İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

ASYNCHRONOUS JAVA RMI

M.Sc. Thesis by
Orhan AKIN

Department : Computer Engineering

Programme : Computer Engineering

Thesis Supervisor: Prof. Dr. Nadia ERDOĞAN

JUNE 2009

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

ASENKRON JAVA RMI

M.Sc. Thesis by
Orhan AKIN
(504061527)

Date of submission : **02 May 2009**
Date of defence examination: **08 June 2009**

Supervisor : **Prof. Dr. Nadia ERDOĞAN (ITU)**
Members of the Examining Committee : **Prof. Dr. Oya KALIPSIZ (YTU)**
Assis. Prof. Dr. Feza BUZLUCA (ITU)

JUNE 2009

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

ASENKRON RMI

YÜKSEK LİSANS TEZİ

Orhan AKIN
(504061527)

Tezin Enstitüye Verildiği Tarih : 02 Mayıs 2009
Tezin Savunulduğu Tarih : 08 Haziran 2009

Tez Danışmanı : Prof. Dr. Nadia ERDOĞAN (İTÜ)
Diğer Jüri Üyeleri : Prof. Dr. Oya KALIPSIZ (YTÜ)
Yrd. Doç. Dr. Feza BUZLUCA (İTÜ)

HAZİRAN 2009

FOREWORD

I would like to express my deep appreciation and thanks for my advisor Prof. Dr. Nadia Erdoğan. She was always very helpful and patient to me, I have come up with a better performance and more efficient usage of my asynchronous RMI framework with her ideas and directions. Almost every week we had meetings on this study. I also would like thanks to her for the contributions to our paper 'Enhancing Java RMI with Asynchrony Trough Reflection'. I took some classes like parallel programming, advanced operating systems from Mrs. Erdoğan, and had been knowing her for two years, besides her technical/theoretical knowledge, personally she is really constructive, creative and modest person.

In addition, special thanks to ITU Mustafa Inan Library personnel for their understanding, they always had to warn me that it was 09:30pm and library was closing.

June 2009

Orhan Akın

Computer Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS

ARMI	**:** Asynchronous Remote Method Invocation
CORBA	**:** Common Object Request Broker Architecture
GC	**:** Garbage Collector
IDL	**:** Interface Definition Language
IP	**:** Internet Protocol
JDK	**:** Java Development Kit
JRE	**:** Java Runtime Environment
ORB	**:** Object Request Broker
POJO	**:** Plain Old Java Object
RMI	**:** Remote Method Invocation
RPC	**:** Remote Procedure Call
SDK	**:** Standard Development Kit
TCP	**:** Transmission Control Protocol
UDP	**:** User Datagram Protocol
VM	**:** Virtual Machine

LIST OF TABLES

Page

LIST OF FIGURES

Page

ASYNCHRONOUS RMI

SUMMARY

Java RMI's synchronous invocation model may cause scalability challenges when long duration invocations are targeted. One way of overcoming this difficulty is adopting an *asynchronous* mode of operation. An asynchronous invocation allows the client to continue with its computation after dispatching a call, thus eliminating the need to wait idle while its request is being process by a remote server. In this study we define and implement an execution framework which extends Java RMI functionality with asynchrony.

From out of the box point of view, there is a java client program and a server program both written in java programming language and living in different virtual machines that can be in different physical locations and using Java's RMI (Remote Method Invocation) technology to communicate. Considering previous work carried on this topic to achieve asynchrony on Java's RMI special preprocessing of the client stub, modification of the remote objects or stubs or modification of the Sun's Java VM (virtual machine) is required which is not straight forward since Sun does not provide built-in mechanisms for those operations.

In this study, we implemented an asynchronous RMI execution framework on the top of RMI technology, using the thread pooling capability and the reflection mechanism of Java. It differs from previous work as it does not require any external tool, preprocessor, or compiler and it may be integrated with previously developed software, as no modification of target remote objects is necessary. User of the framework is provided a simple jar packet with couple of basic classes to make asynchronous calls to remote objects that (s)he develops or had been developed.

Brief code examples with real life problems will be given with the usage of the asynchronous RMI framework, obvious performance gain over standard RMI calls will be shown and the techniques which were used to optimize Java's multi threading and reflection in this study will be explained briefly with comparisons and results.

ASENKRON RMI

ÖZET

Senkron çalışan Java RMI'ın senkron çağrı modeli uzun zaman alan uzak yordam çağrıları için ölçeklenebilirlik sorunları yaratmaktadır. Bu problemleri aşmanın bir yöntemi de uzak yordamı asenkron bir çağrı modeline uyarlamaktır. Asenkron çağrılar sayesinde, istemci çağrıyı yürüttükten sonra sunucu çağrıyı işlerken gereksiz yere beklemez ve çalışmasına devam edebilmektedir. Bu çalışmada java uzak yordam çağırma yöntemleri üstüne asenkron olarak uzak yordam çağırmamızı sağlayacak bir yazılım çatısının tanımını yapıp gerçeklemesini anlatacağız.

Dışarıdan bakacak olursak, java programlama dili ile yazılmış, çoğu zaman farklı fiziksel yerlerde bulunan farklı sanal makineler üzerinde yaşayan ve birbirleri ile RMI ile haberleşen istemci ve sunucu programlar bulunmaktadır. Konu üzerinde asenkronluk özelliğin eklenmesi için yapılmış önceki çalışmalara bakacak olursak; benimsenen yöntemler istemci vekilin ön işlemden geçirilmesi, uzak nesnelerin veya uzak nesne vekillerinin değiştrilmesi ya da Sun'ın Java Sanal Makinesi'nin değiştirilmesi gibi pek de kolay olmayan çalışmalar gerektirmektedirler.

Bu çalışmada Java'nın iplik havuzu ve yansıtma gibi becerilerinden yararlanarak, kendi uzak yordam çağırma kütüphanesinin üstüne asenkron uzak yordam çağırma yazılım çatısı gerçeklemekteyiz. Çalışmamız Java dışında farklı bir araç, ön işleme, özel derleyici gerektirmemesi, hali hazırda geliştirilmiş olan eski uzak nesnelerde değişikliğe gerek duymayıp onlarla sorunsuz entegre çalışabilmesi açışından önceki çalışmalara göre farklılık göstermektedir. Kullanıcıya sadece bir jar paketi sunulmakta olup, yazılım çatısı içindeki temel birkaç sınıf kullanılarak yeni geliştirilen uzak nesneler veya eskiden geliştirilmiş uzak nesneler asenkron olarak kullanılabilmektedir.

Asenkron yazılım çatısının kullanımı gerçek dünya problemlerinin çözümünde ne şekilde kullanıldığı, açık kod örnekleriyle verilmektedir. Java'nın standart uzak yordam çağırma yöntemine göre nasıl kazanç sağladığımız açıkca ortaya konup, java çok iplikli programlama ve yansıtma yöntemlerinin performans kazanmak için optimize edilme yöntemleri karşılaştırmalar ve sonuçlarla ortaya konmaktadır.

1. INTRODUCTION

Communication is a fundamental issue in distributed computing. Middle-ware systems offer a high level of abstraction that simplifies communication between distributed object components. Java's Remote Method Invocation (RMI) [1] mechanism is such an abstraction that supports an object-based communication framework. It extends the semantics of local method calls to remote objects, by allowing client components to invoke methods of objects that are possibly located on remote servers. In RMI, method invocation is *synchronous*, that is, the operation is blocking for the client if it needs the result of the operation or an acknowledgement. This approach generally works fine in LANs where communication between two machines is generally fast and reliable [2]. However, in a wide-area system, as communication latency grows with orders of magnitude, synchronous invocation may become a handicap. In addition, RMI's synchronous invocation model may cause scalability challenges when long duration invocations are targeted. One way of overcoming these restrictions is adopting an *asynchronous* mode of operation. This type of invocation provides the client with the option of doing some useful work while its call request is being processed, instead of being blocked until the results arrive. Asynchronous invocations have the following advantages:

- to overlap local computation with remote computation and communication in order to tolerate high communication latencies in wide-area distributed systems,
- to anticipate the scheduling and the execution of activities that do not completely depend on the result of an invocation,
- to easily support interactions for long-running transactions
- to enforce loose coupling between clients and remote servers

1.1 Purpose of the Thesis

We propose an execution framework which extends RMI functionality with asynchrony. Clients are equipped with an interface through which they can make asynchronous invocations on remote objects and continue execution without the need to wait for the result of the call. They can later on stop to query the result of the call if it is required for the computation to proceed, or they may completely ignore it, as some invocations may not even produce a result.

Our framework focuses on the four most commonly used techniques, namely fire and forget, sync with server, polling, and result callback for providing client-side asynchrony. The design of the framework is based on a set of asynchrony patterns for these techniques described in detail in [3]; actually implementing the asynchrony patterns on top of synchronous RMI calls.

This study describes the design and implementation issues of the proposed framework. Java's threading capability has been used to provide a mechanism for asynchronous invocation. One contribution of this work is its use of run-time reflection to do the remote invocation, thus eliminating the need for any byte code adjustments or a new RMI preprocessor/compiler. Another contribution is the rich set of asynchronous call alternatives it provides the client with, which are accessible through an interface very similar to that of traditional RMI calls. As the framework is implemented on the client side and requires no modification on server code, it is easily integrated with existing server software.

1.2 Background

The Agents project [27] is a collection of class libraries implement and combine several important features that are essential to supporting distributed and parallel computing using Java. Such features include: the ability to easily create objects on remote hosts, interact with those objects through either synchronous or asynchronous remote method invocations, and to freely migrate objects to heterogeneous hosts. But its asynchronous communication aspect does not provide much functionally, only the Polling pattern is implemented partially using the 'Feature' objects.

The ARMI project [28] is built on the top of Java RMI and allows concurrent execution of local and remote computations. To achieve asynchrony special compiler

(armic) which generates stubs that handle the asynchronous communication. These stubs provide synchronous communication by wrapping each remote method invocation inside a thread. Server side of the connection must also be threaded which means we can not use ARMI with past remote objects (remote objects written by others).

Object Systems has recently provided a communication system called Voyager [4].

Voyager provides facilities for remote invocation, as well as the creation of mobile computing agents. It provides several communication modes allowing for synchronous invocation, asynchronous invocations with no reply (oneway), and asynchronous invocation with a reply (futures/promise). The major difference between ARMI and Voyager is that Voyager aims at providing a completely new, more comprehensive, system, thus making it incompatible with currently existing RMI technology [5]. There have been many efforts to introduce asynchronous facilities into programming languages and communication libraries. Examples of asynchronous RPCs may be found in [6–8]. Nexus Communication System, which is described in [9, 10], supports asynchronous invocations. It is intended as a target for parallel compilers. It is a flexible and a powerful framework, but unlike the ARMI programs, which are simple to write, programs written using Nexus are verbose and complex.

There are several projects investigating fast implementations of RMI. The Manta project [11] is investigating a high performance implementation of RMI through native compilation and their own marshaling and serialization protocols. The Manta approach attempts to overcome the latency issues in Java RMI by performing as much compile time analysis of serialization as possible in place of time consuming reflection. Manta also allows dynamic development of serialization code for remote objects to support polymorphic operations. Manta has been able to achieve near RPC performance from their specialized RMI implementation, which is a significant cost improvement for Java RMI. Manta supports both standard Java RMI and its own serialization protocols so, although they are not compliant at the underlying protocol level, it is possible to interoperate with standard Java RMI classes but with no performance increase.

The JavaParty project [12] adopts a similar approach to Manta, using an efficient serialization protocol but without static native code compilation. The JavaParty implementation faces similar problems to that of the Manta project; these projects can be viewed as non-standard drop-in replacements for RMI.

Sampemane [13] and Java/DSM [14] describe implementations of JVMs and RMI on top of other distribution protocols. Sampemane describes an implementation of RMI above Fast Messages [15], while the Java/DSMproject explores an implementation above Treadmarks [16]. Both of these methods maintain the flexibility of RMI by retaining the standard API; however they share the interoperability problems of the Manta and JavaParty projects.

The NinjaRMI project [17], part of the UCSB Ninja [18] project, is a completely rebuilt version of RMI with extended features, including asynchronous communication. However, NinjaRMI is not wire compliant with standard Java RMI. NinjaRMI is not a high performance version of RMI but is intended to provide language extensions and exhibits performance at least as good as Sun's implementation.

The HORB project [19, 20] introduces a Java ORB with an alternative communications protocol to RMI that supports both synchronous and asynchronous communication. Using the HORB communications mechanisms, performance close to C sockets is achieved; however HORB supports a completely different wire protocol to standard Java RMI and hence can not be used in combination with standard Java RMI objects.

Implementations of CORBA [21, 22] over Java allow for one-way or asynchronous method invocations using IIOP as the underlying protocol. These costs have been investigated in [23, 24].

These improved serialization and RMI protocols implemented in the systems described above may eventually be integrated into the Java RMI standard to improve performance, but currently they are not interoperable with clients or servers using standard Java [25].

1.3 Objectives

This study provides asynchronous method calls to remote java objects, with the following objectives as advantages to previous related work;

Independence of any external tools, preprocessors, or compilers:

Our main objective has been to present an execution framework that is completely compatible with standard Java, compilers, and run-time systems and does not require any preprocessing or a modified stub compiler. We have used RMI as the underlying communication mechanism and implemented asynchrony patterns on top RMI calls (as depicted in Figure 1.). Therefore, as long as a client is able to access a remote object using standard RMI, both invocation models, asynchronous /synchronous invocations, are possible. The main benefit for the developer is that he may choose the appropriate invocation model according to the needs of the application.

No modification of existing server software:

Another design objective is that the framework should require no modifications on the server side so that previously developed remote objects can still be accessed, now asynchronously as well. There is no necessity for a remote object to implement a particular interface or to be derived from a certain class as the framework is located on the client side.

Performance related concerns:

Performance issues have also been the focus of the implementation. As threading and reflection produce extra runtime overhead, special care has been taken to keep their use at minimum, so that they do not introduce a performance penalty on method execution.

2. JAVA RMI

RMI is the distributed object system that is built into the core Java environment. RMI can thought as a built-in facility for Java that allows to interact with objects that are actually running in Java virtual machines on remote hosts on the network. With RMI, a reference can be got to an object that "lives" in a remote process and invoked methods on it as if it were a local object running within the same virtual machine as local code (hence the name, "Remote Method Invocation API").

2.1 Introduction to RMI

RMI was added to the core Java API in Version 1.1 of the JDK (and enhanced for Version 1.2 of the Java 2 platform), in recognition of the critical need for support for distributed objects in distributed–application development. Prior to RMI, writing a distributed application involved basic socket programming, where a "raw" communication channel was used to pass messages and data between two remote processes. Now, with RMI and distributed objects, an object can be “exported” as a remote object, so that other remote processes/agents can access it directly as a Java object. So, instead of defining a low–level message protocol and data transmission format between processes in distributed application, Java interfaces can be used as the "protocol" and the exported method arguments become the data transmission format. The distributed object system (RMI in this case) handles all the underlying networking needed to make your remote method calls work.

Java RMI is a Java–only distributed object scheme; the objects in an RMI–based distributed application have to be implemented in Java. Some other distributed object schemes, most notably CORBA, are language–independent, which means that the objects can be implemented in any language that has a defined binding. With CORBA, for example, bindings exist for C, C++, Java, Smalltalk, and Ada, among other languages. The advantages of RMI primarily revolve around the fact that it is "Java–native." Since RMI is part of the core Java API and is built to work directly with Java objects within the Java VM, the integration of its remote object facilities

into a Java application is almost seamless. RMI–enabled objects can really be used as if they live in the local Java environment. And since Java RMI is built on the assumption that both the client and server are Java objects, RMI can extend the internal garbage–collection mechanisms of the standard Java VM to provide distributed garbage collection of remotely exported objects.

2.2 RMI in Action

Before examining the details of using RMI, a simple RMI remote object at work example will be given. We can create an Account object that represents some kind of bank account and then use RMI to export it as a remote object so that remote clients (e.g., ATMs, personal finance software running on a PC) can access it and carry out transactions.

The first step is to define the interface for our remote object. 0shows the Account interface. You can tell that it's an RMI object because it extends the java.rmi.Remote interface. Another signal that this is meant for remote access is that each method can throw a java.rmi.RemoteException. The Account interface includes methods to get the account name and balance and to make deposits, withdrawals, and transfers.[26]

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;
public interface Account extends Remote {
      public String getName() throws RemoteException;
      public float getBalance() throws RemoteException;
      public void withdraw(float amt) throws RemoteException;
      public void deposit(float amt) throws RemoteException;
      public void transfer(float amt, Account src) throws
                                                  RemoteException;
  public void transfer(List amts, List srcs) throws RemoteException;
}
```

Figure 2.1 : A Remote Account Interface

The next step is to create an implementation of this interface, which leads to the AccountImpl class shown in Figure 2.2. This class implements all the methods listed in the Account interface and adds a constructor that takes the name of the new account to be created.

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.util.List;
import java.util.ListIterator;
```

```
public class AccountImpl extends UnicastRemoteObject implements
Account {
private float mBalance = 0;
private String mName = "";
// Create a new account with the given name
public AccountImpl(String name) throws RemoteException {
      mName = name;
}
public String getName() throws RemoteException {
      return mName;
}
public float getBalance() throws RemoteException {
      return mBalance;
}
// Withdraw some funds
public void withdraw(float amt) throws RemoteException {
   mBalance -= amt;
   // Make sure balance never drops below zero
   mBalance = Math.max(mBalance, 0);
}
// Deposit some funds
public void deposit(float amt) throws RemoteException {
      mBalance += amt;
}
// Move some funds from another (remote) account into this one
public void transfer(float amt, Account src) throws RemoteException
{
      src.withdraw(amt);
      this.deposit(amt);
}
// Make several transfers from other (remote) accounts into this one
public void transfer(List amts, List srcs) throws RemoteException {
      ListIterator amtCurs = amts.listIterator();
      ListIterator srcCurs = srcs.listIterator();
      // Iterate through the accounts and the amounts to be
      // transferred from
      // each (assumes amounts are given as Float objects)
      while (amtCurs.hasNext() && srcCurs.hasNext()) {
           Float amt = (Float)amtCurs.next();
           Account src = (Account)srcCurs.next();
           this.transfer(amt.floatValue(), src);
      }
}
}
```

Figure 2.2 : Implementation of the Remote Account Interface

Once the remote interface and an implementation of it are complete, both Java files needs to be compiled with a Java compiler. After this is done, the RMI stub/skeleton compiler is used to generate a client stub and a server skeleton for the AccountImpl object. The stub and skeleton handle the communication between the client application and the server object. With Sun's Java SDK, the RMI compiler is called *rmic*, and can invoked for this example like so:

% rmic –d /home/classes AccountImpl

The stub and skeleton classes are generated and stored in the directory given by the –d option (*/home/classes*, in this case). This example assumes that the AccountImpl class is already in the CLASSPATH before runing the RMI compiler. There's just

one more thing needs to be done before we can actually use our remote object: register it with an RMI registry as shown in Figure 2.3, so that remote clients can find it on the network. The utility class that follows, RegAccount, does this by creating an AccountImpl object and then binding it to a name in the local registry using the java.rmi.Naming interface. After it's done registering the object, the class goes into a wait(), which allows remote clients to connect to the remote object:

```
import java.rmi.Naming;
public class RegAccount {
      public static void main(String argv[]) {
      try {
         // Make an Account with a given name
         AccountImpl acct = new AccountImpl("Oakin");
         // Register it with the local naming registry
         Naming.rebind("Oakin", acct);
         System.out.println("Registered account.");
      }
      catch (Exception e) {
        e.printStackTrace();
      }
      }
}
```

Figure 2.3 : Register Remote Account

After the RegAccount class is compiled, its main() method can be run to register an Account with the local RMI registry. First, however, the registry needs to be started. With Sun's Java SDK, the registry can be started using the *rmiregistry* utility. On a Unix machine, this can be done like so:

objhost% rmiregistry &

Once the registry is started, the main() method on the RegAccount class can be invoked simply by running it:

objhost% java RegAccount

Registered account. Now we have a remote Account object that is ready and waiting for a client to access it and call its methods. The following client code in Figure 2.4 does just this, by first looking up the remote Account object using the java.rmi.Naming interface (and assuming that the Account object was registered on a machine named *objhost.org*), and then calling the deposit method on the Account object:

```
import java.rmi.Naming;
public class AccountClient {
      public static void main(String argv[]) {
      try {
      // Lookup account object
      Account oakinAcct =
```

```
                    (Account)Naming.lookup("rmi://objhost.org/Oakin");
        // Make deposit
        oakinAcct.deposit(12000);
        // Report results and balance.
        System.out.println("Deposited 12,000 into account owned by " +
                                                oakinAcct.getName());
        System.out.println("Balance now totals: " +
                                               oakinAcct.getBalance());
        }
        catch (Exception e) {
        System.out.println("Error while looking up account:");
        e.printStackTrace();
        }
        }
}
```

Figure 2.4 : Remote account client

The first time you run this client, here's what you'd do:

% java AccountClient

Deposited 12,000 into account owned by Oakin

Balance now totals: 12000.0

For the sake of this example, it is assumed that the client process is running on a machine with all the necessary classes available locally (the Account interface and the stub and skeleton classes generated from the AccountImpl implementation

2.3 RMI Architecture

There are three layers that comprise the basic remote–object communication facilities in RMI:

- The *stub/skeleton* layer, which provides the interface that client and server application objects use to interact with each other.
- The *remote reference* layer, which is the middleware between the stub/skeleton layer and the underlying transport protocol. This layer handles the creation and management of remote object references.
- The *transport protocol* layer, which is the binary data protocol that sends remote object requests over the wire.

These layers interact with each other as shown in Figure 2.5. In this figure, the server is the application that provides remotely accessible objects, while the client is any remote application that communicates with these server objects. In a distributed

object system, the distinctions between clients and servers can get pretty blurry at times. Consider the case where one process registers a remote–enabled object with the RMI naming service, and a number of remote processes are accessing it. We might be tempted to call the first process the server and the other processes the clients. But what if one of the clients calls a method on the remote object, passing a reference to an RMI object that's local to the client. Now the server has a reference to and is using an object exported from the client, which turns the tables somewhat. The "server" is really the server for one object and the client of another object, and the "client" is a client and a server, too. For the sake of discussion, a process in a distributed application will be refered as a server or client if its role in the overall system is generally limited to one or the other. In peer–to–peer systems, where there is no clear client or server, I'll refer to elements of the system in terms of application–specific roles (e.g., chat participant, chat facilitator).

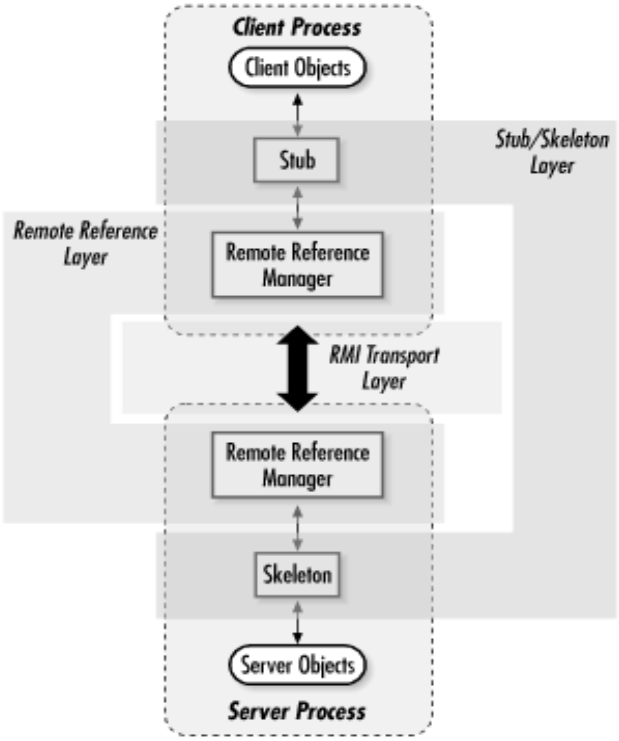

Figure 2.5 : The RMI runtime architecture

As you can be seen in Figure 2.5, a client makes a request of a remote object using a client–side stub; the server object receives this request from a server–side object skeleton. A client initiates a remote method invocation by calling a method on a stub

object. The stub maintains an internal reference to the remote object it represents and forwards the method invocation request through the remote reference layer by *marshalling* the method arguments into serialized form and asking the remote reference layer to forward the method request and arguments to the appropriate remote object. Marshalling involves converting local objects into portable form so that they can be transmitted to a remote process. Each object is checked as it is marshaled, to determine whether it implements the java.rmi.Remote interface. If it does, its remote reference is used as its marshaled data. If it isn't a Remote object, the argument is serialized into bytes that are sent to the remote host and reconstituted into a copy of the local object. If the argument is neither Remote nor Serializable, the stub throws a java.rmi.MarshalException back to the client.

If the marshalling of method arguments succeeds, the client−side remote reference layer receives the remote reference and marshaled arguments from the stub. This layer converts the client request into low−level RMI transport requests according to the type of remote object communication being used. In RMI, remote objects can (potentially) run under several different communication styles, such as point−to−point object references, replicated objects, or multicast objects. The remote reference layer is responsible for knowing which communication style is in effect for a given remote object and generating the corresponding transport−level requests. In the current version of RMI (Java 2), the only communication style provided out of the box is point−to−point object references. For a point−to−point communication, the remote reference layer constructs a single network−level request and sends it over the wire to the sole remote object that corresponds to the remote reference passed along with the request.

On the server, the server−side remote reference layer receives the transport−level request and converts it into a request for the server skeleton that matches the referenced object. The skeleton converts the remote request into the appropriate method call on the actual server object, which involves *unmarshalling* the method arguments into the server environment and passing them to the server object. As might be expected, unmarshalling is the inverse procedure to the marshalling process on the client. Arguments sent as remote references are converted into local stubs on the server, and arguments sent as serialized objects are converted into local copies of the originals.

If the method call generates a return value or an exception, the skeleton marshals the object for transport back to the client and forwards it through the server reference layer. This result is sent back using the appropriate transport protocol, where it passes through the client reference layer and stub, is unmarshaled by the stub, and is finally handed back to the client thread that invoked the remote method.

2.4 RMI Object Services

On top of its remote object architecture, RMI provides some basic object services to be used in distributed application. These include an object naming/registry service, a remote object activation service, and distributed garbage collection.

2.4.1 Naming/Registry Service

When a server process wants to export some RMI–based service to clients, it does so by registering one or more RMI–enabled objects with its local RMI registry (represented by the Registry interface). Each object is registered with a name clients can use to reference it. A client can obtain a stub reference to the remote object by asking for the object by name through the Naming interface. The Naming.lookup() method takes the fully qualified name of a remote object and locates the object on the network. The object's fully qualified name is in a URL–like syntax that includes the name of the object's host and the object's registered name.

It's important to note that, although the Naming interface is a default naming service provided with RMI, the RMI registry can be tied into other naming services by vendors. Sun has provided a binding to the RMI registry through the Java Naming and Directory Interface (JNDI)[29]

Once the lookup() method locates the object's host, it consults the RMI registry on that host and asks for the object by name. If the registry finds the object, it generates a remote reference to the object and delivers it to the client process, where it is converted into a stub reference that is returned to the caller. Once the client has a remote reference to the server object, communication between the client and the server commences.

2.4.2 Object Activation Service

The remote object activation service provides a way for server objects to be started on an as−needed basis. Without remote activation, a server object has to be registered with the RMI registry service from within a running Java virtual machine. A remote object registered this way is only available during the lifetime of the Java VM that registered it. If the server VM halts or crashes for some reason, the server object becomes unavailable and any existing client references to the object become invalid. Any further attempts by clients to call methods through these now−invalid references result in RMI exceptions being thrown back to the client.

The RMI activation service provides a way for a server object to be activated automatically when a client requests it. This involves creating the server object within a new or existing virtual machine and obtaining a reference to this newly created object for the client that caused the activation. A server object that wants to be activated automatically needs to register an activation method with the RMI activation daemon running on its host.

2.4.3 Distributed Garbage Collection

The last of the remote object services, distributed garbage collection, is a fairly automatic process that application developers should never have to worry about. Every server that contains RMI−exported objects automatically maintains a list of remote references to the objects it serves. Each client that requests and receives a reference to a remote object, either explicitly through the registry/naming service or implicitly as the result of a remote method call, is issued this remote object reference through the remote reference layer of the object's host process. The reference layer automatically keeps a record of this reference in the form of an expirable "lease" on the object. When the client is done with the reference and allows the remote stub to go out of scope, or when the lease on the object expires, the reference layer on the host automatically deletes the record of the remote reference and notifies the client's reference layer that this remote reference has expired. The concept of expirable leases, as opposed to strict on/off references, is used to deal with situations where a client−side failure or a network failure keeps the client from notifying the server that it is done with its reference to an object.

When an object has no further remote references recorded in the remote reference layer, it becomes a candidate for garbage collection. If there are also no further local references to the object (this reference list is kept by the Java VM itself as part of its normal garbage−collection algorithm), the object is marked as garbage and picked up by the next run of the system garbage collector.

2.5 Defining Remote Objects

Defining a remote RMI object involves specifying a remote interface for the object, then providing a class that implements this interface. The remote interface and implementation class are then used by RMI to generate a client stub and server skeleton for your remote object. The communication between local objects and remote objects is handled using these client stubs and server skeletons. The relationships among stubs, skeletons, and the objects that use them are shown in Figure 2.6.

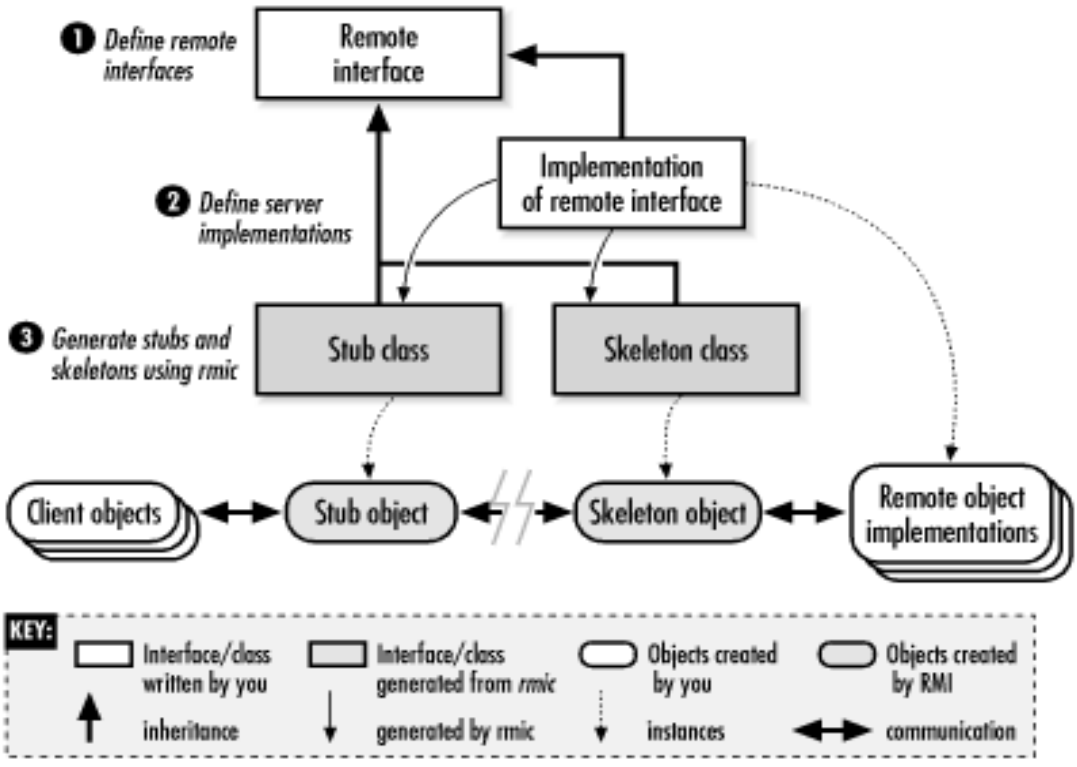

Figure 2.6 : Relationships among remote object, stub, and skeleton classes

When a client gets a reference to a remote object and then calls methods on this object reference, there needs to be a way for the method request to get transmitted

back to the actual object on the remote server and for the results of the method call to get transmitted back to the client. This is what the generated stub and skeleton classes are for. They act as the communication link between the client and your exported remote object, making it seem to the client that the object actually exists within its Java VM.

The RMI compiler (*rmic*) automatically generates these stub and skeleton classes. Based on the remote interface and implementation class provided, *rmic* generates stub and skeleton classes that implement the remote interface and act as go−between for the client application and the actual server object. For the client stub class, the compiler generates an implementation of each remote method that simply packages up (marshals) the method arguments and transmits them to the server. For the server skeleton class, the RMI compiler generates another set of implementations of the remote methods, but these are designed to receive the method arguments from the remote method call, unpackage(unmarshal) them, and make the corresponding method call on the object implementation. Whatever the method call generates (return data or an exception), the results are packaged and transmitted back to the remote client. The client stub method (which is still executing at this point) unpackages the results and delivers them to the client as the result of its remote method call.

So, the first step in creating remote objects is to define the remote interfaces for the types of objects needed to use in a distributed object context. This isn't much different from defining the public interfaces in a nondistributed application, with the following exceptions:

- Every object wanted to be distributed using RMI has to directly or indirectly extend an interface that extends the java.rmi.Remote interface.
- Every method in the remote interface has to declare that it throws a java.rmi.RemoteException or one of the parent classes of RemoteException.

RMI imposes the first requirement to allow it to differentiate quickly between objects that are enabled for remote distribution and those that are not. During a remote method invocation, the RMI runtime system needs to be able to determine whether each argument to the remote method is a Remote object or not. The Remote

interface, which is simply a tag interface that marks remote objects, makes it easy to perform this check.

The second requirement is needed to deal with errors that can happen during a remote session. When a client makes a method call on a remote object, any number of errors can occur, preventing the remote method call from completing. These include client–side errors (e.g., an argument can't be marshaled), errors during the transport of data between client and server (e.g., the network connection is dropped), and errors on the server side (e.g., the method throws a local exception that needs to be sent back to the remote caller). The RemoteException class is used by RMI as a base exception class for any of the different types of problems that might occur during a remote method call. Any method declared in a Remote interface is assumed to be remotely callable, so every method has to declare that it might throw a RemoteException, or one of its parent interfaces.

Figure 2.7 shows a simple remote interface that declares two methods: `doThis()` and `doThat()`. These methods could do anything that we want; in our `Account` example, we had remote methods to deposit, withdraw, and transfer funds. Each method takes a single `String` argument and returns a `String` result. Since we want to use this interface in an RMI setting, we've declared that the interface extends the `Remote` interface. In addition, each method is declared as throwing a `RemoteException`.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface ThisOrThatServer extends Remote {
       public String doThis(String todo) throws RemoteException;
       public String doThat(String todo) throws RemoteException;
}
```

Figure 2.7 : The ThisOrThatServer Interface

With the remote interface defined, the next thing we need to do is write a class that implements the interface. Figure 2.8 shows the `ThisOrThatServerImpl` class, which implements the `ThisOrThatServer` interface.

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
public class ThisOrThatServerImpl
extends UnicastRemoteObject implements ThisOrThatServer {
public ThisOrThatServerImpl() throws RemoteException {}
       // Remotely accessible methods
       public String doThis(String todo) throws RemoteException {
          return doSomething("this", todo);
       }
       public String doThat(String todo) throws RemoteException {
```

```
            return doSomething("that", todo);
        }
        // Non-remote methods
        private String doSomething(String what, String todo) {
            String result = "Did " + what + " to " + todo + ".";
            return result;
        }
}
```

Figure 2.8 : Implementation of the ThisOrThatServer

This class has implementations of the `doThis()` and `doThat()` methods declared in the `ThisOrThatServer` interface; it also has a nonremote method, `doSomething()`, that is used to implement the two remote methods. Notice that the `doSomething()` method doesn't have to be declared as throwing a `RemoteException`, since it isn't a remotely callable method. Only the methods declared in the remote interface can be invoked remotely. Any other methods you include in your implementation class are considered nonremote (i.e., they are only callable from within the local Java virtual machine where the object exists

2.5.1 Key RMI Classes for Remote Object Implementations

Our ThisOrThatServerImpl class also extends the UnicastRemoteObject class. This is a class in the java.rmi.server package that extends java.rmi.server.RemoteServer, which itself extends java.rmi.server.RemoteObject, the base class for all RMI remote objects. There are four key classes related to writing server object implementations:

RemoteObject

RemoteObject implements both the Remote and java.rmi.server package, it is used by both the Serializable interfaces. Although the RemoteObject class is in the client and server portions of a remote object reference. Both client stubs and server implementations are subclassed (directly or indirectly) from RemoteObject. A RemoteObject contains the remote reference for a particular remote object. RemoteObject is an abstract class that reimplements the equals(), hashCode(), and toString() methods inherited from Object in a way that makes sense and is practical for remote objects. The equals() method, for example, is implemented to return true if the internal remote references of the two RemoteObject objects are equal, (i.e., if they both point to the same server object).

RemoteServer

RemoteServer is an abstract class that extends RemoteObject. It defines a set of static

methods that are useful for implementing server objects in RMI, and it acts as a base class for classes that define various semantics for remote objects. In principle, a remote object can behave according to a simple point−to−point reference scheme; it can have replicated copies of itself scattered across the network that need to be kept synchronized; or any number of other scenarios. JDK 1.1 supported only point−to−point, nonpersistent remote references with the UnicastRemoteObject class. The Java 2 SDK 1.2 has introduced the RMI activation system, so it provides another subclass of RemoteServer, Activatable.

UnicastRemoteObject

This is a concrete subclass of RemoteServer that implements point−to−point remote references over TCP/IP networks. These references are nonpersistent: remote references to a server object are only valid during the lifetime of the server object. Before the server object is created (inside a virtual machine running on the host) or after the object has been destroyed, a client can't obtain remote references to the object. In addition, if the virtual machine containing the object exits (intentionally or otherwise), any existing remote references on clients become invalid and generate RemoteException objects if used.

Activatable

This concrete subclass of RemoteServer is part of the new RMI object activation facility in Java 1.2 and can be found in the java.rmi.activation package. It implements a server object that supports persistent remote references. If a remote method request is received on the server host for an Activatable object, and the target object is not executing at the time, the object can be started automatically by the RMI activation deamon.

2.6 Creating the Stubs and Skeletons

After you define the remote Java interface and implementation class, you compile them into Java bytecodes using a standard Java compiler. Then you use the RMI stub/skeleton compiler, *rmic*, to generate the stub and skeleton interfaces that are used at either end of the RMI communication link, as was shown in Figure 2.5. In its simplest form, you can run *rmic* with the fully qualified classname of your implementation class as the only argument. For example, once we've compiled the `ThisOrThatServer` and `ThisOrThatServerImpl` classes, we can generate

the stubs and skeletons for the remote `ThisOrThatServer` object with the following command (Unix version):

```
% rmic ThisOrThatServerImpl
```

If the RMI compiler is successful, this command generates the stub and skeleton classes,`ThisOrThatServerImpl_Stub` and `ThisOrThatServerImpl_Skel`, in the current directory. The *rmic* compiler has additional arguments that let you specify where the generated classes should be stored, whether to print warnings, etc. For example, if you want the stub and skeleton classes to reside in the directory

/usr/local/classes, you can run the command using the `-d` option:

```
% rmic -d /usr/local/classes ThisOrThatServerImpl
```

This command generates the stub and skeleton classes in the specified directory.

2.7 Accessing Remote Objects as a Client

Now that we've defined a remote object interface and its server implementation and generated the stub and skeleton classes that RMI uses to establish the link between the server object and the remote client, it's time to look at how you make your remote objects available to remote clients

2.7.1 The Registry and Naming Services

The first remote object reference in an RMI distributed application is typically obtained through the RMI registry facility and the `Naming` interface. Every host that wants to export remote references to local Java objects must be running an RMI registry daemon of some kind. A registry daemon listens (on a particular port) for requests from remote clients for references to objects served on that host. The standard Sun Java SDK distribution provides an RMI registry daemon, *rmiregistry*. This utility simply creates a `Registry` object that listens to a specified port and then goes into a wait loop, waiting for local processes to register objects with it or for clients to connect and look up RMI objects in its registry. You start the registry daemon by running the *rmiregistry* command, with an optional argument that specifies a port to listen to:

```
objhost% rmiregistry 5000 &
```

Without the port argument, the RMI registry daemon listens on port 1099. Typically, you run the registry daemon in the background (i.e., put an `&` at the end of the command on a Unix system or run `start rmiregistry [`*port*`]` in a DOS window on a Windows system) or run it as a service at startup.

Once the RMI registry is running on a host, you can register remote objects with it using one of these classes: the `java.rmi.registry.Registry` interface, the `java.rmi.registry.LocateRegistry` class, or the `java.rmi.Naming` class.

A `Registry` object represents an interface to a local or remote RMI object registry. The `bind()` and `rebind()` methods can register an object with a name in the local registry, where the name for an object can be any unique string. If you try to `bind()` an object to a name that has already been used, the registry throws an `AlreadyBoundException`. If you think that an object may already be bound to the name you want to register, use the `rebind()` method instead. You can remove an object binding using the `unbind()` method. Note that these three methods (`bind()`, `rebind()`, and `unbind()`) can be called only by clients running on the same host as the registry. If a remote client attempts to call these methods, the client receives a `java.rmi.AccessException`. You can locate a particular object in the registry using the `lookup()` method, while `list()` returns the names of all the objects registered with the local registry. Note that only `Remote` objects can be bound to names in the `Registry`. `Remote` objects are capable of supporting remote references. Standard Java classes are not, so they can't be exported to remote clients through the `Registry`

The `LocateRegistry` class provides a set of static methods a client can use to get references to local and remote registries, in the form of `Registry` objects. There are four versions of the static `getRegistry()` method, so that you can get a reference to either a local registry or a remote registry running on a particular host, listening to either the default port (1099) or a specified port. There's also a static `createRegistry()` method that takes a port number as an argument. This method starts a registry running within the current Java VM on the given local port and returns the `Registry` object it creates.

Using the `LocateRegistry` and `Registry` interfaces, we can register one of our `ThisOrThatServerImpl` remote objects on the local host with the following code:

```
ThisOrThatServerImpl server = new ThisOrThatServerImpl();
Registry localRegistry = LocateRegistry.getRegistry();
try {
      localRegistry.bind("TTServer", server);
}
catch (RemoteException re) { // Handle failed remote operation }
catch (AlreadyBoundException abe) { // Already one there }
catch (AccessException ae) { // Shouldn't happen, but... }
```

If this operation is successful (i.e., it doesn't raise any exceptions), the local registry has a `ThisOrThatServerImpl` remote object registered under the name "TTServer." Remote clients can now look up the object using a combination of the `LocateRegistry` and `Registry` interfaces, or take the simpler approach and use the `Naming` class.

The `Naming` class lets a client look up local and remote objects using a URL–like naming syntax. The URL of a registered RMI remote object is typically in the format shown in Figure 2.9. Notice that the only required element of the URL is the actual object name. The protocol defaults to *rmi:*, the hostname defaults to the local host, and the port number defaults to 1099. Note that the default `Naming` class provided with Sun's Java SDK accepts only the *rmi:* protocol on object URLs. If you attempt to use any other protocol, a `java.net.MalformedURLException` is thrown by the `lookup()` method.

If we have a client running on a remote host that wants to look up the `ThisOrThatServerImpl` we registered, and the `ThisOrThatServerImpl` object is running on a host named rmiremote.objhost.org, the client can get a remote reference to the object with one line of code:

```
    ThisOrThatServer rmtServer =
(ThisOrThatServer)Naming.lookup("rmi://rmiremote.objhost.org/TTServer");
```

If we have a client running on the same host as the `ThisOrThatServerImpl` object, the remote reference can be retrieved using the degenerate URL:

```
ThisOrThatServer rmtServer = (ThisOrThatServer)Naming.lookup("TTServer");
```

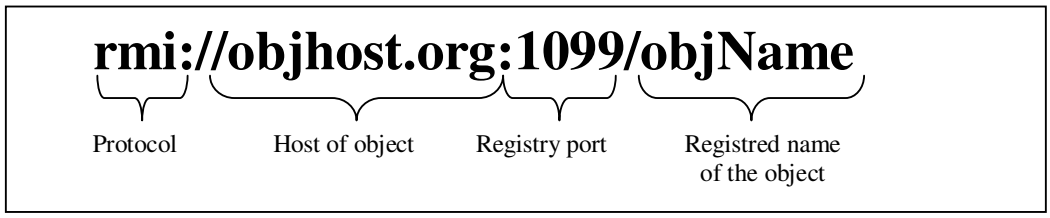

Figure 2.9 : Anatomy of an RMI object URL

Alternately, you can use the `LocateRegistry` and `Registry` interfaces to look up the same object, using an extra line of code to find the remote `Registry` through the `LocateRegistry` interface:

```
Registry rmtRegistry =LocateRegistry.getRegistry("rmiremote.objhost.org");
ThisOrThatServer rmtServer =
                        (ThisOrThatServer)rmtRegistry.lookup("TTServer");
```

When you look up objects through an actual `Registry` object, you don't have the option of using the URL syntax for the name, because you don't need it. The hostname and port of the remote host are specified when you locate the `Registry` through the `LocateRegistry` interface, and the RMI protocol is implied, so all you need is the registered name of the object. With the `Naming` class, you can reduce a remote object lookup to a single method call, but the name must now include the host, port number, and registered object name, bundled into a URL. Internally, the `Naming` object parses the host and port number from the URL for you, finds the remote `Registry` using the `LocateRegistry` interface, and asks the `Registry` for the remote object using the object name in the URL.

The principal use for the `Registry` and `Naming` classes in an RMI application is as a means to bootstrap your distributed application. A server process typically exports just a few key objects through its local RMI registry daemon. Clients look up these objects through the `Naming` facility to get remote references to them. Any other remote objects that need to be shared between the two processes can be exported through remote method calls.

3. EXECUTION FRAMEWORK IMPLEMENTATION

The asynchronous RMI execution model has been implemented by an execution framework (Figure 3.1) which is developed in Java. It consists of a Java package itu.rmi.* containing the classes that provide the basic services for asynchronous invocations. The classes that are visible to the client are depicted in Figure 4.1. The execution flow of an asynchronous invocation (arrows 1 through 7 in 0) and its implementation details are described below, assuming that a server has already registered a remote object with an RMI registry, making it available to remote clients.

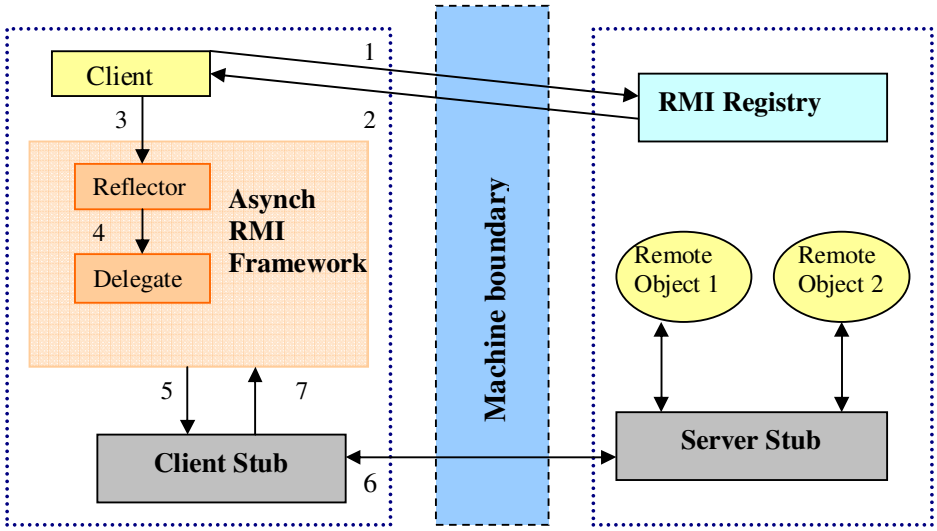

Figure 3.1 : Asynchronous RMI Framework Execution Flow

Initialization Phase:

1. This step involves the determination of the remote host. The client queries the lookup service *rmi registry* to retrieve a remote reference to the target remote object.

2. The lookup call returns a remote reference and results in the placement of a client stub which provides the interface to interact with the remote object.

Asynchronous Invocation Request:

3. An asynchronous invocation requires an instance of an invocation object to be created for the specific type of asynchronous call *(FireandForget, SyncWithServer, Polling, Callback)* the client wishes to make. The asynchronous call is dispatched as the client invokes the public void call(Object remoteObject, String methodName, Object params[]) overloaded call method of the invocation object with the actual parameters that specify the client stub reference, the name and the list of parameters of the remote method to be called. The client resumes execution as soon as the call method returns.

Asynchronous Invocation Processing:

4. The call method receives an invocation request and calls the appropriate method on the remote object transparently, applying the semantics of the specific type of asynchronous call. For this purpose, it uses reflection to assemble a method call during runtime. Once the method name and the argument types are resolved, they are matched with those of the incoming request to detect errors such as invoking a non-existing method, passing an incorrect number of arguments, or passing incorrect argument types to a method, raising exceptions that are caught by client. If no error exists, the call method returns after starting a service thread which handles the remainder of the invocation, allowing the client to resume execution while the asynchronous call proceeds in the new service thread. The service thread, called a *delegate* in the our framework, is activated from a thread pool with parameters that include invocation details such as the remote object, method name, and parameters.

5. The delegate simply executes an invoke method call in its run method, (_remoteMethod.invoke (_remoteObject,_parameters);) which performs a standard RMI for the requested method over the client stub.

6. This phase involves standard RMI activity following its parameter passing semantics. The server stub passes the call request with its parameters to the

server stub, which in turn, executes the method call on the remote object and returns the results back to the client stub over the network.

7. The client stub returns the result of the remote call to the *delegate*. The delegate responds differently on retrieving the result, according to the specific type of asynchronous call.

Life cycle of a sample asynchronous call is shown in Figure 3.2,

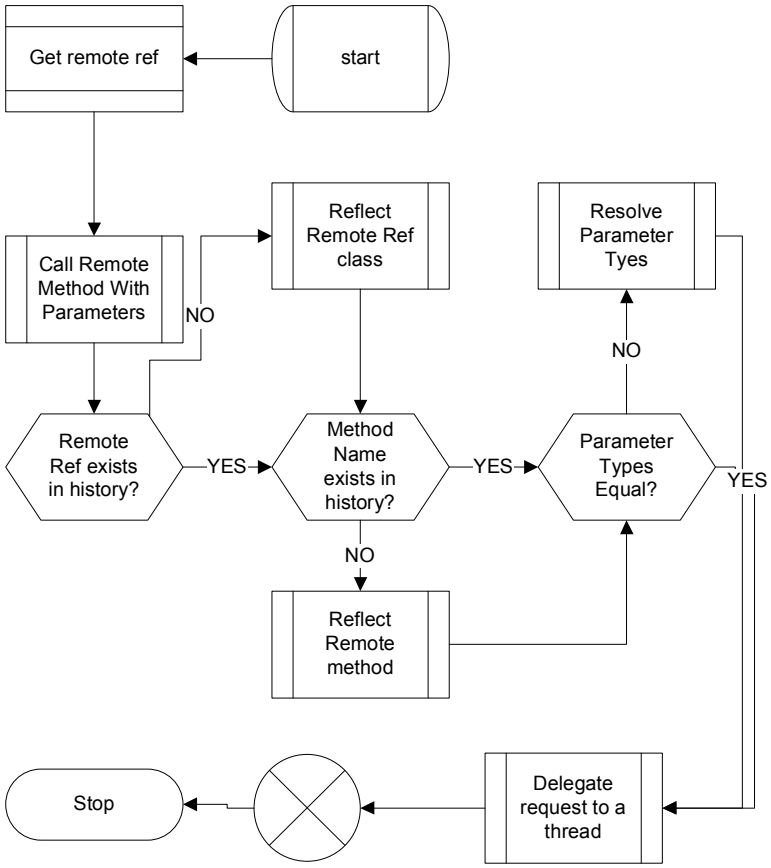

Figure 3.2 : Asynchronous call life cycle

4. TYPES OF ASYNCHRONOUS CALLS

The execution framework supports the four most commonly used asynchronous calls. In the following sections, we describe their execution patterns [3] and give code samples to show how to utilize the framework for such calls, assuming the existence of the remote server objects logger, docConverter, searchEngine, and Emergency on a host with IP '192.168.1.3'.

Public classes of the asynchronous execution framework which are visible to user is depicted in 0.

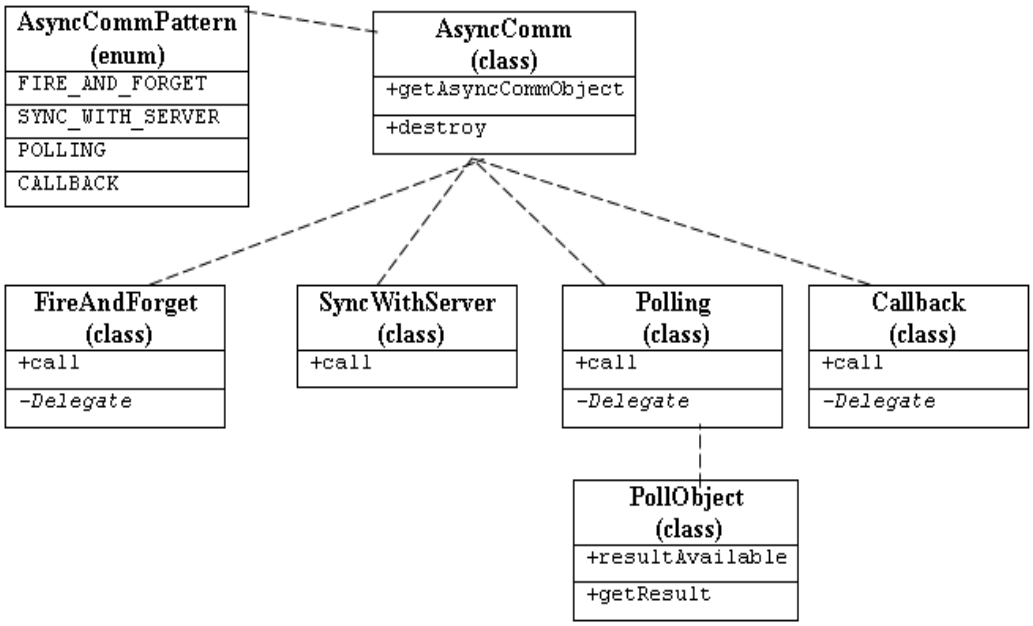

Figure 4.1 : Execution Framework Public Classes

Implementation details of the asynchronous communication patterns will be given using an example scenario for each patter, which are Remote Logger for 'Fire and Forget' , Remote Document Converter for 'Sync with Server', Remote Search Engine for 'Polling' and Remote Emergency Alarm for 'Callback'.

Remote Logger: It is a remote java object registered in the host's registry which is responsible for writing logs to a text file in its hosts file system.

Remote Document Converter: It is a remote java object which accepts a microsoft word document, upon to retrieval returns an ACK to its client, and silently converts received document to PDF format.

Remote Search Engine: This remote java object gets a string keyword and searchs for it on the internet when the search is completed a list of results is sent to client.

Remote Emergency Alarm: This remote java object has to be configured first by the client with criterion, when the criterion is matched result is returned to the client indicating if the emergency is sent to all peers or not.

4.1 Fire and Forget

Fire and forget is well suited for applications where a remote object needs to be notified and a result or a return value is not required. Reliability is not a concern for both sides. When the client issues such an invocation request, the call returns back to the client immediately. The client does not get any acknowledgment from the remote object receiving the invocation. Figure 4.2 shows 'fire and forget' asynchronous communication pattern.

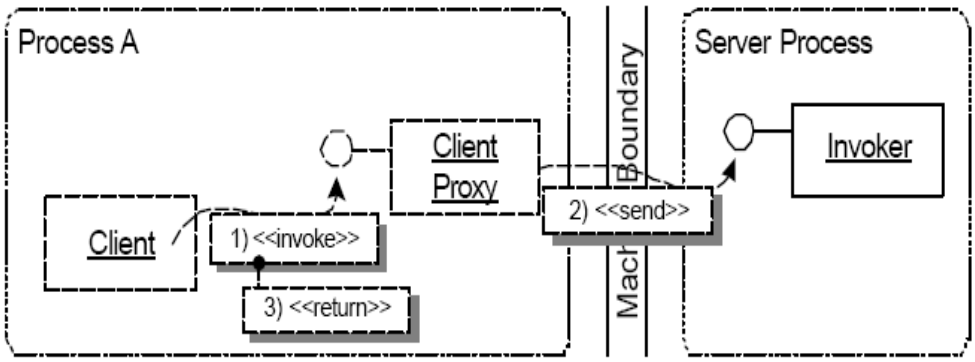

Figure 4.2 : Fire and Forget Communication Pattern

The client invokes asynchronously the log method of the remote object logger with the string parameter "my log message". The call method returns right after dispatching the delegate with the call request. Thus, the client thread and the delegate thread run concurrently, the client returning back to its execution while the delegate blocked waiting for the call to return. In Figure 4.3 usage of the pattern within our asynchronous RMI framework is shown.

```
import itu.rmi.*;
import java.rmi.*;
public class TestAsyncRMI {
public static void main(String[] args) {
  // use standard rmi to get remote reference
  Logger logger = (Logger)Naming.lookup("rmi://192.168.1.3/logger");
  // get a 'fire and forget' invocation object
  FireAndForget fireAndForget = (FireAndForget)
    AsyncComm.getAsyncCommObject(AsyncCommPattern.FIRE_AND_FORGET);
  // asynch call over  invocation object through call method
  fireAndForget.call(logger, "log", new Object[] {"my log
      message"});
  // client continues  execution immediately          after call returns
}
```

Figure 4.3 : Fire and Forget pattern with asynchronous RMI

4.2 Sync with Server

Sync with Server is similar to fire and forget; however, it ensures that the invocation has been performed reliably. Again, a remote object needs to be notified and a result of the remote computation is not required. The difference is that, the call does not immediately return to the client but waits for an acknowledgement from the remote object to ensure that the request has been successfully transferred to the server application. Only then, control passes to the client. Meanwhile, the server application independently executes the invocation. Figure 4.4 shows 'sync with server' asynchronous communication pattern.

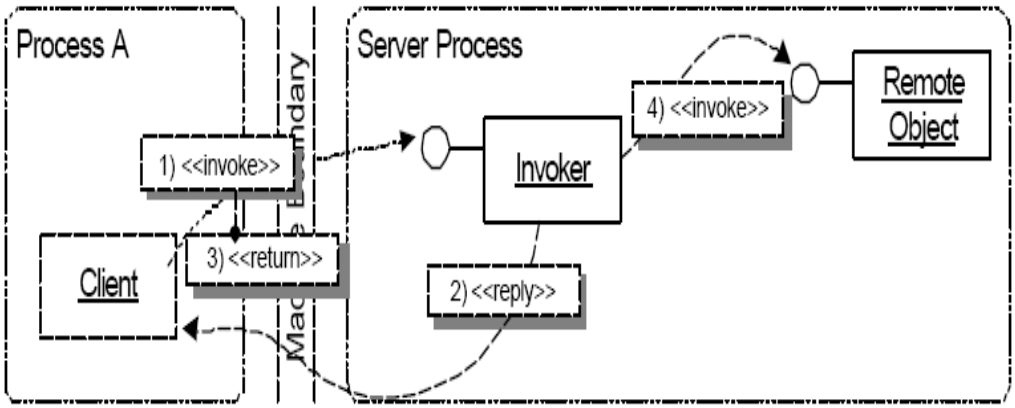

Figure 4.4 : Sync with Server Communication Pattern

The client invokes asynchronously the convert method of the remote object docConverter with the input parameter msDoc. The call method blocks until the server application docConverter returns an acknowledgement that it has successfully received the request. From that point on, both the client and the server in parallel, the client returning back to its execution while the docConverter proceeds with the document conversion process. In Figure 4.5 usage of the pattern within our asynchronous RMI framework is shown.

```
import itu.rmi.*;
import java.rmi.*;
public class TestAsyncRMI {
public static void main(String[] args) {
  DocConverter docConverter = (DocConverter)
                    Naming.lookup("rmi://192.168.1.3/docConverter");
  SyncWithServer syncWithServer = (SyncWithServer)
    AsyncComm.getAsyncCommObject(AsyncCommPattern.SYNC_WITH_SERVER);
  MSDoc msDoc = new MSDoc();
  // asynch call over  invocation object through call method, blocking until server returns an ack.
boolean b = ((Boolean)syncWithServer.call(docConverter,
"convert",new Object[] { msDoc })).booleanValue();
  //continue execution while server object converts the document
}
```

Figure 4.5: Sync with Server pattern with asynchronous RMI

4.3 Polling

Polling is suitable for applications where a remote object needs to be invoked asynchronously, and yet, a result is required. However, as the results may not be needed immediately for the client to proceed with its computation, the client may continue with its execution and retrieve the results later. In such a case, a poll object receives the result of remote invocations on behalf of the client. The client, at an appropriate point in its execution path, uses this object to query the result and obtain it. It may poll the object and continue with its computation if the result has not yet arrived, or it may block on the object until the result becomes available. Figure 4.6 shows 'polling' asynchronous communication pattern.

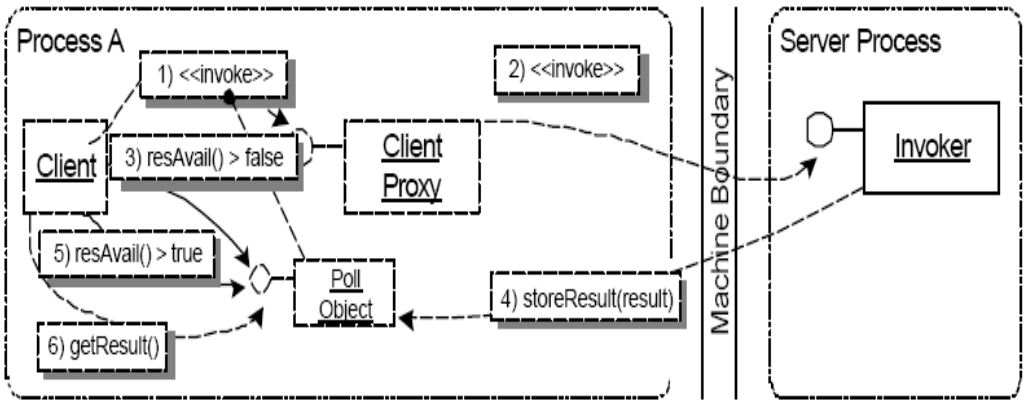

Figure 4.6 : Polling Communication Pattern

The client issues a polling call to the search method with the keyword "java" as the parameter. This time, the method call returns with a poll object as soon as it dispatches the delegate with the call request, allowing the client to continue with its execution while remote object processes the query and produces a result. When a result is returned, the delegate thread retrieves it in the poll object, sets a flag in the object to indicate that it is available and notifies any thread blocked on the object. The resultAvailable() method of the poll object returns a boolean value and may be checked by the client to see if a result has arrived. The client may also call the getResult()method of the poll object to get the result. However, this is a blocking call and does not return until the result actually becomes available. In Figure 4.7 usage of the pattern within our asynchronous RMI framework is shown.

```
import itu.rmi.*;
import java.rmi.*;
public class TestAsyncRMI {
public static void main(String[] args) {
  SearchEngine searchEngine = (SearchEngine)
  Naming.lookup("rmi://192.168.1.3/searchEngine");
  Polling polling = (Polling)
 AsyncComm.getAsyncCommObject(AsyncCommPattern.POLLING);
  // asynch call over invocation object through call method- the call returns a poll object
  PollObject pollObject = (PollObject)polling.call(searchEngine,
                                "search", new Object[] {"java"});
  // query to find out if the result of the asynchronous call is available
  boolean b = pollObject.resultAvailable();
  // blocking call that returns the result if it is available
  // or blocks the client until it becomes available and resumes it with the result
  List<String> result = (List<String>)pollObject.getResult();
}
```

Figure 4.7 : Polling pattern with asynchronous RMI

4.4 Callback

Similar to Polling, a remote operation needs to be invoked asynchronously and a result is required. However, in this case, the client needs to react to the result immediately it becomes available, not at a future time of its choice. Therefore, the client requests to be actively notified of the returning result. To this end, the client passes a callback object together with the invocation request to the execution framework. The call returns after sending the invocation to the server object and the client resumes execution. Once the result becomes available, a predefined operation on the callback object is called, passing it the result of the invocation. Figure 4.8 shows 'callback' asynchronous communication pattern.

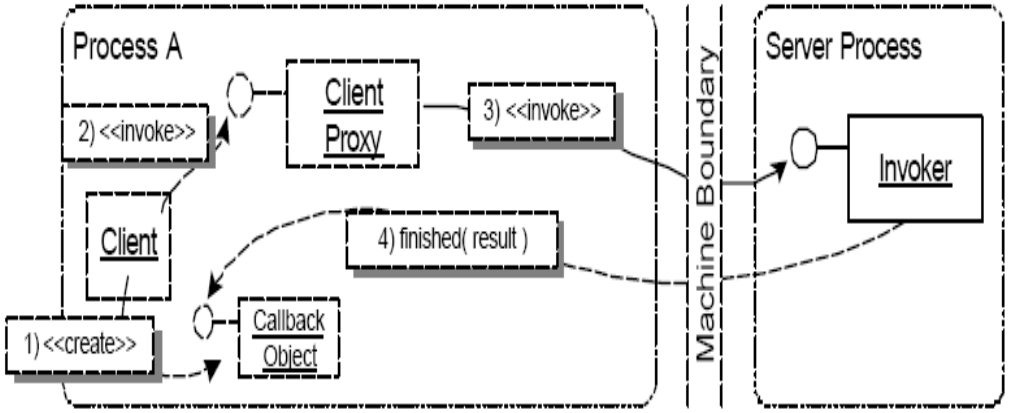

Figure 4.8 : Callback Communication Pattern

Our implementation makes use of the Observer Pattern [5], therefore, callbackObject, an instance of a class which implements Observer interface is created before the client makes a call to the sendEmergency method, supplying a criterion "pressure is below 10" and a callback object callbackObject. Once the result of the remote call becomes, the update method of the observer object is invoked automatically with result parameters by the delegate using built-in method notifyObservers. In Figure 4.9 usage of the pattern within our asynchronous RMI framework is shown.

```
import itu.rmi.*;
import java.rmi.*;

class CallbackObject implements Observer{
  public void update(Observable o, Object arg){
      // arg is the  result returned by the  remote object
  }
}
public class TestAsyncRMI {
public static void main(String[] args) {
  Emergency emergency = (Emergency)
                       Naming.lookup("rmi://192.168.1.3/emergency");
  Callback callback = (Callback)
            AsyncComm.getAsyncCommObject(AsyncCommPattern.CALLBACK);
  // create  callback object
  CallbackObject callbackObject = new CallbackObject();
 //asynch call over  invocation object through call method, callback object also passed as a param
  callback.call(emergency, "sendEmergency", new Object[]
                          {"pressure is below 10"}, callbackObject);
}
```

Figure 4.9 : Callback pattern with asynchronous RMI

5. PERFORMANCE OPTIMIZATION

Performance of the framework has been a central concern during implementation. We tried to determine the greatest sources of runtime overhead and observed them to be related to threading facility and reflection mechanism of Java. Below, we describe the optimizations we have done.

5.1 Thread Pooling

Programs that are amenable to threading are generally broken down into thread-sized pieces using either functional decomposition or data decomposition. Functional decomposition assigns a thread to each distinct task. (For example, in a word processor, assigning the task of real-time spell checking to a thread would be an instance of functional decomposition.) Data decomposition takes a data-oriented task and breaks it into smaller chunks of data and gives one chunk to each of several threads to process in parallel. For example, processing large data arrays use this technique. Each thread, let's say, processes one quarter of the array—with all four threads running in parallel on different processor cores.

Many programs use both types of decomposition. They assign specific tasks to identified threads and then create a bunch of threads to handle data processing. Managing this bunch of threads has certain challenges: creating threads is expensive, so you want to create the minimum needed for optimal performance. Moreover, when the threads are finished with their initial assignment, you'd rather they not die off (the default behavior) but stay alive to handle upcoming work. In addition, you need a mechanism that doles out the work to the available threads.

Writing such a mechanism yourself is certainly possible, but it would require a lot of delicate code. Fortunately, Java provides us with a thread pool, which is the instantiation of just this concept: a bunch of threads, a manager that keeps them alive when they’re done working, and a queue that feeds them tasks. The Java thread pool

APIs were greatly improved with the Java 5.0 release, which provides some easy to use high-level constructs

In most other cases, however, you will want to call the API Executors.newCachedThreadPool(), which creates a thread pool of an indeterminate size. The number of threads is enough to handle the initial tasks handed to the pool; after which, the threads are reused. But for this one change, the remaining code in this example would remain the same.

The preference for not hard-coding the thread count is consistent with a principle that appears time and again in parallel programming: the more you trust the operating system or runtime framework to manage threads, the better off (and more portable) your code will be. This is due to the fact that operating systems are very finely tuned when it comes to thread scheduling, so the less you limit their operation with explicit constraints, the faster your code will run, and the less you'll have to keep tweaking it for different runtime environments

The framework transfers each new asynchronous invocation request to a new thread. Considering the overhead of spawning a new thread on each call, we optimized the interaction with the thread package and switched to using a ThreadPool which creates new threads as needed, but reuses previously created threads when they are available.

We use Executors class of Java to create thread pool, here is how our thread pool is created for the asynchronous RMI execution framework.

```
static ExecutorService ThreadPool = Executors.newCachedThreadPool();
```

Once we have the static thread pool, whenever we need to spawn a new thread, thread pool is used as following;

```
AsyncComm.ThreadPool.execute(new Delegate(remoteObject,
                                          _remoteMethod, params));
```

Table 5.1 shows te performance gain for sequential calls of creating a thread for simple int add(int, int) operation.

Table 5.1: Sequantial call speed with threading

# Calls	No Thread Pool(ms)	Thread Pool(ms)
1	0	0
10	0	0
100	0	0
1000	63	31
10000	531	94
100000	5203	687
1000000	52641	8703
10000000	527172	119047

Figure 5.1. displays the performance gain in invocation response time, which becomes evident especially with high number of simultaneous requests.

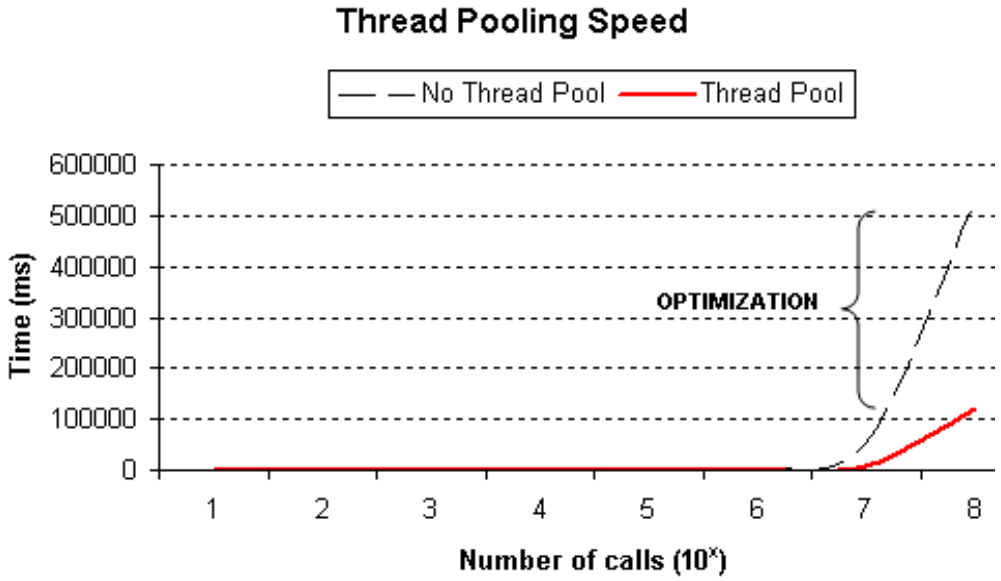

Figure 5.1 : Thread Pooling Speedup

Note: When we use Executors thread pool, we should shutdown the thread pool in order to exit from our calling program, if we don't, thread pool will be alive with background threads. Our Asynchronous RMI Framework provides a static destroy method to be called when we are done with the framework.

5.2 Reflection

Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of

the program. For example, it's possible for a Java class to obtain the names of all its members and display them.

The ability to examine and manipulate a Java class from within itself may not sound like very much, but in other programming languages this feature simply doesn't exist. For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program.

Using reflection is different from normal Java programming in that it works with *metadata* -- data that describes other data [30]. The particular type of metadata accessed by Java language reflection is the description of classes and objects within the JVM. Reflection gives you run-time access to a variety of class information. It even lets you read and write fields and call methods of a class selected at run time.

Reflection is a powerful tool. It lets you build flexible code that can be assembled at run time without requiring source code links between components. But some aspects of reflection can be problematic.

Beginners' class

The starting point for using reflection is always a `java.lang.Class` instance. If you want to work with a predetermined class, the Java language provides an easy shortcut to get the `Class` instance directly:

```
Class clas = MyClass.class;
```

When you use this technique, all the work involved in loading the class takes place behind the scenes. If you need to read the class name at run time from some external source, however, this approach isn't going to work. Instead, you need to use a class loader to find the class information. Here's one way to do that:

```
// "name" is the class name to load
Class clas = null;
try {
  clas = Class.forName(name);
} catch (ClassNotFoundException ex) {
  // handle exception case
}
// use the loaded class
```

If the class has already been loaded, you'll get back the existing `Class` information. If the class hasn't been loaded yet, the class loader will load it now and return the newly constructed class instance.

Reflections on a class

The `Class` object gives you all the basic hooks for reflection access to the class metadata. This metadata includes information about the class itself, such as the package and superclass of the class, as well as the interfaces implemented by the class. It also includes details of the constructors, fields, and methods defined by the class. These last items are the ones most often used in programming, so I'll give some examples of working with them later in this section.

For each of these three types of class components -- constructors, fields, and methods -- the `java.lang.Class` provides four separate reflection calls to access information in different ways. The calls all follow a standard form. Here's the set used to find constructors:

- Constructor getConstructor(Class[] params) -- Gets the public constructor using the specified parameter types
- Constructor[] getConstructors() -- Gets all the public constructors for the class
- Constructor getDeclaredConstructor(Class[] params) -- Gets the constructor (regardless of access level) using the specified parameter types
- Constructor[] getDeclaredConstructors() -- Gets all the constructors (regardless of access level) for the class

Each of these calls returns one or more java.lang.reflect.Constructor instances. This Constructor class defines a newInstance method that takes an array of objects as its only argument, then returns a newly constructed instance of the original class. The array of objects are the parameter values used for the constructor call. As an example of how this works, suppose you have a TwoString class with a constructor that takes a pair of Strings, as shown in Figure 2.5:

```
Public class TwoString {
    private String m_s1, m_s2;
    public TwoString(String s1, String s2) {
        m_s1 = s1;
        m_s2 = s2;
    }
}
```

Figure 5.2 : Class constructed from pair of strings

The code shown in Figure 5.3 gets the constructor and uses it to create an instance of the `TwoString` class using `String`s `"a"` and `"b"`:

```
Class[] types = new Class[] { String.class, String.class };
Constructor cons = TwoString.class.getConstructor(types);
Object[] args = new Object[] { "a", "b" };
TwoString ts = (TwoString)cons.newInstance(args);
```

Figure 5.3 : Reflection call to constructor

The code in Listing 2 ignores several possible types of checked exceptions thrown by the various reflection methods. The exceptions are detailed in the Javadoc API descriptions, so in the interest of conciseness, I'm leaving them out of all the code examples.

While I'm on the topic of constructors, the Java programming language also defines a special shortcut method you can use to create an instance of a class with a no-argument (or default) constructor. The shortcut is embedded into the Class definition itself like this:

Object newInstance() -- Constructs new instance using default constructor
Even though this approach only lets you use one particular constructor, it makes a very convenient shortcut if that's the one you want. This technique is especially useful when working with JavaBeans, which are required to define a public, no-argument constructor.

Fields by reflection

The Class reflection calls to access field information are similar to those used to access constructors, with a field name used in place of an array of parameter types:

- Field getField(String name) -- Gets the named public field
- Field[] getFields() -- Gets all public fields of the class
- Field getDeclaredField(String name) -- Gets the named field declared by the class
- Field[] getDeclaredFields() -- Gets all the fields declared by the class

Despite the similarity to the constructor calls, there's one important difference when it comes to fields: the first two variants return information for public fields that can be accessed through the class -- even those inherited from an ancestor class. The last

two return information for fields declared directly by the class -- regardless of the fields' access types.

The java.lang.reflect.Field instances returned by the calls define getXXX and setXXX methods for all the primitive types, as well as generic get and set methods that work with object references. It's up to you to use an appropriate method based on the actual field type, though the getXXX methods will handle widening conversions automatically (such as using the getInt method to retrieve a byte value).

Figure 5.4 shows an example of using the field reflection methods, in the form of a method to increment an int field of an object by name:

```
public int incrementField(String name, Object obj) throws... {
    Field field = obj.getClass().getDeclaredField(name);
    int value = field.getInt(obj) + 1;
    field.setInt(obj, value);
    return value;
}
```

Figure 5.4: Incrementing a field by reflection

This method starts to show some of the flexibility possible with reflection. Rather than working with a specific class, incrementField uses the getClass method of the passed-in object to find the class information, then finds the named field directly in that class.

Methods by reflection

The Class reflection calls to access method information are very similar to those used for constructors and fields:

- Method getMethod(String name, Class[] params) -- Gets the named public method using the specified parameter types
- Method[] getMethods() -- Gets all public methods of the class
- Method getDeclaredMethod(String name, Class[] params) -- Gets the named method declared by the class using the specified parameter types
- Method[] getDeclaredMethods() -- Gets all the methods declared by the class

As with the field calls, the first two variants return information for public methods that can be accessed through the class -- even those inherited from an ancestor class.

The last two return information for methods declared directly by the class, without regard to the access type of the method.

The java.lang.reflect.Method instances returned by the calls define an invoke method you can use to call the method on an instance of the defining class. This invoke method takes two arguments, which supply the class instance and an array of parameter values for the call.

5.2.1 Reflection Optimization in Asynchronous RMI Framework

The second optimization we performed concerns reflection. When the call method of an invocation object receives an invocation request, it uses reflection to resolve the method name, the number of its arguments and their types. To minimize the overhead introduced by reflection, we cache the recently resolved information in private fields of the invocation object so that successive requests for a particular invocation instantly use the local data, instead of reflection lookups.

Here is how our caching performed in the 'call' method of any invocation object

```
if(_remoteObject != remoteObject) {
   _remoteObject = remoteObject;
   _remoteClass = remoteObject.getClass();
}
if(_methodName.compareTo(methodName) != 0  ||
           !Arrays.equals(_parameterTypes, Types.GetTypes(params))){
   _methodName = methodName;
   _parameterTypes = Types.GetTypes(params);
_remoteMethod = _remoteClass.getMethod(methodName, _parameterTypes);
}
```

Table 5.2 show the details of normal reflection, reflection without lookup (caching), and direct call to a simple method int add(int, int).

Table 5.2: Sequential call performance with diffrent reflection approaches

# Calls	Lookup(ms)	No Lookup(ms)	Direct Call(ms)
1	0	0	0
10	0	0	0
100	0	0	0
1000	0	0	0
10000	16	0	0
100000	125	16	0
1000000	1109	94	16
10000000	11079	1047	62
100000000	110968	10328	672
1000000000	1110032	102312	6719

Figure 5.5. displays speed up gained through reflection optimization. Comparing three approaches direct method call, method call using reflection with lookup, and method call with reflection without lookup.

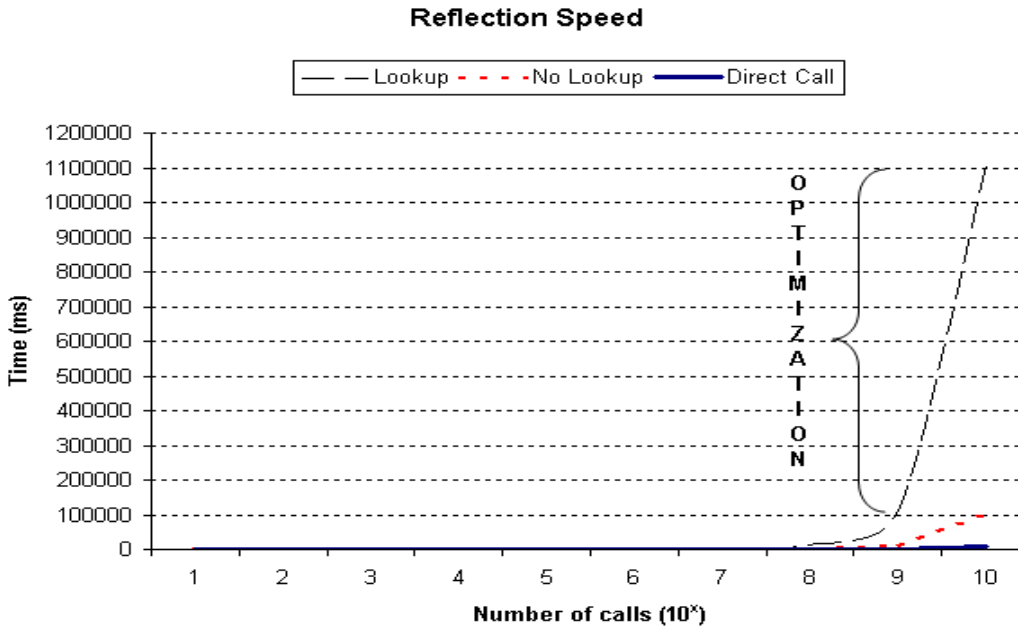

Figure 5.5: Reflection speedup

6. MEASUREMENT RESULTS

We have conducted experiments in order to measure the performance of the framework and compare it with Java RMI. The test computer is Intel Core 2 Duo CPU T8300 2.4 GHz, 2GB of RAM, Windows XP Professional OS, Java build 1.6.0_11-b0. and the remote computer is Intel Pentium 4 Mobile CPU 1.70GHz, 512 MB of RAM. Java build 1.6.0_11-b03, Windows XP Professional OS.

Figure 6.1. displays the execution cost for an asynchronous invocation of a remote method with the signature 'int add(int lhs, int rhs)' that simply adds the two input parameters and returns the result. For each invocation type, five test runs were executed and their average is reported. *Call Time* is the elapsed time for the client to make the call and resume execution. *Result Retrieval Time* is the time it takes for the result to be available. For polling type of invocation, this is the time duration until the blocking call getResult returns. For Callback, it is the duration until the update method of the observer object gets invoked.

Table 6.1: Asynchronous invocation execution costs

	Fire and Forget	Sync With Server	Polling	Callback
Call Time (ns)	3901948	11767026	3126207	2879528
Result Retrieval Time (ns)	NaN	11767026	4229923	4486045

We have implemented the remote objects introduced in Section 4 to demonstrate the different types of asynchronous calls and executed both standard RMI and asynchronous RMI calls to compare their execution time values. We assume that it takes 400 ms for the remote search engine to find a given keyword and a criterion is met every 2200 ms at the remote emergency object. Figure 6.1 illustrates the results, where the idle waiting state of the client issuing a standard RMI is clearly seen.

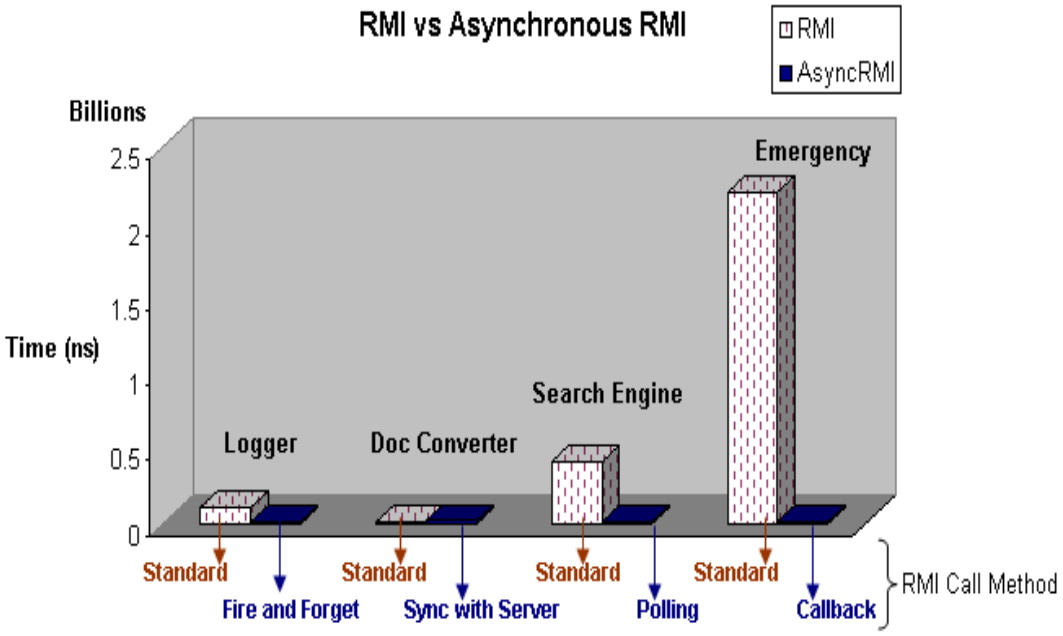

Figure 6.1 : Standard RMI versus Asynchronous RMI

7. CONCLUSION

This work presents an execution framework which extends Java RMI to support asynchronous communication, mainly focusing on the techniques of fire and forget, sync with server, polling, and result callback. We have used RMI as the underlying communication mechanism and implemented asynchrony patterns on top of RMI calls. The threading facility and reflection mechanism of Java has made it possible for the framework to be independent of any external tools, preprocessors, or compilers. Also, as the framework requires no modifications on the server side, therefore previously developed remote objects can still be accessed asynchronously. The results of performance measurements show that, with optimizations on threading and reflection activity, the enhanced asynchronous RMI communication we have described in this work produces a dramatic performance increase by removing unnecessary delays caused by blocking on synchronous RMI invocations.

We recommend our framework to be used in java projects where RMI is widely used, considering long running remote operations, if there exists a java distributed object system where distributed or parallel programming is used with RMI, our asynchronous RMI framework would increase the trough-put of the client programs.

We do not recommend it to be used for simple remote method calls, in such a case there will not be much performance gain but performance lost. As described before even use optimized reflection techniques in our framework it still causes a little bit performance lost comparing to standard RMI calls, so in such cases it is recommended to use standard RMI calls which are synchronous rather that our asynchronous RMI framework.

.

REFERENCES

[1] Sun Microsystems, 2004: Inc. Java (TM) Remote Method Invocation Specification.

[2] Sun Microsystems, 2004: Inc. Java (TM) Object Serialization Specification.

[3] Voelter M., Kircher M., Zdun U., Englbrecht M., Patterns for Asynchronous Invocations in Distributed Object Frameworks

[4] Glass G., 1997: 'Voyager: the new face of distributed computing', Object Magazine,URL, http://www.sigs.com/publications/docs/objm/9706/9706.glass.html.

[5] Rajeev R. R., Joseph I. W. and Boyles M., 1997: Asynchronous Remote Method Invocation (ARMI) mechanism for Java.

[6] Coulouris G., Dollimore J. and Kindberg T., 1994: Distributed Systems – Concepts and Design, Addison Wesley.

[7] Davison A., Drake K., Roberts W. and Slater M., 1992: DistributedWindow Systems, a Practical Guide to X11 and OpenWindows. Addison Wesley, Workingham.

[8] Liskov B. and Shrira L., 1998: 'Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems', in Proceedings of SIGPLAN'88 Conference Programming Language Design and Implementation.

[9] Foster I., Kesselman K. and Tuecke S., 1994: 'The Nexus task-parallel runtime system', Proceedings of 1st International Workshop on Parallel Processing.

[10] Foster I. and Tuecke S., 1997: 'Enabling technologies for Web-based ubiquitous supercomputing', to appear in Proceedings of 5th IEEE Symposium in High Performance Distributed Computing, URL, http://www.mcs.anl.gov/nexus/nexusjava.html.

[11] Maassen J., Nieuwpoort R. V., Veldema R., Bal H. E., and Plaat A., 1999: An Efficient Implementation of Java's Remote Method Invocation. In Proc. ACM Symposium on Principles and Practice ofPar allel Programming, May.

[12] Nester C., Philippsen M., and Haumacher B., 1999: A More Efficient RMI for Java. In Proc. of ACM 1999 Java Grande Conference, pages 152–157, June.

[13] Sampemane G., Rivera L., Zhang L., and Krishnamurthy S., 1999: HP-RMI : High Performance Java RMI over FM. Available from http://www-csag.ucsd.edu/individual/achien/cs491-f97/projects/hprmi.html.

[14] Pakin S., Lauria M., and Chien A., 1995: High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In Supercomputing '95, December.

[15] Pakin S., Karamcheti V., and Chien A. A., 1997: Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. IEEE Concurrency, 5(2), April/June.

[16] Culler D., Dusseau A., Goldstein S., Krishnamurthy A., Lumetta S., Eicken T. V., and Yelick K., 1993: Parallel Programming in Split-C. Supercomputing

[17] Welsh M., 1999: Ninja RMI : A Free Java RMI. Available from http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html,.

[18] Goldberg I., Gribble S. D., Wagner D., and Brewer E. A., 1999: The Ninja Jukebox. In Proc. 2nd USENIX Symposium on Internet Technologies and Systems, October.

[19] Hirano S., 1997: HORB: Distributed Execution of Java Programs. In Worldwide Computing And Its Applications, Lecture Notes in Computer Science, volume 1274.

[20] Hirano S., Yasu Y., and Igarashi H., 1998: Performance Evaluation of Popular Distributed Object Technologies for Java. In Proc. ACM 1998 Workshop on Java for High-performance network computing, February.

[21] Mowbray T. J. and Zahavi R., 1995: The Essential CORBA: Systems Integration Using Distributed Objects. Wiley, ISBN 0-471-10611-9.

[22] Object Management Group (OMG), 1994: Common Object Services Specification, Volume 1. Available from http://www.omg.org/, March .

[23] Gokhale A. and Schmidt D., 1997: Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks. In Proc. 17th International Conference on Distributed Computing Systems.

[24] Gokhale A. and Schmidt D C., 1998: Principles for Optimising CORBA Internet Inter-ORB Protocol. In Proc. HICSS Conference, January.

[25] Falkner K. E. K., Coddington P. D. and Oudshoorn M. J., 1999: Implementing Asynchronous Remote Method Invocation in Java, Technical Report DHPC-072, July.

[26] Flanagan D., Farley J., Crawford W., Magnusson K., 1999: Java Enterprise In A Nutshell, A Desktop Quick Reference. ISBN 1-56592-483-5

[27] Izatt, M., Chan P., Brecht T., 1999: Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications

[28] Raje R., William J. I., and Boyles M., 1997:An asynchronous remote method invocation (ARMI) mechanism for Java. In ACM 1997 Workshop on Java for Science and Engineering Computation, June.

[29] Url-1 <*http://java.sun.com/products/jndi/1.2/javadoc/***>**, accessed at 16.04.2009.

[30] Url-2 <*http://www.ibm.com/developerworks/library/j-dyn0603/*>, accessed at 22.04.2009

CURRICULUM VITA

Candidate's full name:	**Orhan Akın**
Place and date of birth:	**Kırcaali, 08.05.1982**
Permanent Address:	**Ortaklar cad. Sakızağacı sk. Suda apt. D:8,Fulya, Şişli, İstanbul**
Universities and Colleges attended:	**Dokuz Eylül Üniversitesi, Bilgisayar Mühedisliği**

Publications:

- **Akın. O., Dalkılıç G.,** 2005: Anahtar Tabanlı Gelişmiş Rotor Makinesi, Akademik Bilişim.
- **Akın O., Erdoğan N.,** 2009: Enhancing Java RMI with Asynchrony through Reflection, EuropeComm, The First International ICST Confrence on Communications Infrastructure, Systems and Applications in Europe. 11-13 August 2009, London, UK.