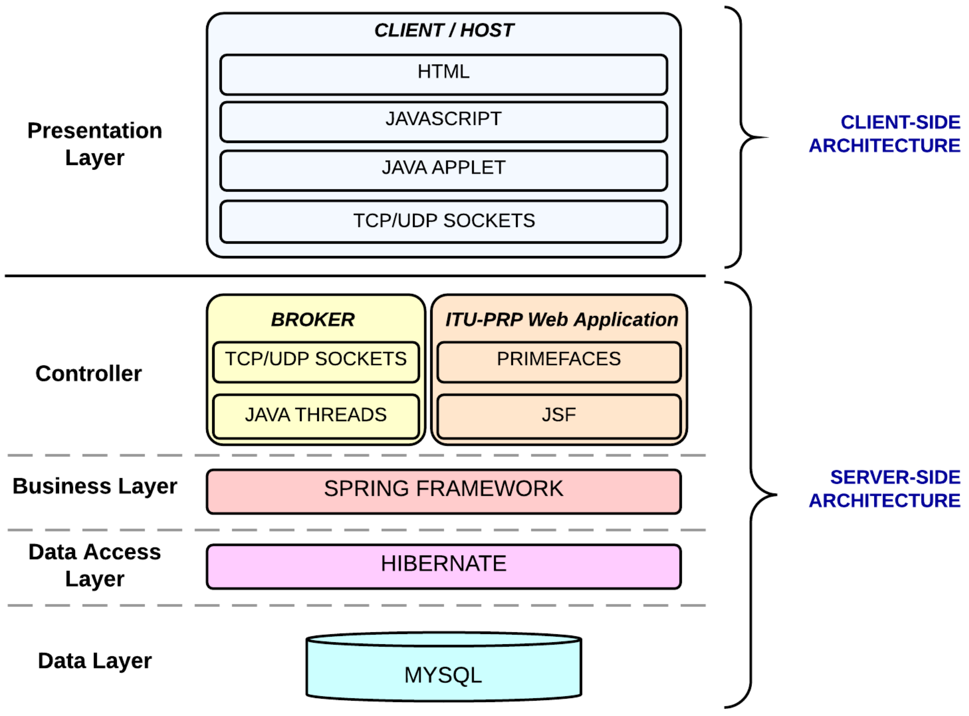


Figure 3.1 : ITU-PRP architecture.

The key point for creating the regarded architecture was selection of open source and standard technologies. Especially, various Java technologies such as Spring Framework, Hibernate, JSF, PrimeFaces, Java Threads, Java Socket programming methods and Java Applets are utilized for realization of the Web Based integrated system. MySQL was also utilized as the Relational Database tool, which is also an open source product. Hibernate is selected as the component on on data access layer in order to benefit its Object-Relational Mapping feature which makes possible mapping MySQL tables to Java classes. Java Spring Framework was utilized due to its features, which make accessing data access layer from upper layers easy. Front end of ITU-PRP Web Application is implemented by JSF framework and PrimeFaces component library. Broker, which is one of the key entities within the system, is implemented with Java's TCP/UDP Socket components and Thread based operations are realized by Java standard Threading libraries. Client side operations which consist Presentation Layer are realized by Java Applet technology.

Server-Side Architecture is hosted and deployed on Apache Tomcat Web Server hosted on a globally accessible Server. Client-Side architecture works on Web Browser, usable without installing any additional application on client or host's computer, which totally fits ITU-PRP's vision of wide usage and scalability. Application will work on any Java enabled Web Browser which is active in default, unless user or any security application disables Java Applets. These structure aims the system to make widely used without special requirements.

ITU-PRP consists three main entities in terms of system design, which are shown on the diagram in Figure 3.2:

- **Clients**
- **Hosts**
- **Broker**

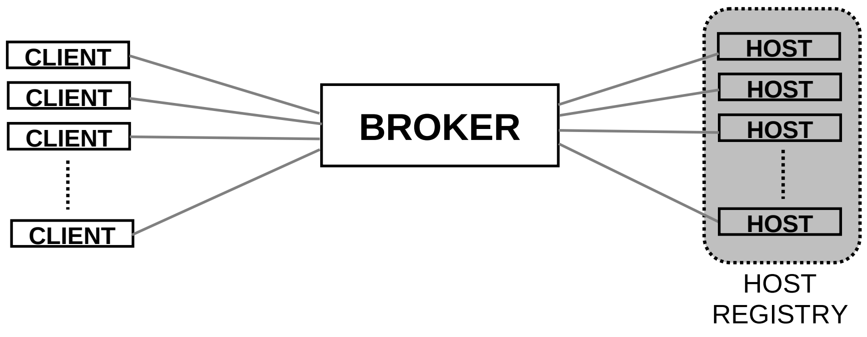


Figure 3.2 : ITU-PRP design.

Broker, Client and Hosts operate within an integrated Web Based System, each one with its own role. The system will assign a role to each user that connects the system. Initially, user will be redirected to a web based application, on which a Java Applet will be initialized and will create a process and a set of threads for parallel processing interaction. Broker as the coordinator entity, manages hosts and client activities. Broker is implemented as a set of Java Threads with active TCP socket connections to client and hosts. On the other hand clients and hosts are implemented through Java Applet technology. In order to make Java Applet to function, some security configuration on client's java configuration files is required. Java.policy file set up is the most known security policy configurations for Java Applets. Detailed configuration is explained on appendix.

3.1 Client

Application requesting Parallel execution will send parameters to Broker, which may be usable as decisive information about Task Planning. Client getting Task Plan as response will through the regarded parts of application code, divide data to fractions, send to hosts which are interconnected as P2P. Client creates Threads as subtasks, which will wait for the calculation results from other hosts. Distributed tasks sent to hosts are collected and reunited on Threads created during task distribution. Clients and hosts communicate with P2P sockets. In order to make less overhead interchanged data streams are kept as small byte buffers.

3.2 Host

Hosts are registered on Host Registry during the initial connection to System. IP Addresses, Country, City, Location Info, Response Time, intensity, CPU and Configuration information of hosts are saved on Host Registry in order to be considered for decision purposes during Task Plan creation phase. This information will be retrieved by Broker and saved to Host Registry, in this case there host will not be sending its information to Broker which will reduce overhead on host.

3.3 Broker

Serves clients requesting parallel operation by providing assigned resources and task plans. Clients complete their initialization by creating subscription requests to Broker. Incoming connection requests from clients are processed, by creating a record for each of them. Records are added to Host Registry, which is characterized as collection of available hosts for Task Executor clients. Records on Host Registry may behave both as client or hosts depending on the activity status of the client. If a client is on the state of requesting an execution plan, acts as client. On the other hand, if any subscribed client is on idle state, it acts as host available for Parallel Execution resource for other clients. Client connecting to Broker during the initialization is registered as Host-to-Host Registry. Also, hosts leaving the system are removed from Host Registry. Clients requesting parallel resource from Broker, should have provided required parameters about application’s complexity, data size, its own resource power and parallel task model, so that Broker may decide on assigned resources. Sufficient number of qualified hosts according to the requirements with given parameters are provided to requesting client as Task Plan. The initial execution of the Application running on client, will be followed by the distribution of tasks to hosts on the regarded parts of the application code, then execution results will be collected on initial client. Client and hosts will do task distribution and result collection operations through a special P2P messaging protocol designed for this system. Broker may refuse request of client, if it decides that a Multi-Processor or Sequential execution would be more optimal rather than the Multi-Host task execution. This usually may occur on the cases when clients own resource power would make execution more optimal rather than a Multi-Host task execution or there are no sufficient available host resources. Broker is implemented as a Java Thread listening to incoming TCP socket connections from Clients/Hosts and keeps active socket connections for interactions during host activities.

3.3.1 Host registry

Registered hosts on Host Registry go through some information retrieval phases during their lifetime. This is achieved by some threads, which collect periodic information.

- Broker adds host name information generated by the host. This would be hash information of the host while doing operations on the Broker.
- Broker registers the host session for the user name which is logged on to the System
- Broker retrieves IP address of hosts and clients connected.

- Broker sends tiny TCP/IP messages to hosts on periodic intervals, so that potential response times of resource hosts can be calculated. A kind of Ping and Pong messaging are sent in order to calculate response times..
- Based on Host's registered IP numbers Country, City, Region, Longitude/Latitude, Time information is collected from a third party IP Information Service provider. Check below URL for information about IP Information provider.
- *http://api.ipinfodb.com/v3/ip-city/?key=24e8ee217d936586d72aa698e044c75ec56300801da99373def22955b76b8468&ip=78.180.100.160&format=xml*
- Information about host's hardware is sent by host itself. Free memory and number of processors are registered as host hardware information to the Broker.

Check Data Structure of a Record on Host Registry on Table 3.1.

Table 3.1: Record on Host Registry.

Information	**Explanation**	**Example**
Host Name	This information is generated by the Host and sent to Broker during Host registration	*26951e56-3b18-4efd-82de-ef2c3f339b8d*
User	User Name of the Client or Host	*genericuser*
IP Address	Broker Retrieves the IP Address of the Host doing the registration	127.0.0.1
Country	Got from IP Address	
City	Got from IP Address	
Lattitude / Longitude	Got from IP Address	
Time Zone	Got from IP Address	
Response Time	Renewed Periodically through PING / PONG messages	*1 ms*
Free Memory	Free memory declared by Host	*111720208 (Byte)*
Available Processors	Available Processor count declared by Host	*4 (core)*
Active	Broker switches the Host to Client or Client to Host depending on the active role of Host	*true/false*

3.3.2 Host registry operations on Broker

Broker accepts host registration and host unregistration requests from clients and updates Host Registry accordingly. A client sends a registration request to the Broker as soon as the Java Applet is initialized on the client computer and the initial process is created. The Broker will create a host instance and register to mapped list on Host Registry according to the sequence diagram in Figure 3.3.

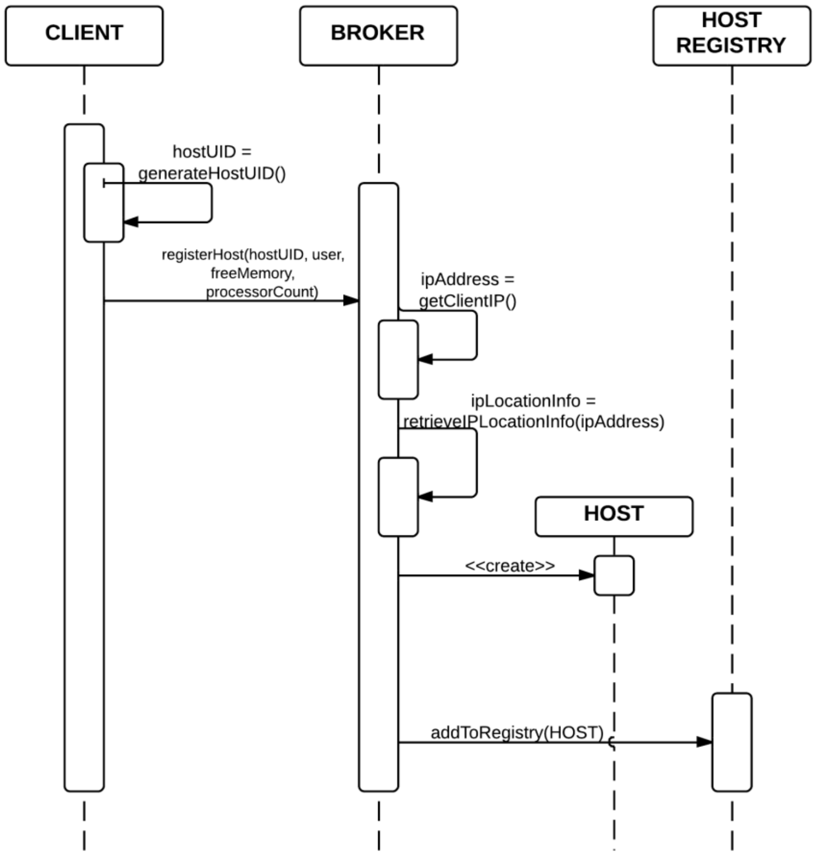

CLIENT
BROKER
HOST REGISTRY
hostUID = generateHostUID()
registerHost(hostUID, user, freeMemory, processorCount)
ipAddress = getClientIP()
ipLocationInfo = retrieveIPLocationInfo(ipAddress)
HOST
<<create>>
addToRegistry(HOST)

Figure 3.3 : Host registration.

Unregistration of hosts is another operation performed by Broker in order to keep Host Registry up-to-date for task assignment operations. Client Applet initiates Unregister host operation during destroy event of Applet, which is triggered prior applet process ending. Check Figure 3.4 for the sequence of operation performed during unregister host operation.

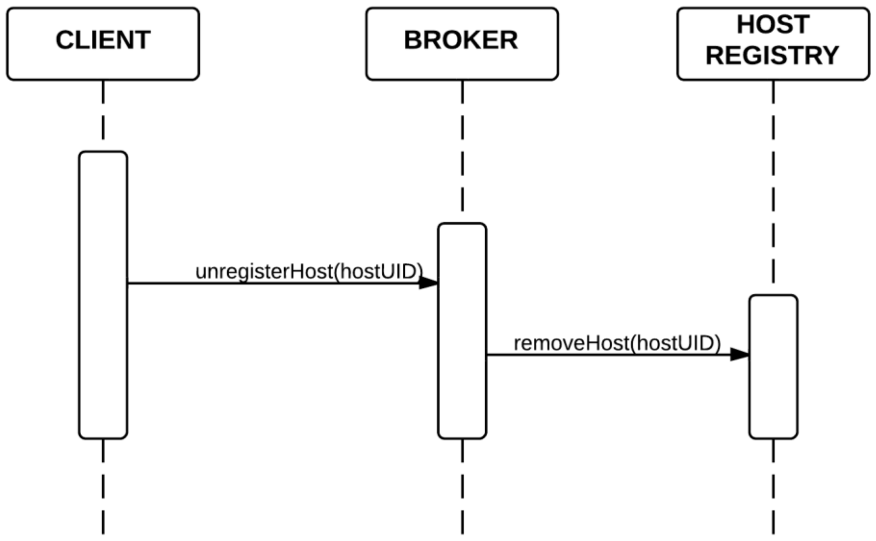


Figure 3.4 : Host unregistration.

List of available hosts is updated frequently and considered as the list of potential resource hosts for task assignments. A snapshot of Host Registry logged by the Broker's Application Server during different phases of operations would as on Figure 3.5.

```
HOSTS
---------------------------
1. HOSTNAME:e3a08936-5a92-4d70-96f8-17b5b3f5cb2e, ACTIVE:true, RESPONSETIME:1
 Free Memory: 1516880, availableProcessors: 1
 IP:192.168.56.102, COUNTRY:- (-), CITY:- -, LATTITUDE:0, LONGITUDE:0, TIMEZONE:-
---------------------------
2. HOSTNAME:6b2ea425-4014-47f1-abae-52fe7b9192ae, ACTIVE:true, RESPONSETIME:1
 Free Memory: 110477432, availableProcessors: 4
 IP:127.0.0.1, COUNTRY:- (-), CITY:- -, LATTITUDE:0, LONGITUDE:0, TIMEZONE:-
---------------------------
```

Figure 3.5 : Host list on Host Registry.

3.3.3 Broker-to-Host Ping Pong messages

Broker updates information regarding to hosts periodically and modifies them on Host Registry. Response Time is the key information on this point. Broker sends special PING messages to any host on the Host Registry and expects a special type of message called as PONG message. Host Listener Thread on the host listens for incoming Ping messages from Broker anytime and responds with a Pong message accordingly. The interval between Ping and Pong messages gives the response time of communication between Broker and host.

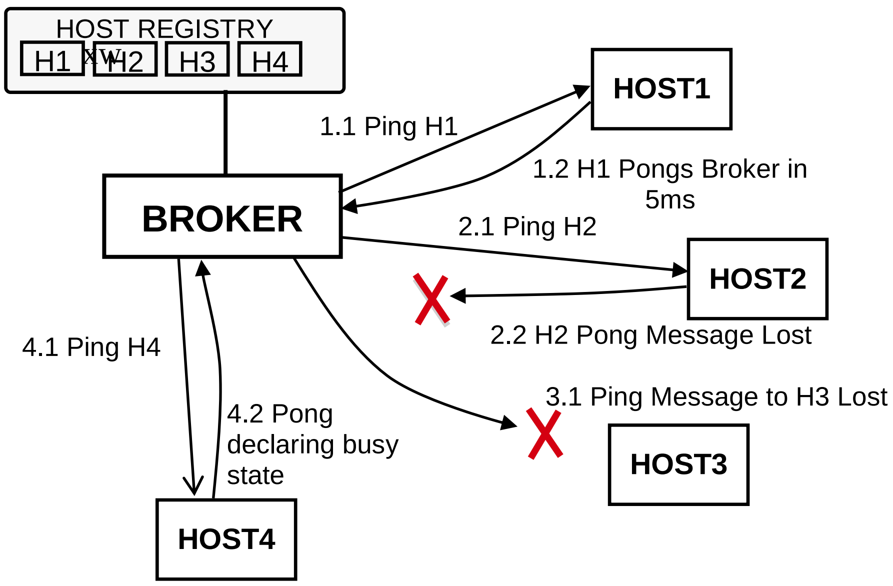


Figure 3.6 : Ping Pong mechanism.

As illustrated on Figure 3.6, Broker sends Ping messages to hosts in order to calculate response times, check the availability and reachability of hosts via sockets connected to host addresses. 4 different possible cases regarding to Ping Pong messaging are being considered. Figure 3.6 shows examples of these 4 cases, on which 4 hosts are registered to Host Registry, each one showing a different case.

1. Step 1.1 and 1.2 shows Broker sending Ping message to Host 1 and getting Pong response from Host 1. Broker Updates response the entry of Host 1 on Host Registry and set the response time to 5ms as on example on 1.2.
2. Broker sends Ping message to Host 2, according to Host 2 entry on Host Registry. Example on Step 2.2 expresses the case when the Pong message not able to be sent to Broker due to communication issues. Broker in this case considers Host 2 as unavailable. In this case, Response Time is set as -1 and active status as false.
3. Step 3.1 shows the case of which Broker is not able to send Ping message to Host 3. In this case, Response Time is set as -1 and active status as false also.
4. Step 4.1 and 4.2 shows the case of which Broker gets the Pong feedback from Host 4, but host Host 4 declares itself on busy state. In this case Broker sets the busy parameter to true for Host Registry entry of Host 4.

As explained above, response status of host entries in Host Registry is updated periodically. The actual status regarding to 4 cases above would be logged as on Figure 3.7.

```
1- HOST1 -- active: true  -- busy: false -- response time: 5 ms
2- HOST2 -- active: false -- busy: false -- response time: -1 ms
3- HOST3 -- active: false -- busy: false -- response time: -1 ms
4- HOST4 -- active: true  -- busy: true  -- response time: 7 ms
```

Figure 3.7 : Host Registry response status.

3.3.4 Client and host state transitions

A Java Applet process created on client computer may behave as client or host depending on its activity status. Client User during its initialization connects Broker and registers itself as host on the Host Registry. By default the client remains in available host state unless the user requests for a Task Execution or serves as execution resource to other requester clients. Check Figure 3.8 for illustrated state transitions of client and host.

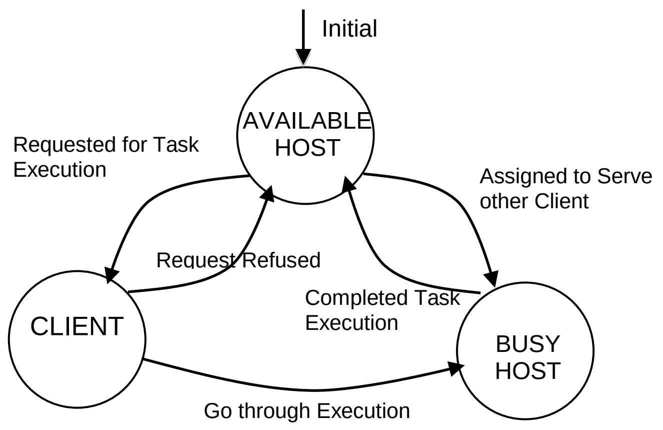


Figure 3.8 : Client and host state transitions.

Initially Java Applet is on available host. It may remain on Available state unless 2 possible cases trigger status change to client or busy host states. In case of user selecting an application for execution the state of available host will be transformed to client state. Another transition occurs when Broker assigns the host with a task for serving some other client in terms of executional power. In this case available host will be transformed to busy host state. Whenever the host completes its task process will return to its original state, available host in this case. Also, client while doing parallel task execution it will be transformed to busy host during the execution and return to original available host state on finalization.

3.4 Task Execution

Task execution involves a set of operations under Task Execution Middleware, in order to complete parallel processing. Initially, users do request for execution of their Applications. Then Broker would respond to requests with a set of assigned resource hosts. The decision for assigning hosts is made according to a scoring mechanism performed by the Broker. As a result, high rated available hosts are provided to requestor clients. In the meantime client will be responsible for initiating the main task, distributing the fragmented data to sub-tasks, sending divided data sets to each task and collecting back after each hosts execution is finalized.

Data messaging between client and hosts are made via peer-to-peer protocol instead of a centralized protocol. On the other hand, idle hosts, which are on the available host state, have their dedicated Listener Threads, which wait for incoming Task assignments. Both client and hosts download and cache packaged JAR application from the Task Repository during execution. Java Reflection API, Remote Class Loading and Object Serialization are the technologies implemented for these purposes. Also, Remote Jar Packages executed under the Context of Java Applet, are cached and executed from the local cache unless the JAR is modified or the Cache is cleared.

3.4.1 Task (application) repository

Task Repository consists entries of uploaded parallel applications. Users, which have developed their application implemented on Parallel Programming Library, submit their JAR packaged applications on ITU-PRP Web application. Any submitted application will be registered to Task Repository and will be available for execution by the client and hosts. User submitting its application should provide parameter information as on Table 3.2.

Table 3.2: Entry on Task Repository.

Information	Explanation	Example
Application ID	A unique key assigned by the broker from a sequence	*37*
Parallel Task Name	Task name defined by the user. This is the identity name of the task on ITU-PRP web application	*Parallel Sums*
Main Task Name	Main task name defined for Reflection API to initialize the Main Task on Client Applet. Main task involves Java implementation of the main method executed by Requestor Client.	*com.itu.ppp.examples.SumsWithFuture*
Callable Task Name	Task name defined for Reflection API to execute the implementation code of sub-task on Host Applet. Callable Task involves the implementation of the work of the parallel task performed on the assigned Host.	*com.itu.ppp.examples.Sum*
Required Host Count	Suggested host count for parallel processing. This value is defined by user, but is not the fixed value, may be defined by Broker on real time.	*4*
Parallel Task JAR URL	The URL Path of the JAR application uploaded by the user. The JAR name is given by the user before the submission, the path is defined by ITU-PRP system.	*http://parallelpattern:8080/AppletParallelProgImplementation/jars/ParallelProgram.jar*
Parallel Task Version	Revision number of the application submitted. The version number should be increased according to versioning standards and defined by the user.	*1.00*
User	Is defined by the System with the username of the Task uploader. User information is defined for security purposes, to set the owner of the Task and restrict the usage for the	*genericuser*

3.4.2 Host resource request

Users requesting for host resources contact Broker Service to get Task Execution Plan for parallel operation. User transforms its state from available host to client during this operation. The set of activities performed during Host Resource Request are shown on the Sequence Diagram illustrated on Figure 3.9.

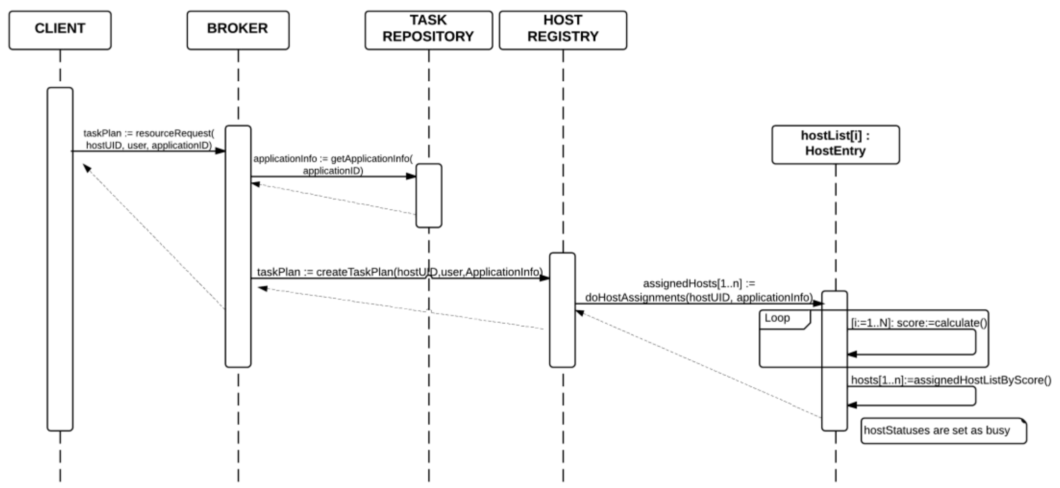

CLIENT
BROKER
TASK REPOSITORY
HOST REGISTRY
hostList[i] : HostEntry
taskPlan := resourceRequest(hostUID, user, applicationID)
applicationInfo := getApplicationInfo(applicationID)
taskPlan := createTaskPlan(hostUID,user,ApplicationInfo)
assignedHosts[1..n] := doHostAssignments(hostUID, applicationInfo)
Loop
[i:=1..N]: score:=calculate()
hosts[1..n]:=assignedHostListByScore()
hostStatuses are set as busy

Figure 3.9 : Activity during host resource request.

At first step, client asks Broker to provide the required Task Execution Plan in order to perform Parallel Processing. On the following steps, Broker will get the information regarded to requested Application for execution from Task Repository. This information is named as *applicationInfo*, which contains parameters like required host count, main task name, callable task name, Task URL and so on. Especially required host count an important information, on which Host Registry assigns a number of assigned hosts resources according to this. Although required host count is predefined, Broker may assign for a different number of hosts depending on the actual available hosts the measured experimental results during recent Task Executions.

Just after application Info retrieval, Broker contacts Host Registry and calls *createTaskPlan* function, in which Host Registry will create a Task plan for the requested Application with the required number of host resources. The combination of the results for both application and host resources is called as Task Plan. Host Registry decides for the assigned hosts from the list of available hosts. Host Registry makes this decision by scoring available hosts and assigning the highest ones. Assigned hosts will become resources to be provided to the client and their states will be set as busy hosts during Parallel Processing lifecycle.

3.4.3 Scoring for host selection

Broker should provide the best possible resource in order to achieve worth parallel performance over sequential performance. This is achieved by a scoring algorithm performed by Host Registry. Information like client's location, host's location, Host

Response Time, Free Memory and number of CPU cores are being considered. The scoring algorithm is cost based, the hosts with lowest costs are being selected.

Geographical information of hosts, which involves Longitude and Latitude are special prioritized parameters during Score calculation. Considering that Parallel Processing is made by peer-to-peer protocol between client and hosts, the assumption that hosts with close location to each other will spend less time on network latencies may be a reasonable consideration. Longitude and Latitude information is effectively used, unless location information is not be provided by IP Location information provider. In case of missing location information, distance consideration will be ignored. On the other hand, Host Response Time, which is the measured time difference between Ping/Pong messages gives idea about response status of the host. Other additional information regarding to host's computational power, which are Free Memory and number of CPU cores are other considerations during scoring operation.

Below pseudo code shows the formula of cost calculation that express that distance between client and hosts are calculated geometrically and the reference point is client's location. So the client should be ideally the center of selected hosts. Higher CPU and Free Memory are the measures, which decrease the cost.

Pseudo Code:

```
function calculateCost (clientLongitude, clientLattitude, hostLattitude,
                           hostLongitude, respTime, freeMem, cpu)
begin

    distance = sqrt( abs(hostLattitude-clientLattitude)^2 + abs(hostLongitude-
clientLongitude)^2 );
    cost = respTime/10 + distance – cpu*(freeMem/1000000);

    return cost;

end
```

The calculated cost values for each host are compared to each other and the lowest ones are picked and provided for Task Plan creation. Additional scoring mechanisms, like Volunteer Reward System and Statistical Performance Records for recent executions are future works to be implemented on the scoring system.

3.4.4 Task plan

Task plan, which is generated as a response for Host Resource Request, is structured from a list of assigned Resource Hosts for the requested task. Assigned hosts are assumed to be on available host status and waiting for client's contact in order to perform their task. Length of resource host entries within Task Plan would be number of Parallel CPU's doing the Task Execution for Parallel Processing.

Table 3.3 gives an example of a Task Plan provided by Broker to client. Client Host Address, Host Name, JAR URL, Callable Task Name information on its disposition. Host Address contains IP Address and port number of resources host, to which client will connect and notify for an execution request. Host name is the information used for Task Execution Report, which will be sent as a performance feedback to Broker. Also, host name is used for logging purposes. On the other hand, JAR URL and Callable Task Name is sent to host to specify which application and function will be executed by the host. Host's Task Listener thread getting those parameters will know that its assigned task is to run *com.itu.ppp.examples.Sum* task on the *http://parallelpattern:8080/AppletParallelProgImplementation/jars/ParallelProgram.jar* application as shown on the example on Table 3.3.

Table 3.3: Task plan example.

Resource Host	Resource Detail
1	**Host Address:** 192.168.56.103:2049 **Host Name:** b4b45ff2-04e3-4af5-b1af-19d62d711807 **JAR URL:** http://parallelpattern:8080/AppletParallelProgImplementation/jars/ParallelProgram.jar **Callable Task Name:** com.itu.ppp.examples.Sum
2	**Host Address:** 192.168.56.101:2049 **Host Name:** c3e37603-45cc-43fb-8e6b-be55e104286e **JAR URL:** http://parallelpattern:8080/AppletParallelProgImplementation/jars/ParallelProgram.jar **Callable Task Name:** com.itu.ppp.examples.Sum
3	**Host Address:** 192.168.56.104:2049 **Host Name:** 6dc711e3-33b5-4a3c-b3cd-73135a0d050a **JAR URL:** http://parallelpattern:8080/AppletParallelProgImplementation/jars/ParallelProgram.jar **Callable Task Name:** com.itu.ppp.examples.Sum
4	**Host Address:** 192.168.56.102:2049 **Host Name:** 2496f352-260e-4799-8790-8eaea4b7109b **JAR URL:** http://parallelpattern:8080/AppletParallelProgImplementation/jars/ParallelProgram.jar **Callable Task Name:** com.itu.ppp.examples.Sum

3.4.5 Parallel application execution

Client Applet that requests for the execution of an Application by user's directive will act according to Task Execution Plan provided by the Broker. Task will be completed without presence of the Broker, client will communicate resource hosts directly via peer-to-peer protocol and will not contact Broker until the end of the Task. By the finalization of the task, client will generate a Task Execution Report as a feedback of Task status and send to Broker. Steps performed during the Task Execution, are illustrated on Figure 3.9, on which Main Task, Sub Tasks, Thread Pool and hosts are the performers of the Task Execution cycle.

Parallel Applications are developed according to implementation pattern of Parallel Programming Library, of which implementation details are described on section 4.2. Developer implements Main Task and Sub Task code blocks, specifies the work to be done on client and hosts. While Main Task is created and executed on the client, Sub Task is created on the client but performed on the host. User would know that Sub Task is executed on the host. Main Task is specified by implementation of *runMainTask* method of *Parallelizable* interface. On the other hand, Sub Task is specified by implementing *calculate* method of *TaskHandler* abstract class. Developer also sets data variables to constructor of the implemented *TaskHandler* object. The mentioned data variables are used as Data set for execution of Sub Task on the host. Main Task Name and Callable Task Name parameters are required to be configured during result application upload to Task Repository. While Main Task Name represents full package presentation of the class to be executed as the Main Task, Callable Task Name is the full package presentation of the class to be executed as Sub Task.

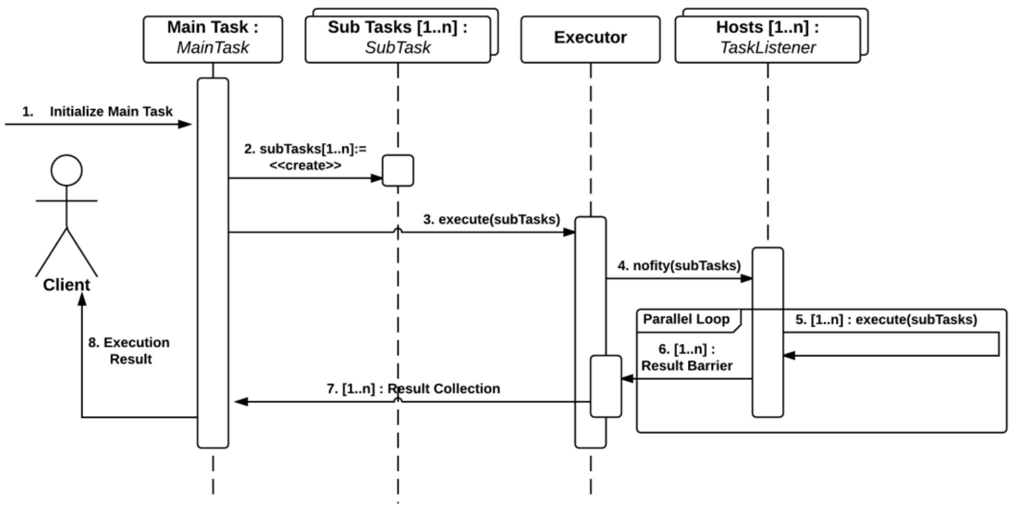

Main Task : MainTask
Sub Tasks [1..n] : SubTask
Executor
Hosts [1..n] : TaskListener
1. Initialize Main Task
2. subTasks[1..n]:= <<create>>
3. execute(subTasks)
4. nofity(subTasks)
Parallel Loop
5. [1..n] : execute(subTasks)
6. [1..n] : Result Barrier
7. [1..n] : Result Collection
8. Execution Result
Client

Figure 3.10 : Parallel task execution steps.

Steps illustrated on Figure 3.10 describe the sequence of operations performed during Parallel Task Execution cycle. More detailed information about these steps are as follows.

1. **Initialization of the Main Task:** By selecting the Application from Task Repository, the user gives the directive for the execution of a Task, which will be initiated on client Applet. The execution mechanism is realized by client's Java Applet, on which application's JAR URL is injected to Applet's Class Loader. Then the object of Main Task is created via Reflection API referencing to the regarded Class on the injected JAR package. By calling *runMainTask* method of created Main Task object, user's Application is initiated and started for execution. A case example of *runMainTask* is shown on Figure 4.8 on Implementation section.

2. **Create a Sub Task Object for each available host:** Developer specifies the work of host by implementing *SubTask* interface and its *calculate* method. *Calculate* method (Figure 4.9) of the implementation should be filled by parallel developer in order to set *result* field on termination. Sub Task's data may be set even by constructor or explicitly depending on requirement. User creates a list of Sub Tasks within the scope of *runMainTask* method as shown on Figure 4.8. User may also set even same data to all sub tasks, which makes shared memory model applicable.

Main loop should have its number of iterations equal to available numbers of resource hosts which will ensure one Sub Task per host as the ideal condition.

3. **Submit Sub Tasks to Task Executor:** User submits *SubTask* list by calling *execute* method of *TaskExecutor* component within the scope of *runMainTask* in order to submit Sub Tasks to be processed in parallel. Executor Service and Thread Pool mechanism of Java Concurrent API is utilized for Thread based operations within *Task Executor* component. In default, Parallel Programming Framework sets Thread Pool size to number of available resource hosts and it should be ideally equal to number of Sub Tasks, which will create parallel threads per host. However, the framework may set Thread Pool size to another value depending on execution type for these cases: Sequential execution; Lack of sufficient resource hosts; Speculative Execution Type. Due to this reason, user should set a proper required host count value to Task Repository entry of the application. In case of available host count being lower than Sub Task count, at least one Sub Task will be processed sequentially. As example, the case when the there are 8 sub tasks, but number of available resource hosts are 4, means that 8 Sub Tasks will be sent to 4 hosts in 2 occurrences, so each host will operate twice. The case of available host count being higher than Sub Tasks, would make some of assigned hosts doing no operation, remaining idle.

4. **Notify each host for task processing by sending the Sub Task to each one:** By the submission of a Sub Task to executor service, notification service makes a connection check to resource host's *TaskListener* Thread via UDP socket connection to the regarded host's IP Address and Port. Connection check is important in order to prevent connection faults. In case of failed connection check, Task Executor sends the sub task to a backup resource host, performs sub task locally or notifies client as failed for failed execution. If the connection check succeeds, notification service will send Application's JAR URL, Callable Task Name of the Sub Task and Serialized Object Stream of the Sub Task to resource host. Sending object stream of the Sub Task instead of parameter specific messaging ensures maintaining application state in both ends.

5. **Sub Task execution on each host:** *TaskListener* Thread on the available host has an active UDP socket waiting continuously for incoming Task assignments. In case when *Task Listener* gets an incoming Task Message from a client, where a JAR URL and a Callable Task Name parameter that will specify the Application and the regarded Sub Task to be executed. The execution mechanism is realized by Java Applet, application’s JAR URL is injected to the Class Loader like on step 1. But differently from step 1, host’s Applet will get the incoming Sub Task Object stream and get the Sub Task with the state and data, which was created on step 2. *Task Listener* executes its task by calling *calculate* method of the Sub Task. By the finalization of processing, *Task Listener* Thread will respond back to client with the Serialized Object Stream of the Sub Task. Note that response Sub Task will have its result set on the *result* field.
6. **Wait until parallel processing is completed on each notified host (Result Barrier):** Host Task Notify Service, which has sent the Sub Task to the host, this time will wait for the response from host, modified Sub Task object with its result field set. In order to finalize Parallel Processing, Executor Service should ensure all Sub Tasks to be processed. A barrier mechanism provided by the Executor Service expects all the results back from the hosts. *Shutdown* function of the executor service releases the Barrier and makes the Main Task to continue through its normal flow. Depending on the design of the developer, barrier release operation may be issued without getting all results also. If developer implements speculative parallelism where same data and same Sub Task is sent to all hosts, developer rely onto a single result from the Host which has processed the Task faster than others. In this case Executor service will release the barrier as it receives the first processing result.
7. **Result collection:** Developer may decide how to manipulate with Sub Task results collected from each Sub Task’s Handlers. Final result may be collected by an aggregate operation (Sum, Average, etc) from result sets or may be manipulated depending on its design. Perhaps any comparison or a matching operation would be a result of the parallel processing. Result collection is flexible in terms of result data manipulation. For instance,

example on Figure 4.8 shows a case where the final result is the sum of collected result sets.

8. **Provide execution result to client:** As final step, Main Task will return the final Result to client Applet, on which the result will be presented to the user. Also the result is included to Task Execution Report, which is sent to Broker after the finalization of Task Execution. Execution Reports are the basis of execution details and results to be viewed in ITU-PRP Web Application on a later time. While user is notified for the final result, Broker is notified with an execution report in order to transform hosts to available state.

3.5 Client, Host and Broker Connectivity

Any host that connects Broker is required to maintain an active connection during its lifecycle. Broker keeps sessions of hosts on Host Registry and updates tracked information on any activity. Due to the nature of the session based design, active connection should be mandatory. A TCP Socket connection with Broker is maintained during initialization of the host and remains active until host terminates its session. On normal conditions, a TCP Socket terminates the connection and remains idle. Stability is ensured by sending Keep-Alive Sockets [12] and Ping messages periodically. The frequency for sending Ping messages to hosts is 20 seconds, which is beyond the timeout of a TCP socket.

Client and hosts, assigned for Task Execution, would communicate each other in order to exchange data parameters required for task execution and result retrieval. Connectivity and delay time during transmission of data between hosts is the key point and delays should be minimal in order to achieve efficient parallel execution.

Achieving optimal delay is very challenging due to heterogeneity of the system. Designed protocol for communication between client and hosts, data sizes sent to hosts affect directly on delay times. Client which request for Task Execution gets a response with ID and addresses of available hosts for execution. This phase of delay is followed with distribution of data parameters to hosts, which is a bigger deal compared to task plan retrieval request.

3.5.1 Approaches on communication flow

Assume communication protocol is designed according Approach 1.

Approach 1: According to this approach, client requests for Task Execution Plan from Broker and gets the list of available hosts with its addresses. On the phase of task distribution to hosts, client will send required parameter data to Broker then Broker will forward data to the regarded hosts. After independent execution and result generation on hosts, hosts will be sending the calculated results to Broker. Then Broker will send the results back to client. According to this centralized protocol, client and hosts are dependent to a centralized Broker service, which will coordinate data forwarding. Check Figure 3.11, for representative schema of this approach. There would be totally 6 hops of communication within task execution cycle, 2 hops for Task Plan retrieval (1, 2), 4 hops for sending and receiving data between client and hosts (3, 4, 5, 6). Also, depending on the intensity on Broker Service and the data size of host parameters, 4 hops of data exchange phase may have additional latencies.

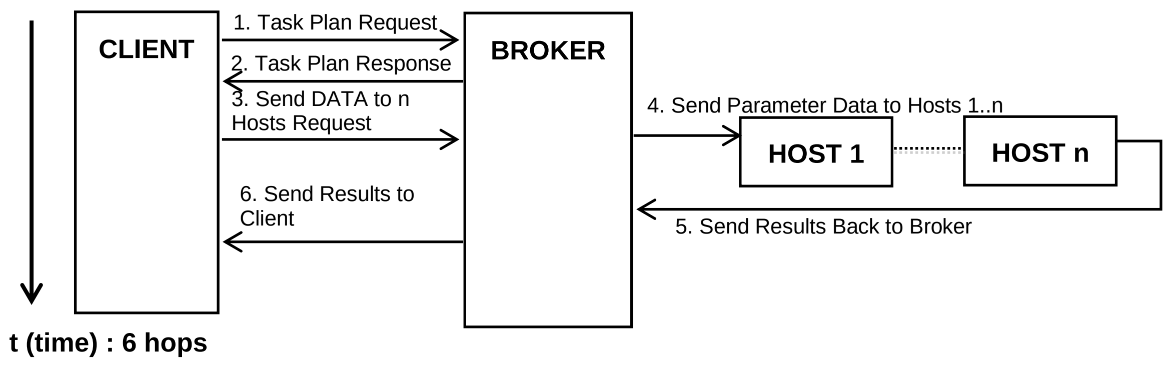

CLIENT
BROKER
1. Task Plan Request
2. Task Plan Response
3. Send DATA to n Hosts Request
4. Send Parameter Data to Hosts 1..n
HOST 1
HOST n
5. Send Results Back to Broker
6. Send Results to Client
t (time) : 6 hops

Figure 3.11 : Approach 1 suggestion for communication protocol.

Protocol on the first approach can be optimized in terms of hop count for sending Data parameters to hosts. A slight reduction may be if client is able to communication hosts directly instead of data distribution through presence of Broker. Assume suggestion in approach 2.

Approach 2: Client requests for Task Execution Plan from Broker and gets the list of available hosts with its accessible addresses. On the phase of task distribution to hosts, client will attempt to establish connections to hosts directly. In case of successful connection, client will send data and get result data after execution on hosts. This method of communication will work without presence of broker during data exchange, in decentralized manner. Check Figure 3.12 for representative schema of this approach. There would be totally 4 hops, 2 hops for Task Plan retrieval (1, 2), 2 hops for sending and receiving data between client and hosts (3, 4). Compared with first approach, there would be no additional latencies caused by intensity on Broker will be reduced. Broker will remain responsible for Task Plan assignment in dedication.

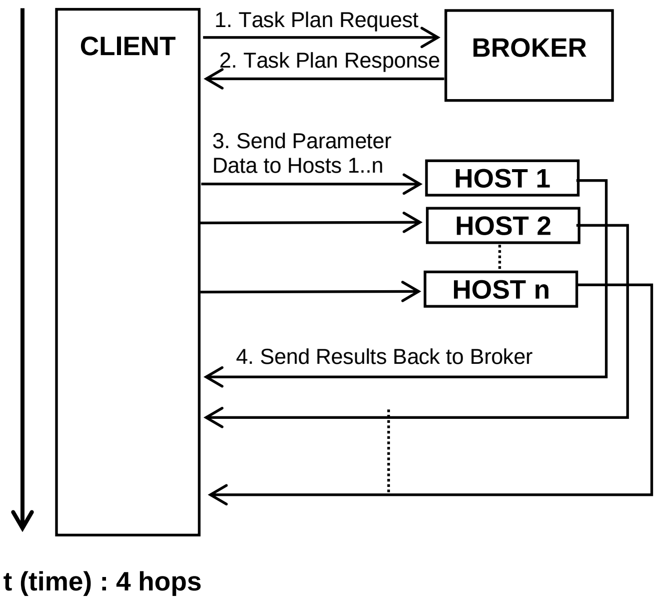


Figure 3.12 : Approach 2 suggestion for communication protocol.

In conclusion, protocol in second approach is much more optimized then the design in first approach. Clients and hosts will communicate through peer-to-peer protocol. Network latencies related to centralized Server bandwidth usage, will be reduced. High throughput on Server processing power will be prevented.

3.5.2 NAT issues of peer-to-peer communication

Peer-to-peer communication between hosts and clients has taken a remarkable part within this research. Peer-to-peer communication is difficult, due to Network Address Translation (NAT) and global accessibility between nodes. Two nodes within a local network would access each other’s IP Address and ports. On the other hand, NATs assign public IP addresses to packets during Internet access. Host’s IP address and port is changed during intra network connections and NAT creates address/port translations maps on Gateway Routers. Figure 3.13 would illustrate this case, on which a client with 192.168.1.10 local IP Address and Port will be translated to a public IP address and the port of the session when reaching another node on a different network. NAT Protocol on the router assigns a global 212.252.140.70 IP Address and another port and adds IP/Port pairs to port mapping. Depending on NAT type some routers may preserve local port number and not change the source port of the IP packet. A connection made to a Server (5.101.107.102) has different IP and Port information on both sides. While client knows its address and port number as 192.168.1.10:31000, Server knows client's address/port pairs as 212.252.140.70:2571.

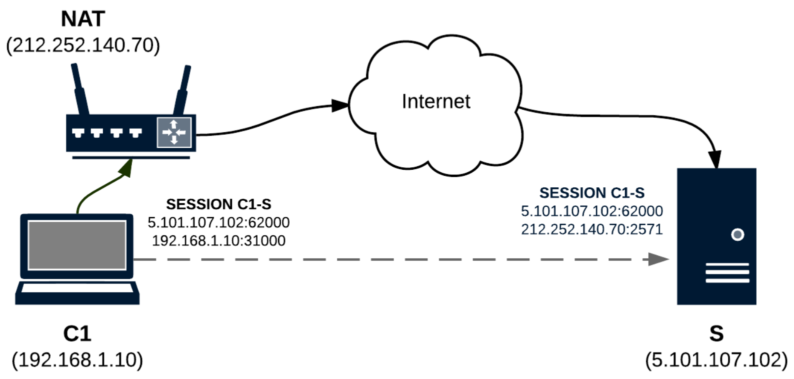


Figure 3.13 : NAT private to public IP translation.

Considering the nature of NAT, two hosts on different networks should know each others global IP address and port number in order to maintain a direct connection. A central server that collects each host's public IP address and ports may perform this task and would coordinate hosts on different networks to connect each other. In fact, such a server may be called "Rendezvous Server". Broker would be the most likely entity within ITU-PRP to perform rendezvous role. Considering that Broker takes part in activities, like accepting initial connections, retrieving Host IP addresses and ports, monitors hosts' connectivity status and processes requests from clients, it obviously would have all required information in order interconnect hosts. In that case, connection attempt between two hosts would be as illustrated as in Figure 3.14. The regarded figure illustrates a case where H1 which is one of 2 registered hosts on Broker will try to connect another H2 host registered to Broker as well. The first assumption is that Broker has collected global ip address and port information of each hosts during initialization. Second assumption is that H1 requests Broker for a host resource for task processing during a time of its lifecycle. Then Broker would respond H1 with a task plan where H2 is the host to connect for task processing. H1 tries to connect H2 through given global 91.42.131.123 ip and 14422 port number. Due to the fact that H2 is in another network behind NAT2 and 91.42.131.123 is global IP address of NAT2 router, the packet will be send to NAT2 router, on which NAT2 will check the source address and port on its port mapping table in order to decide to which endpoint should incomming packet routed. In fact, NAT2 has the information that a previous outgoing message of H2 (10.1.1.2:60493) has been mapped to 14422 port number. So, NAT2 would make the incomming message to be sent to 10.1.1.2. However, most routers perform strict security checks and ensure that incomming packets source ip is available on port mapping table. In our case, the recent port mapping which has set 92.42.131.123:14422 pairs for 10.1.1.2:60493 endpoint was set for Broker's (5.101.107.102) output traffic and only incomming traffic from Broker would be recognized by NAT 2. So, H1's connection attempt to H2 mostly fails, except for some cases: NATs which behave with non-strict rules; hosts which are assigned with a global IP by ISP.

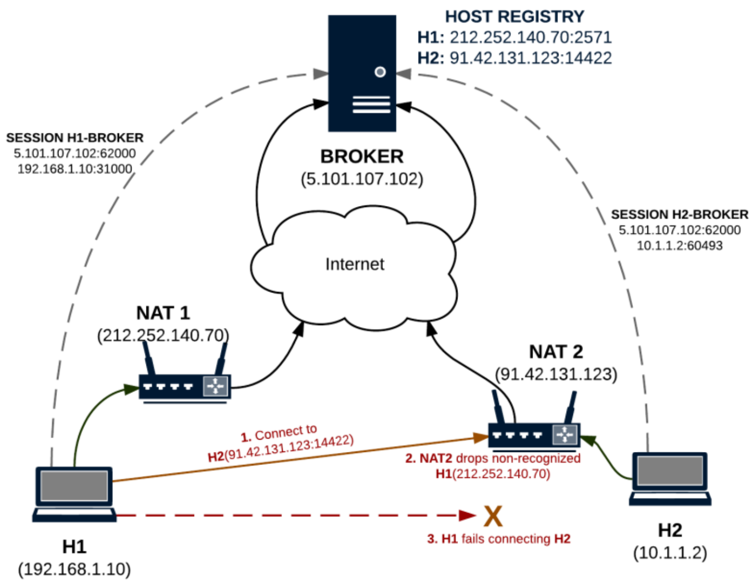


Figure 3.14 : Peer-to-peer connection failure of hosts behind different NATs.

3.5.3 NAT Traversal techniques for peer-to-peer connection

Methods used to overcome NAT difficulties are specified as NAT Traversal Techniques [12]. Relaying, Connection Reversal, Hole Punching are analyzed NAT Traversal Techniques on during literature research.

Relaying is the technique where server handles traffic between peers. Packets are sent to server, then server forwards packets to destination peer. In fact, this technique is the most reliable one and NAT issues do not affect connectivity. However delay times are higher, since messages are sent via server.

Connection Reversal is the technique where at least one of two peers is not behind a NAT and has a public IP. Assuming that a peer is behind NAT is named as P1 and tries to connect another peer with a public IP named P2, the connection would be successful. However, the first connection should be initiated from P1 to P2. If P2 tries to connect P1 initially, P1's NAT will drop the connection and it will fail.

UDP Hole Punching technique enables two peers to connect each other directly, even they are behind NATs. A rendezvous server that collects endpoint information of peers, coordinates peers to connect with each other. The key characteristic of this tehcnique is the fact that rendezvous server collects both public and private ip/port pairs. Peers send their private ip/port pairs to rendezvous server during initialization phase, also server gets the public ip/port of the peers. The reason of collecting private ip/ports is establishing connections of endpoints within same NATs. Peers behind same NAT would be able to connect each other directly, since they are on the same network. On the other hand, server keeps public ip/port of endpoints to make peers behind different NATs connecting each other. The working principle of UDP Hole Punching is a bit tricky which forms a flexible architecture that makes possible internal and global connectivity. Figure 3.15 taken from [12] illustrates a case, on which Client A and Client B create sessions to server. If Client A is supposed to connect Client B, the rendez vous server sends notification to both Client A and Client B on the same time. The notification message send to Client A consists public and private ip/port of Client B. A will send 2 UDP packet to B, one packet to B's public address, one to B's private address. On the other side, Client B gets peer information of Client A and performs the same process. It is supposed that both clients will send UDP packets to each other within a common time slot. Let's assume that first message is sent by Client A. A's UDP packet to B's private address never reaches, because Client B is behind another NAT. On the other hand, Client A's message to Client B's public ip reaches to B's NAT, but the router drops the UDP packet, because the because the source IP of Client A is not known on the port mappint table on NAT B. So Client A's first attempt fales. Anyway, even Client A has failed sending UDP packet to B, A's attempt will create a Hole on its NAT. As described on above quotes, a NAT accepts only incomming messages which are coming from an IP and Port of which a recent message was sent. So, in our case Client B's incomming UDP packet which is sent from the opposite side, is coming from an known known source IP and Port, in fact the first packet of Client A was a fake message to create a hole and make its NAT to treat the incomming B's message as the response of outgoing A's message. Although, the principle makes sense, the mehtod does not guarantee success on real world. While some routers will accept hole punching, some other routers have strict NAT rules especially on port mappings. Such routers do not preserve outgoing port numbers and alter them on every sent message. Such behavior of routers make UDP Hole Punching non-applicable.

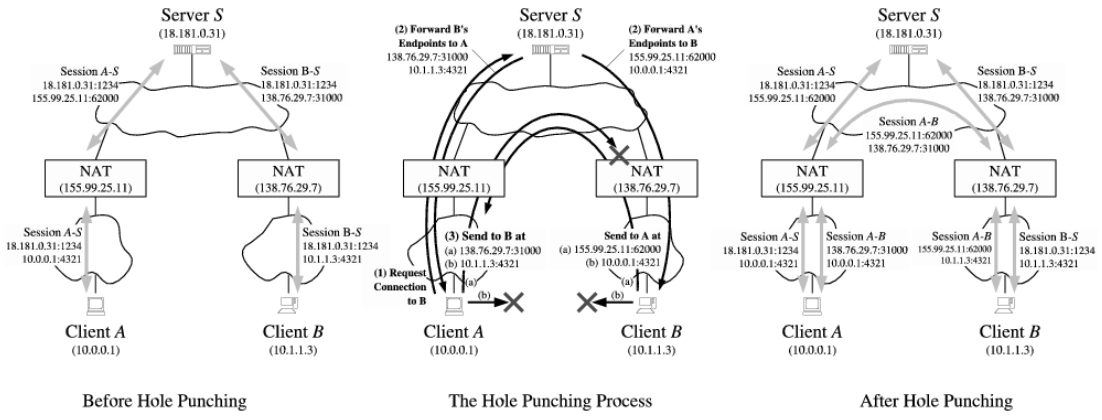


Figure 3.15 : UDP Hole Punching technique [12].

3.5.4 Peer-to-peer Protocol designed for ITU-PRP

Peer-to-peer connectivity is one of the key motivations of ITU-PRP. A widely used and more effective system would be possible only with a perfect peer-to-peer protocol design. ITU-PRP's peer-to-peer protocol combines Relaying, Connection Reversal and UDP Hole Punching techniques in order to cover different types of connectivity challenges. UDP Hole Punching characteristics are adopted to ITU-PRP for connecting hosts within same NAT or behind different NATs if applicable. Also, charasteristics of Connection Reversal are adopted and performs well if one of hosts has public accessibility. Additionally, a relaying mechanism is performed for the cases where Hole Punching and Connection Reversal is not applicable. For all this three techniques, ITU-PRP's Broker takes the role of Rendez Vous Server, hosts and clients fit to the role of peers. The characteristics of the combined design are as follows.

1. **Peer registration:** As mentioned on previous units, hosts subscribe to the system during their initialization. host connects Broker via TCP Socket and UDP socket connections. Host-to-Broker session is kept active by using TCP Socket connection. On the other hand, UDP Hole Punching is applicable with UDP Sockets, because it makes possible connecting ends with the same socket.
2. **Peer information update:** Hosts, send their private IP address and port number to Broker each time they get a ping message. Host's public IP and ports are retrieved by Broker, along with incoming port update message. Port update messages are UDP based. Host binds a UDP socket locally and waits for incoming connections from other peers. On the contrary, TCP socket would not be applicable, because active TCP session would not allow the same port to accept connections from other ends.
3. **Relay pipes:** Rendez vous Server, in fact Broker creates pipes on Server for peer pairs, which will perform relaying. A pipe performs the task of message transmission through Broker. Creation of pipes is made just before notification of peers to connect each other. A logical pipe is created with two physical UDP sockets, of which one will listen for incoming relay messages and server will send to destination peer through other UDP socket.
4. **Peer notification:** All hosts are expected to connect each other in order to form a network of hosts. In order to achieve this, Broker performs Rendez Vous operation during subscription o a new host. Broker sends the list of all hosts to new registered host. Also, peer information of new registered host would be sent to all other recently registered hosts as well. Peer grouping would be implementable in case of increased number of users. Peer group principal would revise the load on Server and also make more specific groups to perform specific tasks. However Peer grouping was kept out of the scope.
5. **Perform punching:** A peer notification message contains three different ip address and port pairs for destination peer, Private Peer address/port; Public Peer address/port; Relay Address/Port. Notified host will initiate 3 messages (private, public, relay) and will keep sending until connection establishment. Figure 3.16 illustrates the set of steps.

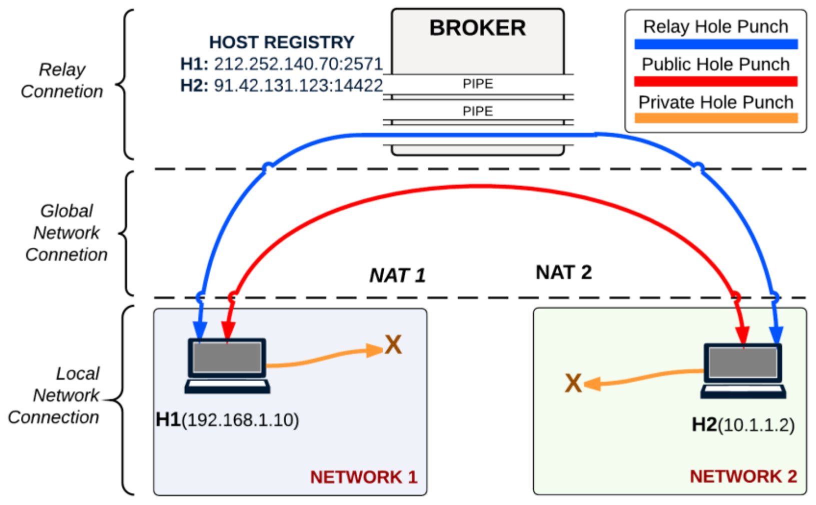


Figure 3.16 : Peer-to-peer connection steps.

6. **Connection establishment:** Hosts which get hole punch messages from its assigned pairs will send acknowledge messages back and will expect acknowledge from the same pair. As soon as two-way handshake is completed, hosts stop sending punch messages. Within 3 punching attempt types, host may connect to private or public or relay address/ports. However, one of established connections will be selected according to prioritization, 1) private; 2) public; 3) relay ip/port connection. The reason of private connectivity being first option lower delay within private networks. For the hosts behind different networks public connectivity is preferred, relay connection would be the last option due to increased delay of centralized connection.

7. **Beacon messages:** Hosts, which have established connection with each other, will send beacon messages during their lifecycle. Beacons would be small UDP packets sent periodically, for keeping inter-host communications active. Beacon exchange will keep host’s peer-to-peer connection state information up to date. Also connection checks during task distribution and executions are performed by beacon exchange, in order to reduce faults during Task Execution.

Peer-to-peer design of ITU-PRP intends creating a network of all registered hosts to connect each other, as showed in Figure 3.17. However heterogenity and conditions of the real World would make any nodes not missing connection with each other. For such cases, Task Execution process performs connection checks and fault prevention steps.

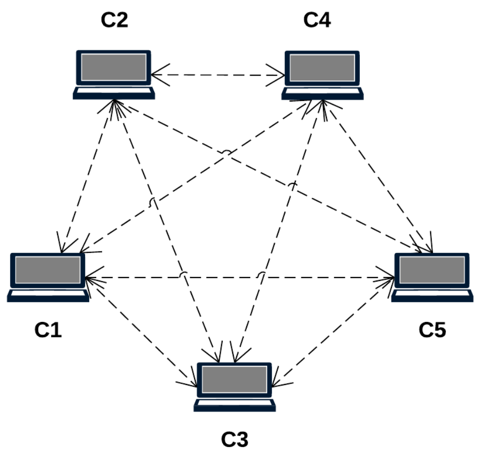


Figure 3.17 : ITU-PRP peer-to-peer network.

3.6 Data Transmission

Data transmission is an important matter which affects delays and performance of Task Executions. Established peer-to-peer connections exchange a wide range of message types with each other, including task notification, beacon or hole punch messages. Also hosts exchange messages with Broker, such as host registration, Peer Information Update, Ping/Pong Messages, etc. A brief schema showing all message time exchanged within system are illustrated on figure 3.18.

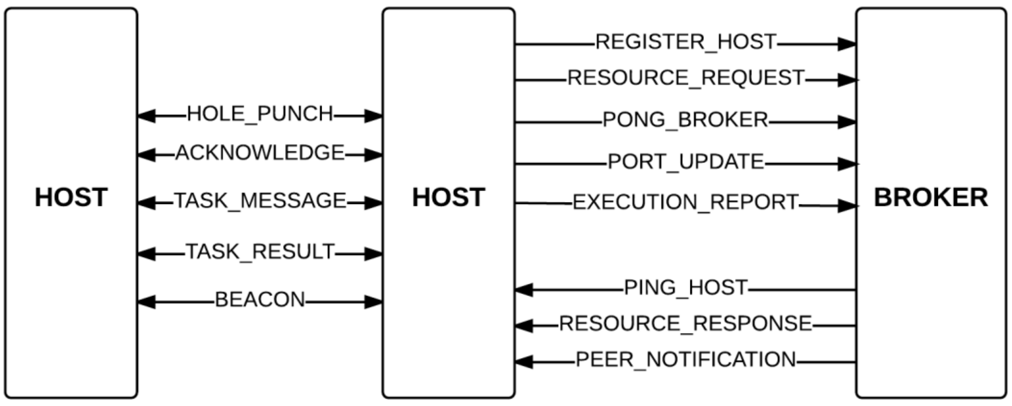


Figure 3.18 : ITU-PRP Peer-to-peer network.

In order to cover all message types a generic messaging type has been implemented. This aims performing a common handling for messaging. Messages contain message header and data sections. Message header contains information about message type, sender Host ID, data length, buffer count, buffer index and message sequence. While header section is 176 Byes long, data section is 400 bytes long, which makes a total of 576 Bytes during message transmission. However data transmitted to another host may exceed 400 Bytes. In this case, data will be devided to multiple buffers. In Figure 3.19 a case for 520 Bytes is given. Buffer count would be 2 and two separate messages will be sent, with 1 and 2 buffer indexes. This principal is same with the Sliding windows protocol of Transport Layer.

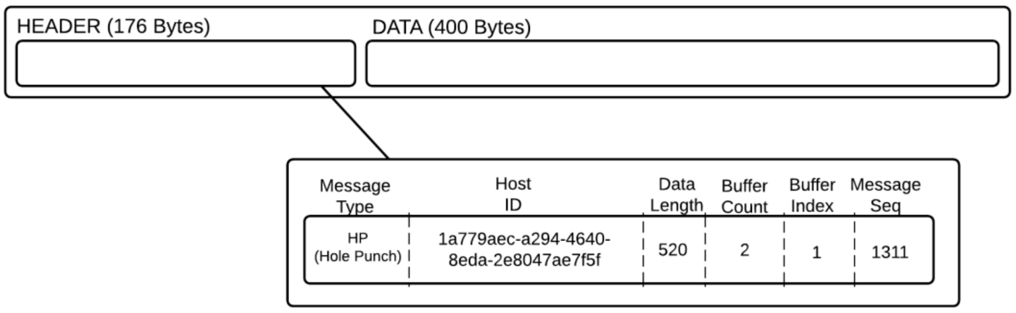


Figure 3.19 : ITU-PRP generic message.

4. ITU-PRP FRAMEWORK IMPLEMENTATION

ITU-PRP's framework has two entities in terms of implementation. While the web application is where user does its operations on his account, Parallel Programming Library is the implementation where the framework for parallel programming has been made possible.

4.1 ITU-PRP Web Application

ITU PRP operations like user subscription, client logon, application upload and modification on repository, task execution are made through a Web application designed in the system. ITU PRP Web application is hosted on an Apache Tomcat Web Server located on the same location with Broker and P2P Server. User logs on to a web based application as showed on screenshot on Figure 4.1.

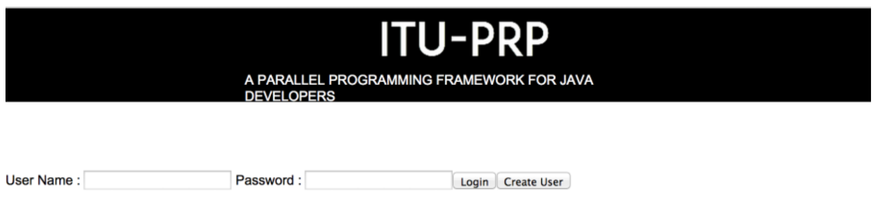
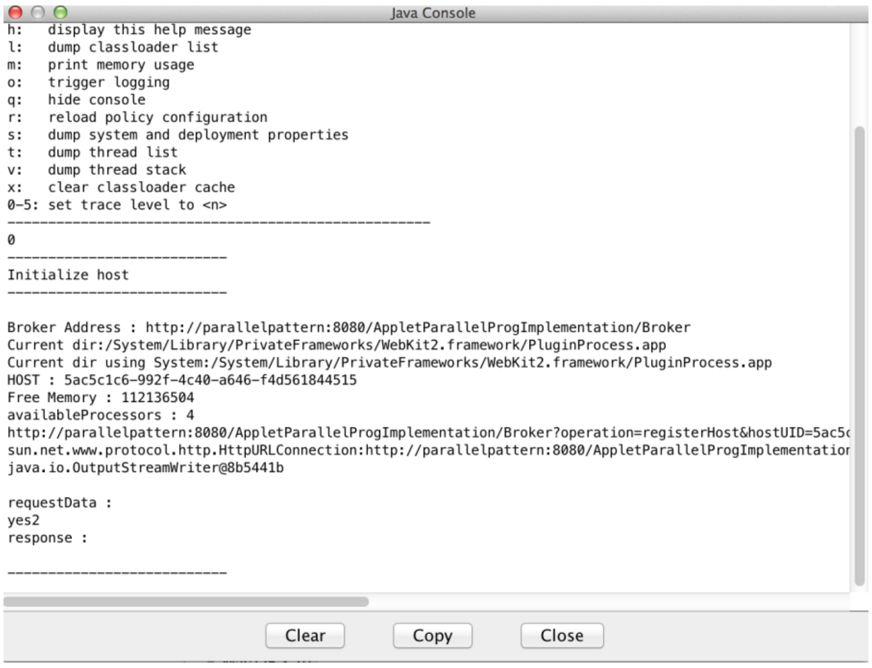


Figure 4.1 : ITU PRP Web application logon screen.

User logs on or signs up on the initial page of ITU PRP Framework. As the user logs on to Web Application, a Java Applet embedded to the web page will initialize and run. The regarded Java Applet creates a host listener thread, which will process as an available host for the system on client's computer. Check Figure 4.2 for Java Applet Console, which shows the activity client/host.

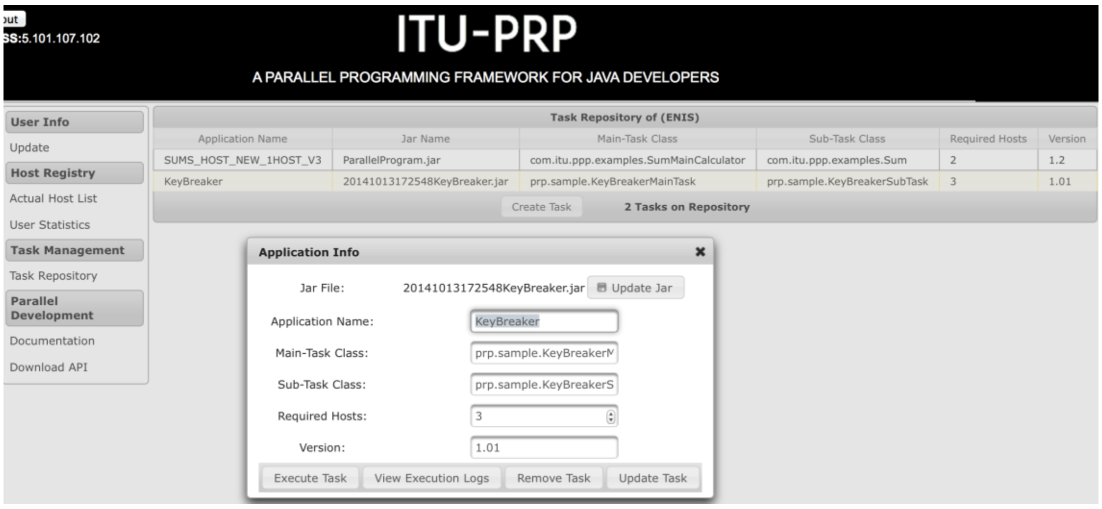


```
h:   display this help message
l:   dump classloader list
m:   print memory usage
o:   trigger logging
q:   hide console
r:   reload policy configuration
s:   dump system and deployment properties
t:   dump thread list
v:   dump thread stack
x:   clear classloader cache
0-5: set trace level to <n>
----------------------------------------------------
0
---------------------------
Initialize host
---------------------------

Broker Address : http://parallelpattern:8080/AppletParallelProgImplementation/Broker
Current dir:/System/Library/PrivateFrameworks/WebKit2.framework/PluginProcess.app
Current dir using System:/System/Library/PrivateFrameworks/WebKit2.framework/PluginProcess.app
HOST : 5ac5c1c6-992f-4c40-a646-f4d561844515
Free Memory : 112136504
availableProcessors : 4
http://parallelpattern:8080/AppletParallelProgImplementation/Broker?operation=registerHost&hostUID=5ac5
sun.net.www.protocol.http.HttpURLConnection:http://parallelpattern:8080/AppletParallelProgImplementatio
java.io.OutputStreamWriter@8b5441b

requestData :
yes2
response :

---------------------------
```

Figure 4.2 : Java Applet on ITU PRP Web application.

User is able to view and select task on its repository with uploaded applications. Application submission to system is made by the user, by filling Application information such as Application/Task name, Java class name with package hierarchy, suggested host count for execution, application version which will be considered base information for Task Execution. User may also select its application from repository and modify its information.

In case of selection of the task from the list of repository, the user will see a screen as shown on Figure 4.3. Task information filled during task creation will be displayed the user. User is able to trigger the execution of the task Sequentially on its own host or in Parallel according to Task Plan created by Broker.



Task Repository of (ENIS)

Application Name	Jar Name	Main-Task Class	Sub-Task Class	Required Hosts	Version
SUMS_HOST_NEW_1HOST_V3	ParallelProgram.jar	com.itu.ppp.examples.SumMainCalculator	com.itu.ppp.examples.Sum	2	1.2
KeyBreaker	20141013172548KeyBreaker.jar	prp.sample.KeyBreakerMainTask	prp.sample.KeyBreakerSubTask	3	1.01

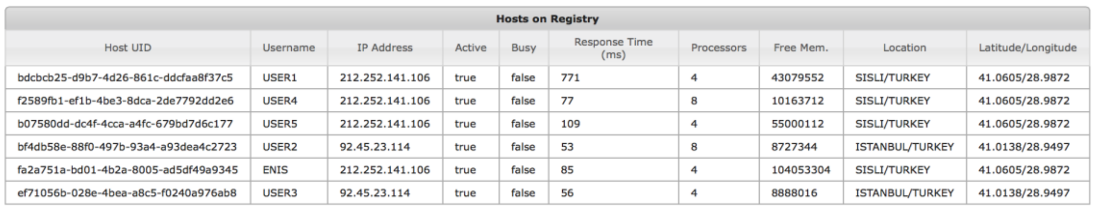
2 Tasks on Repository

Application Info

Jar File:	20141013172548KeyBreaker.jar
Application Name:	KeyBreaker
Main-Task Class:	prp.sample.KeyBreakerM
Sub-Task Class:	prp.sample.KeyBreakerS
Required Hosts:	3
Version:	1.01

Figure 4.3 : Task execution in ITU PRP Web application.

In addition to task operations, user is able to view actual hosts connected to the system as shown on Figure 4.4. Detailed information such as Host ID, user name, Client IP Address, activity and busy status, response times, location information and additional information regarding system resources are available for user's inspection.

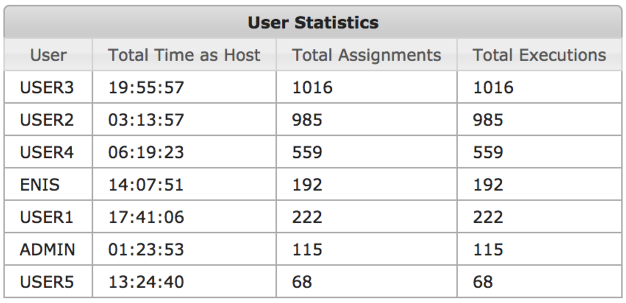


Hosts on Registry

Host UID	Username	IP Address	Active	Busy	Response Time (ms)	Processors	Free Mem.	Location	Latitude/Longitude
bdcbcb25-d9b7-4d26-861c-ddcfaa8f37c5	USER1	212.252.141.106	true	false	771	4	43079552	SISLI/TURKEY	41.0605/28.9872
f2589fb1-ef1b-4be3-8dca-2de7792dd2e6	USER4	212.252.141.106	true	false	77	8	10163712	SISLI/TURKEY	41.0605/28.9872
b07580dd-dc4f-4cca-a4fc-679bd7d6c177	USER5	212.252.141.106	true	false	109	4	55000112	SISLI/TURKEY	41.0605/28.9872
bf4db58e-88f0-497b-93a4-a93dea4c2723	USER2	92.45.23.114	true	false	53	8	8727344	ISTANBUL/TURKEY	41.0138/28.9497
fa2a751a-bd01-4b2a-8005-ad5df49a9345	ENIS	212.252.141.106	true	false	85	4	104053304	SISLI/TURKEY	41.0605/28.9872
ef71056b-028e-4bea-a8c5-f0240a976ab8	USER3	92.45.23.114	true	false	56	4	8888016	ISTANBUL/TURKEY	41.0138/28.9497

Figure 4.4 : Actual hosts connected to system.

System, also provides viewable information regarding statistical information for recent activities of users as showed on Figure 4.5. User Based classified data shows total time of activity and assignment/execution counts in total.

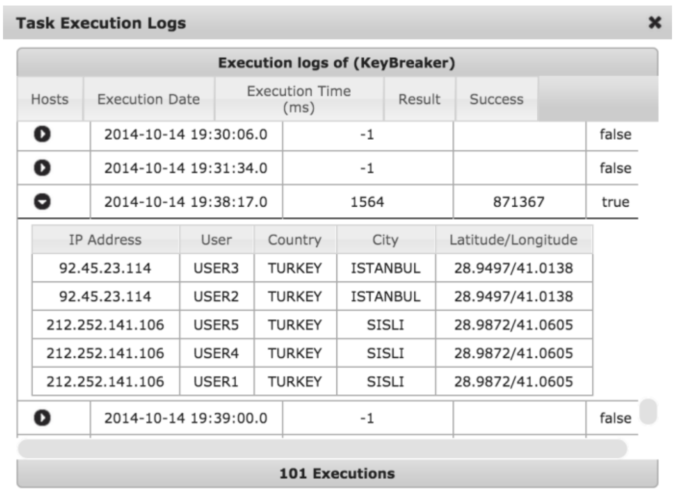


User Statistics

User	Total Time as Host	Total Assignments	Total Executions
USER3	19:55:57	1016	1016
USER2	03:13:57	985	985
USER4	06:19:23	559	559
ENIS	14:07:51	192	192
USER1	17:41:06	222	222
ADMIN	01:23:53	115	115
USER5	13:24:40	68	68

Figure 4.5 : User statistics.

Authenticated user is able to view recent executions and task plans of host executions with detailed information. Check Figure 4.6 for the screenshot of task execution logs.



Task Execution Logs

Execution logs of (KeyBreaker)

Hosts	Execution Date	Execution Time (ms)	Result	Success
▶	2014-10-14 19:30:06.0	-1		false
▶	2014-10-14 19:31:34.0	-1		false
▼	2014-10-14 19:38:17.0	1564	871367	true

IP Address	User	Country	City	Latitude/Longitude
92.45.23.114	USER3	TURKEY	ISTANBUL	28.9497/41.0138
92.45.23.114	USER2	TURKEY	ISTANBUL	28.9497/41.0138
212.252.141.106	USER5	TURKEY	SISLI	28.9872/41.0605
212.252.141.106	USER4	TURKEY	SISLI	28.9872/41.0605
212.252.141.106	USER1	TURKEY	SISLI	28.9872/41.0605

Hosts	Execution Date	Execution Time (ms)	Result	Success
▶	2014-10-14 19:39:00.0	-1		false

101 Executions

Figure 4.6 : Task execution logs

4.2 Parallel Programming Library

A Java library is provided to Parallel Developers to adapt their application codes for Parallel Running Platform. *ParallelPatternFramework.jar* can be downloaded from ITU PRP Web Site. A parallel developer must include the provided library file to Java project and implement its program code according to specifications. Result implementation will be uploaded to ITU PRP Web Site to application repository. Result application should be packaged as Jar file as well. Application developer is required to fill Application/Task name, Java class name with package hierarchy, suggested host count for execution, application version. Library also does consist task distribution, barrier and result collection functions, which are transparent to developer.

4.2.1 Content of ParallelPatternFramework.jar library file

- **MainTask :** Interface which specifies the main task to be execution on initial execution of main task. Developer overrides *runMainTask* method on which routines of main task are implemented.
- **SubTask<GenericResult> :** Interface is implemented in order to specify operation of sub tasks. *GenericResult* generic type is defined in order to make declaration of result type of sub task. *GenericResult* is applicable for Java primitive object types, such as String, Integer, Long, Double, byte arrays, etc. Overrided *calculate* method would consist the operations to be performed on

sub task. On the other hand, *getResult* method implementation would return the result of execution.

- **TaskExecutor :** Consists task distribution and result collection routines which are performed during task execution. Those operations are hided by user and the library takes care of this operations. The only visible *execute(TaskList,ExecutionType)* method is called on MainTask in order to trigger task operations. *TaskList* and *ExecutionType* are two parameters set by user in order to specify the execution work to be performed and the way how they are performed.
- **TaskHandler :** Consists routines performed during sub task distribution. *TaskHandler* converts sub task objects to byte arrays and initializes transmission streams of them to established host connections. Results of transmitted object streams are expected to be received when remote hosts terminate and send back result object streams. Task handler will unmarshal result bytes, extract *SubTask* result object and send back to caller. Task send/receive operations are made synchronously.
- **TaskList :** Holds the list of sub tasks. *TaskList* is prepared by user within the scope of *MainTask* and is send to *TaskExecutor* for execution.
- **ExecutionType :** Enumeration on which task execution type is defined. Parallel application developer decides one of five execution types for task processing. *PARALLEL_MULTIHOST_STRICT* is set for the cases where developer sets the rule of subtasks to be executed only parallel on multiple hosts. This execution type fails in case of any host fails processing the sub task. *PARALLEL_MULTIHOST_ADAPTIVE* is also execution type processed on multiple hosts with a more flexible execution plan in cases of any host fails. If any host fails on subtask execution, main task executes the task on client CPU. *SPECULATIVELY_PARALLEL_MULTIHOST* is defined for a single host to be processed on multiple hosts. First incoming execution result is considered as the final result. *SEQUENTIAL_LOCALHOST* is defined for sequential execution on localhost. *MULTITHREAD_LOCALHOST* is defined if tasks would be executed parallel within multiple threads on client's local workstation.

Relations between classes of Parallel Programming Library and implementation model is illustrated on the UML diagram on figure 4.7.

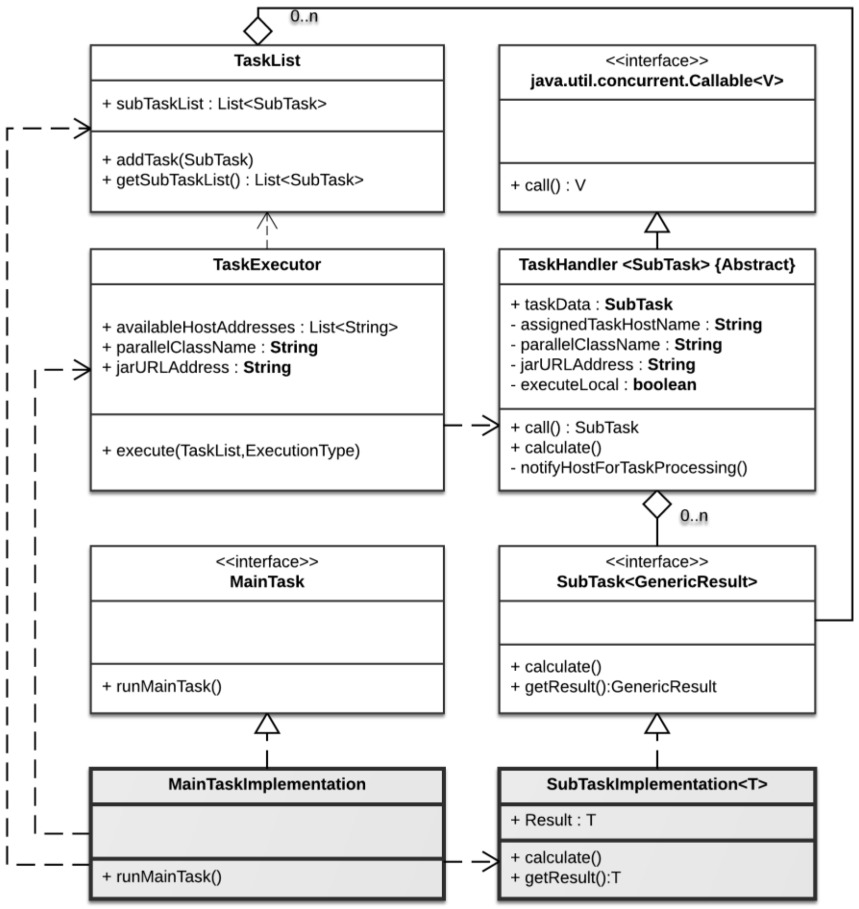


Figure 4.7 : Parallel Programming Library implementation.

4.2.2 Implementation guidelines for developers

Developers using Parallel Programming Framework should follow below guidelines.

- Include ParallelPatternFramework.jar to Java Project
- Create a main class which implements *MainTask* interface. Developer should fill overrided *runMainTask* method. Check Figure 4.8. for implementation example.
- Create a class which implements *SubTask* interface. *Calculate* method is required to be implemented, which will consist the autonomous calculation task made on the distributed task. Check Figure 4.9. for *SubTask* implementation example.

- Developer should export a JAR package form its application, which will be deployed to PRP Repository.
- Parallel Programming Library would be implemented as illustrated on UML in Figure 4.7.

```
@Override
public String runMainTask() {
        TaskExecutor executorInstance = TaskExecutor.getInstance();
        TaskList taskList = new TaskList();
        for (int i = 0; i < 4; i++) {
                long min = 100000000 * i + 1;
                long max = 100000000 * (i+1);
                Sum sum = new Sum(min, max);
                taskList.addTask(sum);
        }

        long sum = 0;
        List<SubTask> resultList =
executorInstance.execute(taskList,ExecutionType.PARALLEL_MULTIHOST_STRICT);

        // Now retrieve the result
        for (SubTask<Long> resultItem : resultList)
        {
                sum += (Long)resultItem.getResult();
        }

        System.out.println("MAIN TASK RESULT : " + sum
        return String.valueOf(sum);

}
```

Figure 4.8 : Main Task implementation example.

```
@Override
public void calculate() {
        sumResult = 0l;
        long tempResult = 0;
         for (long i = from; i <= to; i++) {
                tempResult = tempResult + i;
         }
         sumResult = tempResult;
         System.out.println("SUM between " + from + " " + to + " is " + sumResult);
}
```

Figure 4.9 : Sub Task implementation example.

5. EXPERIMENTAL RESULTS

This section describes the results of experimental tests performed on ITU-PRP. Experiments were conducted on parallel applications developed according Parallel Programing Framework specifications. Multiple hosts located on different networks over Internet have joined and participated the testing sessions. Also, computers with various configuration and computational power were used as hosts. By Task Execution Middleware coordination, a Multi-Host heterogeneous platform was created.

SHA-256 Hash Decoding as a characteristic example for High Performance Computing was selected for case study. Application was developed according Parallel Programming Framework and was uploaded to Task Execution Middleware. As a test scenario, a 6 letter numeric pin encoded with SHA-256 Hash algorithm, will be decoded/decrypted. SHA-256 is a function that encodes a set of characters. However the algorithm has not a reverse function and encoded hash cannot be decoded. In order to decode the Hash, decode function should anticipate the hashed pin by hashing all available combinations and comparing the results with the hashed pin. For example, a pin number 871367 will get below result after performing SHA-256 Hash function.

5513cbf6f4112774fb01961e107714a9f96bee0234a12401f21aef088ac8c1e9

Let’s assume that there is a requirement to decode the hash and get the pin code 871367. Another assumption is that the pin is a 6 letter numeric characters set. In order to find the pin from the hash, hashing function will be performed for all guesses from 100000 to 999999. Decoding function should be implemented like in pseudo code given below.

```
begin
    hash 🡐5513cbf6f4112774fb01961e107714a9f96bee0234a12401f21aef088ac8c1e9
    found🡐false;
    i 🡐 100000
    while found=false and i<=999999  do
    begin
        tempHash 🡐  SHA256(i)
        if tempHash = hash then
        begin
           found🡐true
           decodedPin🡐i
        end
        i 🡐 i+1
    end
end
```

A sequential java console application has decoded SHA-256 hash in 2265 milliseconds. Application was executed on a MacBook Pro laptop with Intel Core i5 2,5 GHz Processor and 8 GB Memory.

SHA-256 Hash decoding application uploaded on ITU-PRP. A client that distributes sub tasks to be executed on resource hosts has performed executions according 4 different task plans provided by Broker. During first test, client process has separated all pin combinations to 2 subtasks, which was executed parallel on 2 resource hosts. Then during each further testing phase an additional resource host has joined the system until a desired result was achieved. Table 5.2 shows resource hosts used for experiments. System details, computational power and operating system of hosts vary and belong to various configurations. These 5 hosts are placed behind different NATs, 3 on a local network, the other 2 on a remote network. All these varieties of hosts form a heterogeneous system, which expresses a typical platform described on this research.

Table 5.1 : Hosts used for experimental results.

Host	System Info	CPU	Memory	Public IP/ Private IP	Operating System
1	Dell Lattitude E6430	Intel Core i7-3630QM 2,4GHz	8GB	92.45.23.114/ 10.1.3.200	Windows 7
2	Asus N550JV-CN127H	Intel Core i7-4700HQ 2,4 GHz	16GB	212.252.141.106/ 192.168.2.107	Windows 8
3	MacBook Air 4,2	Intel Core i5 1,7 GHz	4GB	212.252.141.106/ 192.168.2.155	Mac OS X Mavericks
4	Dell Lattitude E5440	Intel Core i5-4210U 1,7 GHz	4GB	92.45.23.114/ 10.1.3.175	Windows 7
5	MacBook Pro 10,2	Intel Core i5 2,5 GHz	8GB	212.252.141.106/ 192.168.2.103	Mac OS X Mavericks

Client, which performed main task, was a member of 212.252.141.106 network, like other 3 members within same network. Measurement was made based on processing times obtained for 2, 3, 4 and 5 hosts. Detailed results are shown on Table 5.2. While sequential decoding has taken 2265 milliseconds, a first challenge of ITU-PRP with 2 resource hosts was not able to beat an ordinary sequential execution. 2 resources have got a 2358 ms processing time, which is higher than sequential processing. In fact, a sequential process enhanced to 2 parallel processes should achieve a performance gain. However, host 1 is on a remote network far from Client 1 and Host 2 is another end within 212.252.141.106 network. Network delays during message transmissions have caused such a disadvantage. By the attendance of 3rd host, the processing time became head-to-head with sequential processing with a 2253ms of processing time. However, the goal is to beat the sequential processing and this is not achieved yet. Anyway, if the 3rd host would be a resource with stronger computational power, the overall processing time would be lower than the occurred one. According to Parallel Application Execution principals of ITU-PRP, all subtasks should be processed in order to terminate overall processing power. Host 3, which had less computational power and performance compared with others took longer than the others and affected overall processing time by providing the task result later than others. However, this is a good example for impact of heterogeneity over parallel processing on ITU-PRP.

Table 5.2 : SHA-256 processing times.

Application Model	**Hosts**	**Repeated Tests**	**Avg. Processing Time (ms)**	**Speed Up**	**Performance Gain (%)**
Sequential Java Console App.	1	5	2265	-----	------
ITU-PRP 2 Hosts	2	5	2358	0.96	-4,11
ITU-PRP 3 Hosts	3	5	2253	1.01	0.53
ITU-PRP 4 Hosts	4	5	1636	1.38	27.77
ITU-PRP 5 Hosts	5	5	1270	1.78	43.92

A performance optimization was achieved by addition of 4th host, which gave 1636ms of processing times and 1.38 speed up. 5th host also affected positively, which resulted to 1270ms processing time, 1.78 speed up and 43.92 percent performance gain over sequential processing. Figure 5.1. also shows a graphical chart of experimental results for Hash decoding. The overall progress on chart reveals that higher available resource hosts will reduce processing times remarkably. But the processing time would not be beyond network/message transmission delays. So, minimal processing time would be near to network delay times on ideal conditions. And the user should not expect a performance gain for applications that processing times takes less than an ordinary Client-to-Host delay times.

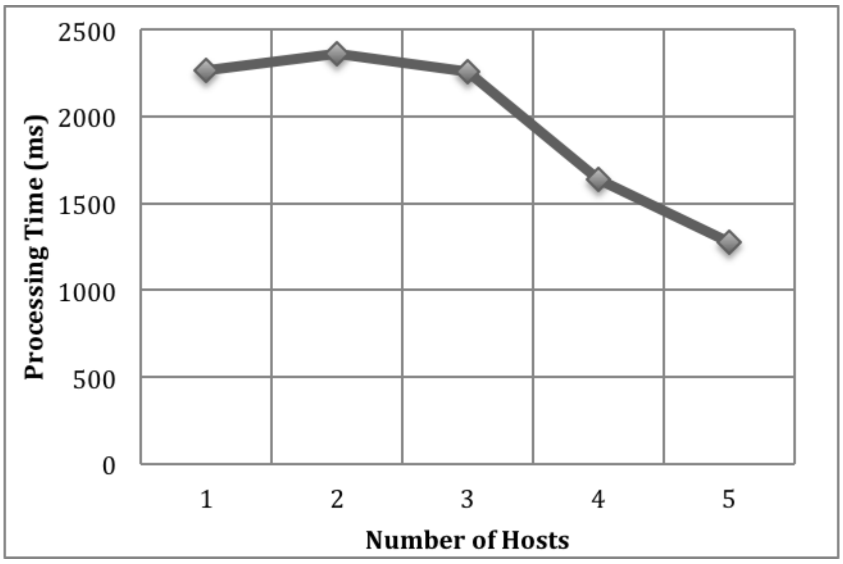


Figure 5.1 : Performance gain achieved on experiments.

Even the performance gain achieved for Hash Decoding reflects a comprehensive performance analysis, any other application implemented and deployed on ITU-PRP may give different performance results. Depending on characteristics of application, the parallel processing performance may be better or worse. The main principal should be applying problems with high processing time or data, for which ITU-PRP promises performance gain.

As mentioned above, minimal processing time would be the maximum delay time within two-way Client-to-Host data transmissions. However, such a delay was not possible to be measured during experiments. In order to measure a delay between two ends, a global synchronized time should be set, which will make possible calculating time differences during transmissions. However, in a heterogeneous platform such as ITU-PRP a global synchronized time is not applicable, so network delay time cannot be extracted from total sub task processing time. Anyway, some estimation can be made to give a delay time during Client-to-Host transmission. Ping tests with 1MB packet data were made in order to get estimated delay times. Ping tests from Istanbul to arbitrary IP addresses to different locations shows that a 1MB ping, to an IP address on New York return in 170-180ms. Other 1MB pings have been measured as, Istanbul-London 110-120ms, Istanbul-Ankara 65-75ms, Istanbul-Istanbul 30-40ms. Which shows the fact that worst case delay time of two way 180ms sub task exchange will affect a round trip delay of 2 x 180ms to a worst case delay of 360ms.

On Data Transmission section, it was mentioned that a buffer size of a transmission message sent to any component of system is defined as 576 Bytes. The regarded buffer size was decided according test results conducted in this section. The observation during the tests was that Multi-Host messages within a local network or between different networks, were affecting transmission performance differently. Within local networks, higher buffer sizes and lower data separation was causing less delays. But transmission during transmissions over Internet, higher buffer sizes were suffering higher packet loss, which was causing unexpected packet loss during Parallel Execution. The highest buffer sizes preserving the reliability of transmission was monitored as 576 Bytes. This value seemed to balance minimal packet loss and higher possible transmission performance.

Another noticeable factor affecting execution performance was power saving modes of laptops used as resource hosts. Especially, laptops performing power saving strategies during their battery consuming mode, were reducing their CPU power. In these cases, reduced CPU power causes higher processing time. However, usage of smaller number of resource hosts during execution is affected by this. Higher available resource hosts absorb such an impact.

6. CONCLUSION

The methodology of Parallel Processing is based on Multi-Threaded task distribution model. ITU-PRP's design on Parallel Processing, in which Main Task creates and sends Sub Tasks to hosts in a within a loop mechanism, aims to provide an object oriented pattern to combine with parallel models. Object oriented pattern and adaptability of this design is also another noticeable feature, compared to conventional native parallel development tools.

ITU-PRP's approach on Data Parallelization is based on user's customization and user is able to define data of Sub Tasks accordingly. Data distribution via Sub Task object serialization ensures users control over data parallelization. Object-based data distribution, instead of message-based distribution is also another feature, which provides flexibility to user to specify data types for distribution.

Peer-to-peer protocol designed for ITU-PRP has been an important issue that defined the scope of the project. Considering the restrictions of maintaining connections between peers over Internet, implementation of peer-to-peer was difficult and required a lot of literature research. A combined protocol with 3 different NAT traversal techniques, such as Relaying, Connection Reversal, UDP Hole Punch made possible connecting hosts over Internet. System was designed that all hosts to connect each other, keep their connection active during their lifecycle. Also, Peer grouping principal aims grouping creating location groups, specific task groups. However, Peer grouping would be possible with the increased number of users and it was kept out of scope, for a future scale increasing of project.

Experimental results have revealed that minimal number of assigned resource hosts have not make a noticeable performance gain during Parallel Application Executions. But higher available resource hosts have achieved remarkable performance optimization results. Tests have also showed that network delays have less impact for negative performances during executions with higher available hosts. Also, applications with higher sequential processing times are more likely applicable to ITU-PRP and easier to achieve a performance optimizations.

Calculated maximum Client-to-Host delay time in Experimental Results section was 360ms. However, real world network problems, may impact to higher delays. Principally the suggestion of ITU-PRP is performing parallelization for sequential applications that take longer processing time then 2 seconds. A higher performance gain is achievable if sequential application with higher processing times is implemented on ITU-PRP. In fact, ITU-PRP does not intend to beat up any multi-core framework and achieve higher performance. The goal of ITU-PRP is providing a heterogeneous Multi-Host parallel processing platform, where some specific High Performance Applications would work on a global environment. And users will be able to benefit a low cost parallel computing environment, which transforms individual idle processing power to a global processing power.

Benchmarking tools applied for performance comparison of other Parallel Programming Frameworks were not applicable to ITU-PRP, due to the heterogeneous Multi-Host architecture formed with a global peer-to-peer protocol. Also, as described above, ITU-PRP has not the goal to challenge the performance of any other parallel framework. Instead it provides a characteristic architecture with specific dynamics in order to achieve a low-cost global parallel processing environment.

Statistics for Parallel Application executions and performance measures will be saved and logged. Any applications statistical information for their recent activity in terms of performance measures would be considered for future executions, so that higher utilization can be achieved on future execution plans.

Higher effective system will be achieved by higher volunteer attendance. A volunteer reward system is provided in order to increase the host numbers in the system. By the increase of available hosts, additional experiments will be done in the future for more concrete measures. Additionally, an established community would get feedbacks from users, in terms of discussions about system’s performance and issues.

REFERENCES

[1] **Thomas Rauber, Gudula Rünger (2010),** *Parallel Programming for Multicore and Cluster Systems*. 2nd Edition, Springer Heidelberg Dordrecht, London, New York

[2] **Sartaj Sahni, George Vairaktarakis,** 1996: *The Master-Slave Paradigm in Parallel Computer and Industrial Settings*, Journal of Global Optimization, Springer

[3] **URL-1** <*http://www.oracle.com/technetwork/articles/java/fork-join-422606.html*>

[4] **Michael J. Quinn,** (2004), *Parallel Programming in C with MPI and OpenMP*, 1st Edition, McGrawHill, New York

[5] **URL-2** <*http://mpj-express.org/docs/guides/windowsguide.pdf*> accessed at 04.04.2013

[6] **URL-3** < *https://code.google.com/p/javampi/*>, accessed at 04.04.2013

[7] **M. Klemm, M. Bezold, R. Valdema and M. Philippsen,** (2007) : JaMP: An Implementation of OpenMP for a Java DSM, University of Erlangen-Nuremberg, Computer Science Department, Erlangen, Germany

[8] **URL-4 <***http://jade.tilab.com/doc/index.html*>, accessed at 16.03.2014.

[9] **Peter Cappello, Bernd Christiansen, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser and Daniel Wu,** 1997: Javelin: Internet-Based Parallel Computing Using Java, University of Bristol, Department of Computer Science, University of California, Santa Barbara.

[10] **Michael O. Neary, Alan Phipps, Steven Richman and Peter Cappello,** 2000: Javelin 2.0: Java-based parallel computing on the Internet, University of California, Santa Barbara

[11] **L. F. Lau, A. L. Ananda, G. Tan, W. F. Wong,** 2000: JAVM: Internet-based Parallel Computing Using Java, School of Computing, National University of Singapore.

[12] **Bryan Ford, Pyda Srisuresh, Dan Kegel,** 2005: Peer-to-Peer Communication Across Network Address Translators, Massachusetts Institute of Technology, Caymas Systems, Inc..

APPENDICES

APPENDIX A

DEPLOYMENT OF ITU-PRP

The related files for deployment of Task Execution Environment are given under DEPLOYMENT folder of attached CD:

- war/itu_prp.war
- tomcat/apache-tomcat-7.0.39
- database/ituprp_test.sql

In order to setup a Server for providing the hosting service of ITU-PRP System, the deployable war package and database backup files are provided. itu_prp.war will be deployed to an Apache Tomcat Web Server. The Web Server may be downloaded from http://tomcat.apache.org/download-70.cgi but is provided on attachment CD as well. ituprp_test.sql is MySQL database backup file for ITU-PRP Database. A MySQL Server should be installed and set up on a Server where Database will be hosted. ITU-PRP Database is deployed by running ”mysql –u root –p ituprp_test < ituprp_test.sql” command. This command will create ituprp_test database to Server.

Apache Tomcat may be set up to any of Linux, Windows or Mac OS X operating systems. The regarded folder of apache-tomcat-7.0.39 may be placed to a preferable folder within the file system. The bin folder contains startup and shutdown (.bat for Windows, .sh for the rest) files, which start and terminate the Server Service. The conf folder contains server.xml and other files on which Tomcat is configured. Connector port of installed server may be changed from server.xml, which is 8080 as default. The webapps folder is the location where applications are deployed.

itu_prp.war file will be copied to webapps folder in order to deploy the Application. Then startup (.bat or .sh) file is executed in order to start the Web Server. In order to verify if Tomcat Server is started successfully, http://localhost:8080/ is opened on a Web Browser and the default page of the Tomcat is expected to show up. For verification if Task Execution Environment has been deployed successfully http://localhost:8080/itu_prp/Login.xhtml would be opened on a Web Browser, where the Web Application of Task Execution environment is showed up.

Deployed ITU-PRP application is configured by modifying below files:

Application.properties: IP, port and ping intervals of Broker are set on this file. Also uploaded jar file path is set withing this configuration file.

Persistence-mysql.properties: MySQL connection parameters are set in this configuration file.

The key point about Task Execution Environment is the connectivity of the Web Application globally. Due to frequent TCP Socket operations between Server and hosts, a global IP address should be assigned to the Server where the Task Execution Environment is deployed, in order to serve globally for parallel processing. The second alternative is providing parallel processing within a network subnet. In this case, a private IP of Task Execution Environment Server is assigned and parallel processing is performed within a local network scope.

APPENDIX B

SOURCES AND DEVELOPMENT ENVIRONMENT SETUP

The related files of Development Environment of the project are provided under SOURCES folder CD:

- ITU_PRP Maven Project
- Eclipse Installation Files
- ParallelPatternFramework.jar Parallel Programming Environment API
- ITU_PRP_IMPLEMENTATION Eclipse Project

The source codes of ITU_PRP project are in SOURCES/ITU-PRP folder of attachment CD. The related project is developed under Eclipse development environment and should be imported to Eclipse for further development. In order to work on the project the latest version of Eclipse can be downloaded on www.eclipse.org. The Eclipse IDE for Java EE Developers is preferable, which already contains configuration and plug-ins for Java EE Web Applications. Also, Eclipse installation with all required project is provided in APPLICATIONS folder of attached CD. It is preferred to use provided Eclipse installation.

The Parallel Programming Environment API is provided on the SOURCES folder as well. This the JAR Library to be used by Parallel Application developers which will create Parallel Applications according to specifications of ITU-PRP given on the implementation section. The source code of a sample implementation for Parallel Programming Framework is given as an Eclipse Project in SOURCES/ITU_PRP_IMPLEMENTATION folder. The project contains source code of SHA256 Hash Decoding example shown on experimental results section.

CURRICULUM VITA

Name Surname: Enis SPAHI

Place and Date of Birth: Prizren / Kosova, 16.09.1985

Address: Gülbahar Mah, Bıldırcın Sk. 14/6, Mecidiyeköy/İstanbul

E-Mail: enisspahi@gmail.com

B.Sc.: Yıldız Technical University, Computer Engineering

Professional Experience and Rewards: Garanti Teknoloji (July, 2008 – February, 2009); AEC Teknoloji (March, 2009 – October, 2009); Hitit Computer Services (November, 2009 - Present)

List of Publications and Patents:

PUBLICATIONS/PRESENTATIONS ON THE THESIS

- **Enis Spahi and D. Turgay Altılar,** 2014: ITU-PRP: Parallel and Distributed Middleware for Java Developers. *1st International Conference in Computer Science, Information System and Telecommunication (ICCSIST 2014)*, November 7-8, 2014, Durres, Albania.