

Application Partitioning - A Dynamic, Runtime, Object-level Approach

Dissertation

submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Anshu Veda
(Roll no. 04329022)

under the guidance of

Prof. Sridhar Iyer



Kanwal Rekhi School of Information Technology

Indian Institute of Technology Bombay

2006

Dissertation Approval Sheet

This is to certify that the dissertation entitled

Application Partitioning - A Dynamic, Runtime, Object-level Approach

by

Anshu Veda
(Roll no. 04329022)

is approved for the degree of **Master of Technology**.

Prof. Sridhar Iyer
(Supervisor)

Prof. Anirudha Sahoo
(Internal Examiner)

Prof. Om Damani
(Internal Examiner)

Prof. Varsha Apte
(Chairperson)

Date: _____

Place: _____

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

CERTIFICATE OF COURSE WORK

This is to certify that **Ms. Anshu Veda** was admitted to the candidacy of the M.Tech. Degree and has successfully completed all the courses required for the M.Tech. Programme. The details of the course work done are given below.

Sr.No.	Course No.	Course Name	Credits
Semester 1 (Jul – Nov 2004)			
1.	IT601	Mobile Computing	6
2.	HS699	Communication and Presentation Skills (P/NP)	4
3.	IT603	Data Base Management Systems	6
4.	IT619	IT Foundation Laboratory	10
5.	IT623	Foundation course of IT - Part II	6
6.	IT694	Seminar	4
Semester 2 (Jan – Apr 2005)			
7.	CS686	Object Oriented Systems	6
8.	EE701	Introduction to MEMS (Institute Elective)	6
9.	IT630	Distributed Algorithms	6
10.	IT608	Datamining and Data Warehousing	6
11.	IT680	Systems Lab.	6
Semester 3 (Jul – Nov 2005)			
12.	CS601	Algorithms and Complexity (Audit)	6
13.	CS681	Performance Evaluation of Computer Systems and Networks	6
Semester 4 (Jan – Apr 2006)			
14.	CS705	Web Search and Mining (Audit)	6
M.Tech. Project			
15.	IT696	M.Tech. Project Stage - I (Jul 2005)	18
16.	IT697	M.Tech. Project Stage - II (Jan 2006)	30
17.	IT698	M.Tech. Project Stage - III (Jul 2006)	42

I.I.T. Bombay

Dy. Registrar(Academic)

Dated:

Abstract

With the advent of increase in the computational complexity of the programs, it seems conducive to distribute a centralized program written for a standalone system onto a network of nodes. Application partitioning is one such technique, that aims at division (allocation) of application components amongst different hosts in the network, thereby getting a standalone application executed in a distributed setting.

Most of the existing work in *application partitioning*, uses classes as the basic component for partitioning. However, the behavior of an application is determined by the interaction of its entities at run-time. In an object-oriented program, the run-time entities are principally objects. We believe that, a more effective and relevant partitioning mechanism can be created by considering objects as the basic components for distribution.

We also believe that, a truly flexible partitioning system should as far as possible, postpone the decision regarding the placement of each component to run-time. Any such decision should try to optimize on both - the number of remote invocations that a distribution strategy would result in, as well as the load distribution on the available hosts.

We thus propose a working architecture that exploits the dependency relationships between the components, to build a model prior to the program execution. At run-time, the model progressively incorporates the information about already allocated components and helps in deciding the position of new component.

Contents

Abstract	i
List of figures	v
1 Introduction	1
1.1 Application Partitioning	1
1.2 Motivation	1
1.3 Problem Statement	3
1.4 Thesis Outline	4
2 Related Work	5
2.1 Introduction	5
2.2 Existing Middleware Technologies	6
2.3 Distribution Infrastructures	6
2.3.1 Issues Related with RMI	6
2.3.2 Distribution Infrastructures - A backend	8
2.3.3 Doorastha	9
2.4 Partitioning Schemes	10
2.4.1 Features of a Partitioning Scheme	11
2.4.2 Addistant and J-Orchestra	12
2.4.3 Coign	12
2.4.4 Pangaea	13
2.5 Application Partitioning - A Taxonomy	14
3 Object-level Partitioning - Architecture and Design	16
3.1 Introduction and Motivation	16
3.2 Architecture for Object Level Partitioning	17
3.3 An example	18
3.4 Dependency Analyzer	19
3.4.1 Significance of initial placement	19
3.4.2 Statistical Approach to O-L Partitioning	20
3.4.3 Bayesian Belief Networks - Applicability	21

3.4.4	Dependency Model	22
3.5	Translator	24
3.6	Run-time Analyzer and Query Engine	24
3.6.1	Object allocation decision	24
3.6.2	Functioning of Run-time Analyzer	25
3.7	Run-time System	26
3.8	Comparison with the Related Work	27
4	Implementation Details	28
4.1	Standalone Profiling	28
4.2	Coding the object model - BBN	29
4.3	Run-time Issues	30
4.4	Extensions to the current Implementation	30
4.4.1	Code Translation and Instrumentation for Run-time	30
5	Conclusion and Future Work	32
	Abbreviations	35
	Bibliography	37
	Appendix	39
A	Example	39
B	Instrumentation and Profiling	42
B.1	Tracker Class	42
B.2	Profiling Implementation	43
B.3	Profiling Output	46
B.3.1	Object Info	46
B.3.2	Dependency Matrix	48
B.4	Model	49
C	Example - Translated Code	50
C.1	Supporting Proxy class	50
C.2	Supporting Wrapper class	51
C.3	Instrumented CarModel class	51

List of Figures

1.1	Standalone to Distributed Application-Reduced network contention and effective resource utilization	2
2.1	Proxy-Based Approach-Transparent handling of remote and local references	10
2.2	Pangaea - Object Graph	13
2.3	Application Partitioning Taxonomy	14
3.1	Class vs Object Level Partitioning	17
3.2	System Overview	18
3.3	An Example	19
3.4	Dependency Analyzer	20
3.5	BBN - An Example network	21
3.6	Runtime-Analyzer	25
3.7	Illustration of the runtime	26
4.1	Profiling Flow	29
4.2	BBN Generator	30
4.3	An example visitor pattern.	31
B.1	Tagged object information extract	47
B.2	Dependency Matrix	48
B.3	Dependency Matrix	49

Chapter 1

Introduction

Distributed computing has emerged as an inevitable need in the modern world of large-scale systems and high power networks. Since quite a long while however, the emphasis has been on client-server programs and to an extent, concurrent programs intended for parallel programming.

In current-day scenarios where there are multiple complex applications, deployed on a network of multiple nodes using widely distributed resources, distributed execution is highly useful. However, coding such an application is not so easy. Programming languages like C/C++, Java provide RPC¹-based RMI², CORBA and like techniques. However, these techniques do not provide enough transparency to the programmer. It is thus desirable to have a system which not only automatically distributes a standalone application, adapting to varying distribution settings, but also provides the advantages of distributed computing.

1.1 Application Partitioning

Application partitioning refers to, *breaking up* of the application into components while preserving the *semantics* of original application. The source application might be designed, implemented and debugged to run on a *standalone* system. The resulting components however, are *distributed* so as to take advantage of distributed setting.

1.2 Motivation

Partitioning of Object-Oriented programs is motivated by the following reasons :

- ***Distribution of program components***

Considering the scenario of multiple large-scale, multi-user, heterogeneous applications using a wide variety of resources in a network. Code for a standalone system based application, as shown in Figure 1.1 would be *monolithic*, functioning at some

¹Remote Procedure Call

²Remote Method Invocation-The object oriented counterpart

central node making expensive network calls. A distributed application would however, have components (fragments) distributed all over the network 1.1, reducing amount of remote communication overhead, thereby optimizing the program execution.

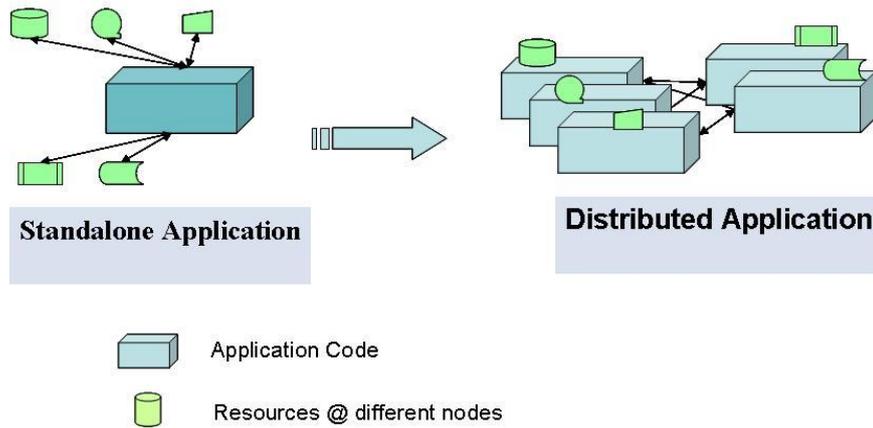


Figure 1.1: Standalone to Distributed Application-Reduced network contention and effective resource utilization

For illustration, consider a large banking application, which has its databases distributed over geographically separated areas. A standalone code would be performing all the computations at the server end and make expensive network calls, whereas a distributed application can have computations dispersed according to resource distribution. In addition, idle hosts can share the load of a busy system by hosting a few components. With the increase in the computational complexity of the programs, as well increase in number of network resources, it makes sense to have a tool that could distribute a centralized program written for a standalone system on to a network of nodes. Such an idea comes with the natural advantages of distributed computing - load reduction, resource sharing and to an extent improvement in concurrent execution. Thus, assuming the presence of an underlying reliable, high-speed computer network, on one hand when we would like to have distributed components of a *single* program running at different computer nodes, at the same time we do not want programmers to worry about the distribution aspects while coding. In other words, we would like to *separate the distribution concerns from application logic*.

- **Abstraction over the middleware**

Java, C# provide *Remote Method Invocation (RMI)*, *CORBA* and like mechanisms,

for developing remotely invocable components. However, these middleware technologies are *complicated* to code. They have special coding constructs, and programmer has to think about the deployment scenario while designing the classes. Java RMI for example, needs each remote class to be specially defined to extend a special interface. Creating a remote object and getting its access on remote node is a tedious job.

- ***Abstraction over deployment scenarios*** Primarily been designed to cater to *client-server* systems, current middleware technologies also fail to provide enough abstraction. The programmer has to think and consider the distribution aspects while designing the application. Finally, these constraints prevent the program to be *re-deployed* efficiently in different distributed settings. The above arguments justify the need of an automatic partitioning system, that optimally partitions source application written for a standalone system. In other words, a system that helps separate distribution concerns from application logic.

We have studied and classified existing partitioning mechanisms, identified their limitations and subsequently designed a working architecture which provides enough flexibility even at run-time to distribute components(objects). We have designed a Decision analyzer which aims at placing components *intelligently* without much human-intervention, accompanied by a run-time algorithm that uses modification of an existing backend (Doorastha [Dah00]) for distribution.

1.3 Problem Statement

This thesis proposes a novel architecture for application partitioning - an automated, object-level, dynamic, run-time partitioning system, more specifically a distribution analyzer and decision engine. We believe, that proposed architecture, would be an efficient and flexible solution to the application partitioning problem.

We have identified different issues involved in each component of the architecture and suggested solutions for the same. The key ideas which make our approach novel, include the following:

- Finer granularity of components, considering objects and not the classes.
- A statistics-based dependency model to represent interdependencies between application components.
- A truly dynamic, run-time allocation scheme for distribution of components

We feel that finer granularity partitioning decisions, would lead to more flexible and efficient partitioning mechanism. In addition to the granularity of partitioning, it is also important to decide the placement of different components on available hosts. Finally, we

strongly feel that there are numerous factors that can affect object allocation decisions, some of which come into picture at run-time. The existing systems are limited in the sense, that they *statically* decide the component allocation, before the execution of program. Most of them depend on user-specification and do not emphasize on initial placement decision. This certainly is not a very good idea for long-running, large-scale programs, having multiple components.

We have implemented a statistics-based Decision Analyzer. This tries to learn and represent dependencies as well as independencies (between the components) as a probabilistic model. The run-time algorithm uses and updates this model. It is flexible enough to account for factors like load distribution, and hence is truly dynamic. We have used *Java* for the demonstration of our ideas. However, the ideas discussed are in general applicable for OO languages.

1.4 Thesis Outline

Chapter 2 discusses related work in application partitioning, based on which we propose a taxonomy for application partitioning. Chapter 3 discusses the various blocks of proposed O-L partitioning architecture including the various design decisions that were taken. Chapter 4 highlights the implementation details. Finally, we conclude in chapter 5 highlighting some limitations and future work.

Chapter 2

Related Work

2.1 Introduction

The idea of distributed computing was introduced, based on RPC by languages like Java. RMI ¹, CORBA in Java or DCOM in C++, allowed user to access remote objects, and execute methods on them. However, all these are more suited for designing client-server paradigm based architectures. After these, there came up systems, which were built over middlewares like RMI/CORBA. They enable easy code generation, for remote objects. These were then built upon by front-end systems which used some partitioning strategies to identify components to be distributed.

The problem of partitioning in literature has also be dealt with mechanisms that work by modifying the execution environment of application. This has been referred to as the *implicit approach* in [Spi02]. These systems do not rely on the programming language. Instead, they use a Distributed Shared Memory(DSM) environment. DSM mechanisms involve caching and replication, which can be done at different levels like page and object. Though they make things more transparent, as discussed in [Spi02], the efficiency of these systems is not very commendable. Also, they are often non-standard and cannot be extended to different machine architectures.

Thus, we have restricted our focus to what has been referred to as explicit approach in [Spi02]. To explain briefly, explicit approach works from within the programming language, rather than modifying execution environment. Thus all the solutions that we discuss, actually form a layer above the standard middleware of the corresponding programming language, and hence are more standard.

The related work has been discussed under four heads. Section 2.2 discusses in brief about existing standard middlewares. Section 2.3 discusses the work done on the distribution infrastructures. Distribution infrastructures implement distribution policy, suggested by the partitioning strategy. Section 2.4 discusses the existing work on partitioning strategies and mechanisms.

¹Remote Method Invocation

2.2 Existing Middleware Technologies

Popular programming languages like C++, Java, .NET provide distributed constructs for remote execution. This design is greatly influenced by the RPC mechanism. In the RPC mechanism, the caller invokes a dummy procedure(stub) on local machine, and this in-turn sends network message to the server which gets the method executed on actual object and returns the results. RMI does this at object-level. Each remote object has a stub which acts as a proxy at the client site for transferring the call to the actual object. RMI [RMI96] is Sun's proprietary client-server oriented distribution technology for the Java language. It allows the programmer to create remote classes, i.e. classes that implement the *java.rmi.Remote* interface. The user also provides an interface listing all the methods that may be called remotely which enforces such methods to be public. It uses the concept of registry or name server, which hosts the objects. Each object is bound with a specific name on the server. When queried, the object server then returns a remote reference of object. This reference can then be used to invoke remote methods. Once the objects are registered by the RMI system, they are represented as network references and remote objects are always passed by-refvalue. The backend used in current implementation is based on RMI.

CORBA is based on the same basic principle, except that its more useful for interoperability between different platforms. DCOM for C++, again uses the same concept of interfaces and stubs.

The essence is that using existing middleware standards, coding a distributed application requires a lot of planning in the design phase of application itself, concerning where different objects and classes should be *created* and how they should be *accessed*. Moreover, it is a lot of hassle to actually code such an application, because of special remote-enabled code constructs.

2.3 Distribution Infrastructures

We define *Distribution Infrastructures* as systems that form a thin layer over the middleware. These when provided with detailed distribution policy specification, usually work by code transformation and some extra code generation. These systems have to deal with issues a little more specific to the middleware being used. Most of the systems based on Java use RMI.

2.3.1 Issues Related with RMI

RMI constructs in Java try to minimize the difference between distributed and non-distributed syntax. Thus method invocation syntax is similar to that of general non-distributed Java programs. However, there are certain issues that arise when we try to

use it for distributing a general-purpose OO program. First of all, each remote class should necessarily have an interface and an implementation. In addition, the exception handling is not very convenient. Actually RMI was basically designed for client-server applications and so is not very convenient to program a general OO system with inter-class relationships. Below is a list of few issues that arise when RMI is used for programming a general OO program.

- Static and final methods cannot be defined in RMI implementation classes. RMI only permits the use of public and abstract modifiers for the method declaration in remote interfaces. Also in cases where class is hosted on multiple nodes, handling static fields is a problem if they are not constants.
- *Consistency of Object References*(Identity of Objects)
 - Equality operator for remote objects(equals or ==) may not work properly. For e.g. If two remote references of same object obtained from different sites, the == operator only compares references, and does not work according to OO semantics.
 - Reference translation does not happen when object is moved amongst multiple hosts. To explain further, if an object is passed from site 1 to 2 and then 3; when referenced from site 3, the object would be referred via site 2. Local object reference retrieved from a remote host as a RMI reference has to be accessed via indirect path.
- *System Classes* (with and without native code).
Source code for system classes is not available. Thus we cannot expect system objects to be remotely accessible. But objects of system classes like Vector, if be defined remotely can help improving performance.
- *Inheritance*
Super class needs to extend UniCastRemoteObject, even if sub-class needs to be made remote-capable. This does not seem to be quite correct if we go by OO semantics.
 - All implementation classes need to extend from UniCastRemote Interface. In Java, only single inheritance is allowed. This implies all super classes need to be modified (OO paradigm violation). Transformations may cause breaks in inheritance chain and instance of, sub-classing may not work as in a centralized program.
 - In case of classes extending system classes, this too is not possible.
- Methods with *package* scope have to be made public as the interface methods have to be public, or otherwise all package references should be guaranteed local.

- **Arguments**

By default a non-RMI object is passed by-value. System Objects cannot be made remotely accessible. If all the objects are made remote capable, we can handle usual pass-by-reference semantics of Java programs. However, this may be too expensive in some cases. Based on these considerations for remote execution in addition to usual passing modes(by-value, by-reference) following modes can be defined.

- *By reference value*(ref value)

The callee operates on the original object via the given reference and thus may alter its state. However, the reference cannot be modified. This mode is used when a centralized Java program runs.

- *By copy*

The callee operates on a (deep) copy of the original object which itself remains unaffected.

- *By visit*

The object is moved back, after the invocation returns.

- *By move*

Here, the runtime system moves the object physically into the callee address space, and redirects any old references.

- *Handling of synchronized classes*

RMI does not allow synchronized methods to be declared in interface defined for the public class.

- *Slow marshalling and de-marshalling process*

- RMI stubs serialize all the parameters being passed(except network references) on a remote invocation. This process is very time consuming.

2.3.2 Distribution Infrastructures - A backend

These distribution infrastructures, have been used as backends. The partitioning schemes as discussed in section 2.4, generate a *distribution* or a *partitioning* policy, which is implemented at run-time by these backends, with remote enabled code, over the middleware.

These systems allow specification of distribution policy for each class and its methods. The specification details some of the following:

1. Location of the class, if needed to be fixed on one node J-Orchestra[TS02]
2. Location of an object when being instantiated Doorastha[Dah00]
3. Parameter passing mode(visit, move, copy) for formal arguments for each method.

4. Classes can be tagged as migratable in which case at run-time their instances can migrate to other nodes as arguments or return values during a remote method invocation.

Most of the systems have used statically determined positions computed by analysis or specified by the user. Thus run-time system never has to bother about the position of object. Some of the ways used to specify the distribution policy include the following:

1. *Additional Constructs*: Additional keywords and constructs extending Java language are provided to declare remotely accessible classes. Java Party[PZ97]
2. *Comments*: Comments written above the classes in a specified format. Doorastha[Dah00]
3. *Configuration File*: Have configuration file per class. J-Orchestra[TS02], uses an *XML* config file, Addistant [TSCI01] uses a special configuration file, indexed by hosts for the same.

These systems have a translation component which processes the distribution policy and transforms the source code into remote-enabled code accordingly. Another major task is to handle the various issues related to the middleware (RMI). There are a number of systems that provide such infrastructure support. Javaparty[PZ97], Doorastha[Dah00], J-Orchestra backend[TS02], Coign Backend[HS99], Addistant[TSCI01]..

Most backends try to solve the issues highlighted above. We are discussing the architecture of Doorastha in detail, as it has been modified and used as distribution backend in our architecture.

2.3.3 Doorastha

Doorastha [Dah00], has a compiler subsystem that uses annotations within the source code (as comments), to generate the RMI equivalent code.

2.3.3.1 Translator

The Doorastha translator, apart from code conversions necessary to handle the RMI issues as discussed in section 2.3.1 is responsible for maintaining transparency between remote and local object communication. The key principle, which has also been used in javaparty[PZ97], J-Orchestra backend [TS02] is a *proxy-based* approach shown in Figure 2.1. This approach tries to maintain the transparency between local and remote references of an object being accessed by making use of elicit mechanism of inheritance and polymorphism. More details can be referred from [Dah00]. The translator generates the necessary code to implement the same.

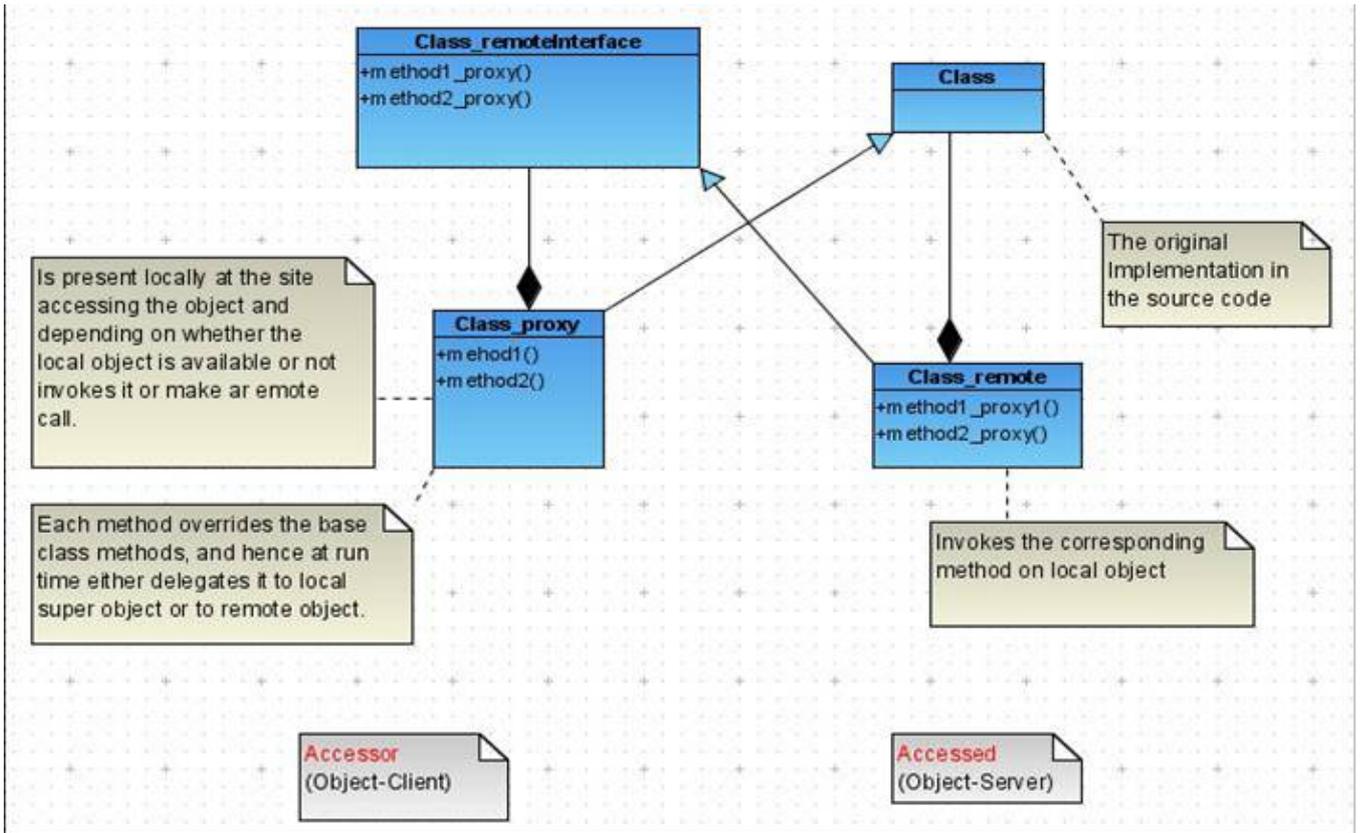


Figure 2.1: Proxy-Based Approach-Transparent handling of remote and local references

2.3.3.2 Runtime System

Doorastha run-time is a distributed algorithm, where run-time entities sit at each host and interact. This sub-system manages translation of object references at run-time. It provides object factories at each host. Also, it maintains references of all objects, instantiated on a particular node. It manages network-related issues like resolving host-names etc. It manages the start-up and running of run-time systems at all the hosts. It also helps managing different argument-passing modes by providing support for migration.

2.4 Partitioning Schemes

The distributed infrastructures discussed in section 2.3 expect a detailed specification about component. Specification in literature [TS02] has been defined as - the location of component instantiation, its mobility, and if it is defined migratable, a migration policy describing when it will move. This can be defined by specifying the parameter-passing mode (visit/move, reference, value) in the configuration file for each function. We have mainly concentrated on initial placement as of now. Thus the distribution policy too considers the same.

A partitioning scheme tries to use some heuristics to generate a distribution policy.

Higher distribution causes more components to be distributed across different hosts. With increase in distribution, more local interactions are converted into remote. Remote invocations increase network overhead. Thus though various reasons suggested above in section 1.2 motivate to distribute, the tradeoff is that distribution increases remote communication overhead. A partitioning mechanism should try to balance between these two contradicting factors. We have identified some factors, that can be used to characterize a partitioning scheme.

2.4.1 Features of a Partitioning Scheme

2.4.1.1 Granularity of Distribution Component

Usually an OO program comprises of classes and objects, and hence on the basis of *granularity* of partitioning component we can classify the existing partitioning schemes.

1. Class-Level Partitioning

This kind of partitioning aims at grouping closely interacting classes together. The source code comprising of different classes can be divided across the hosts. Some systems may also support replication of classes. This kind of partitioning is based on the assumption, that objects of a single class exhibit similar behavior, and will be using same resources and hence should be instantiated on the same host. In a nutshell, all objects of one class are created on same host. Moreover, classes are clustered into groups.

2. Object-Level Partitioning

This kind of partitioning aims at clustering of objects. Object level partitioning stems from the arguments like - reusability being an important aspect of OO programming, objects from the same class may exhibit different behaviors when they interact with different objects. Also, allocating all objects of one class to one node is a great limitation. Thus O-L Partitioning, does not primarily aim at division of source code, but division of objects across the hosts.

2.4.1.2 Analysis Techniques

The partitioning scheme needs to identify closely interacting components so that they can be co-located. This involves program analysis, to identify dependency relationships between the components. The analysis can be static which can in turn be at source level or bytecode level. Alternatively, profiling can be used, to study the component interactions and subsequent logs be then analyzed and classified.

1. Static Analysis

Involves techniques like, call-graph creation and doing dependency analysis based on it. Available libraries for code analysis can be used. *Source-code* level analysis are

more easy to understand, whereas *Bytecode level* analysis techniques are extendible to system classes.

2. Dynamic Analysis

Involves instrumentation of the existing code, to profile application runs. This profiling again can be of two types. It might be on the *standalone code* itself, or *remote-enabled* code.

2.4.1.3 Allocation Decision

This feature is based on the time of actual allocation decision. All the systems we know of, work by pre-deciding the component allocation before the execution of program starts.

1. Offline

Component allocation is decided and specified in some configuration file or in the form of annotations within the program.

2. Online

Component allocation decision is postponed till run-time. As per our knowledge, there are not any such systems using online techniques as yet.

2.4.2 Addistant and J-Orchestra

J-Orchestra [TS02] is a class-based partitioning mechanism. J-Orchestra also has a backend as discussed in section 2.3. Addistant[TSCI01] has a configuration file, which can be indexed by hosts. This file is used by the backend for code generation. For each host, a class-level or package level specification may be done.

J-Orchestra too has a similar configuration file, where-in for each class, properties can be specified. For J-Orchestra however, partitioning mechanism, is expected to generate this file. Once the host specification has been provided, J-Orchestra allows user to drag and drop classes to different hosts. The classes however get allocated in *groups*. A static analyzer and profiler component are used to each time verify the *goodness* of partitioning by profiling. It reports statistics on the interdependencies, based on which an efficient configuration can be decided.

Major contribution of addistant and j-orchestra has been to enable distribution of legacy software. Operating at bytecode level, they try to make objects mobile at run-time in the presence of unmodifiable code (like that of JNI).

2.4.3 Coign

Coign [HS99] is implemented for COM-DCOM systems. It can support object-level partitioning. Input to the Coign system, is a COM application. Coign instruments and profiles this application by invoking DCOM code, to actually calculate the cost, if

components were distributed. A component can be a class or an object. Coign supports different types of classifiers. The system can thus be configured to define an appropriate component. Two of them for eg are -

- *Static type Classifier*: Classification descriptor is just the type.
- *I3C Classifier* : Classification descriptor is full component call chain of instantiation request, concatenated with static type.

Coign then creates a component graph, where nodes include the different component descriptors and edges represent the communication costs, calculated during profiling. As Coign considers communication between remotely located components by-copy, all the functional arguments add up to communication costs.

This graph can then be partitioned using a standard graph partitioning algorithm, into number of partitions equal to number of hosts. The system has been mainly tested, for client-server applications.

2.4.4 Pangaea

Pangaea [Spi02] uses Doorastha [Dah00] as the backend. Pangaea does a primitive level of Object-level Partitioning. It uses a graphical tool to represent the program structure as an *object-graph*. This object-graph is built using purely *static* analysis of the code. An object graph Figure 2.2, represents the following:

- The type information (data type) is considered and the data flow at the type-level is represented as a graph.

- Different types of objects - static and dynamic (Concrete and indefinite) are identified, for each class. The concrete are the ones which are certain to take specific value on creation. The indefinite ones may assume one amongst a range of values. Eg. Producer in producer/consumer example is concrete whereas an integer could be indefinite. Different constructs of the program - Constructors, static initializers, main(), Thread.run() etc have been used to categorize all the objects into the above categories. Such objects become the nodes of the graph

- Edges encode the following relationships between the objects - Creation, Usage, reference(import, export). These can be identified by the analysis of method arguments, return types etc.

- The object definitions are applied recursively and added to the graph. Eg. For each concrete object, concrete and indefinite objects are added for the fields etc accordingly.

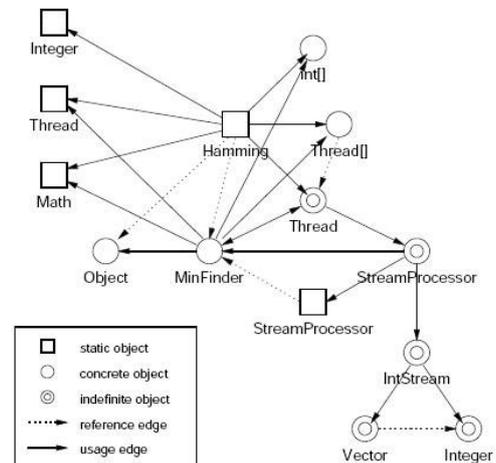


Figure 2.2: Pangaea - Object Graph

- Finally, the references are propagated transitively.

Once the object graph 2.2 is constructed its displayed to user, and user can assign objects to different hosts. For numerous objects round-robin allocation is done amongst the hosts. Pangaea then generates the actual *annotated code* that can be interpreted by Doorastha.

2.5 Application Partitioning - A Taxonomy

Based on our study and analysis, we present a taxonomy of *possible* application partitioning approaches in Figure 2.3. Application partitioning can be dealt with implicit or explicit mechanisms. Explicit mechanism based solution should have a partitioning strategy and a distribution backend. Distribution backend implements a partitioning policy specified by the partitioning strategy. Partitioning strategy again can be classified on the basis of granularity of partitioning component, allocation decision and analysis technique.

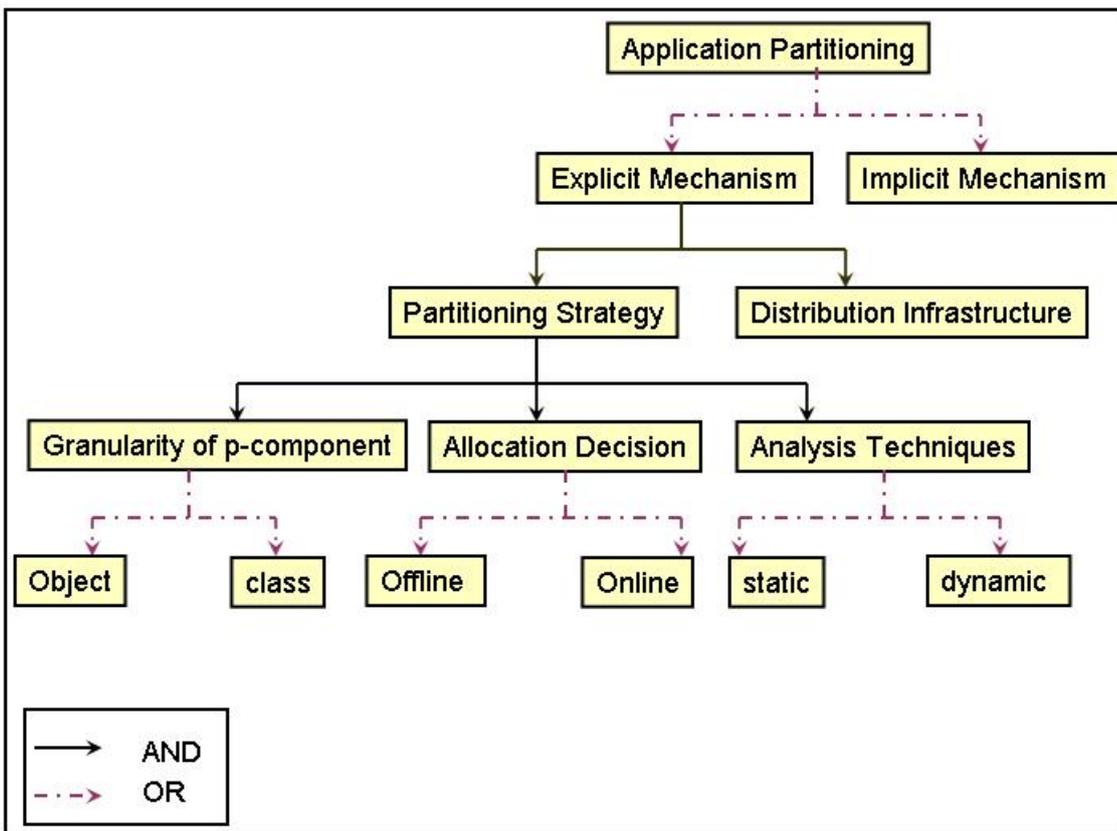


Figure 2.3: Application Partitioning Taxonomy

The related work has been partly concentrated on distribution infrastructures - Doorastha [Dah00], Javaparty [PZ97], Addistant [TSCI01]. Though J-Orchestra does some static

analysis, its major contribution has been on the distributed infrastructures front. Pangaea [Spi02] advocated O-L partitioning in a very limited sense. It needs user to specify allocation details. Coign[Hun98] is O-L system, but has other drawbacks as discussed in section 2.4.3. Its primarily designed for COM but still tries to eliminate human intervention. COM does dynamic analysis(profiling), but its done on DCOM based code.

As per our knowledge, there does not exist any system which takes run-time allocation decisions. Also, none of the systems have a sophisticated analysis technique. Pangaea and J-Orchestra do static analysis and are highly user dependent. Coign does dynamic analysis and profiling but has the limitation that passes all arguments by complete copying.

In the next chapter3, we discuss our architecture for an object-level, dynamic analysis, and run-time allocation-decision based automatic partitioning system.

Chapter 3

Object-level Partitioning - Architecture and Design

3.1 Introduction and Motivation

This section discusses the detailed design of an object-level, dynamic, automatic partitioning system which takes run-time allocation decisions. The key motivation behind this architecture is, that we want to exploit as much information as possible (available), offline and yet provide enough flexibility at run-time, to relocate object allocations depending upon load distributions. Dynamics of an application are decided at the run-time, where objects are most relevant to partition. O-L partitioning as discussed in Section 2.4.1.1 aims at clustering closely interacting objects, to optimize on network overhead, while the application is distributed across a network of hosts. We strongly believe that O-L partitioning is an efficient partitioning technique motivated by the following reasons:

- **Importance of has-a relationships** Class represents the behavior of a program component, whereas an object is its instance, actually functional at run-time. Class-level partitioning systems view program as a collaboration of classes and ignore the has-a relationships. A class can have multiple instances. Multiplicity of objects is one aspect which cannot be considered while doing a class-level partitioning.
- **Instance Behavior** In literature, most of the work has been on partitioning the application by placing its classes on different nodes. However, an important point here to note is that in a large application, one class may be interacting with many other classes, which in turn may not be related at all. To explain with an example, we can consider distribution of a program comprising of Vector class 3.1 being used by two totally unrelated classes for storing objects. Now keeping Vector class and all related classes on one node or making remote references for all classes is not a good idea. The key point to note here is that instances of vector class may be useful in two totally different contexts and hence we argue that a distribution policy considering distribution of objects is more plausible.
- **Argument Passing Modes** Another point to be considered is that in class-level partitioning systems, while defining properties of the class-methods, we fix the

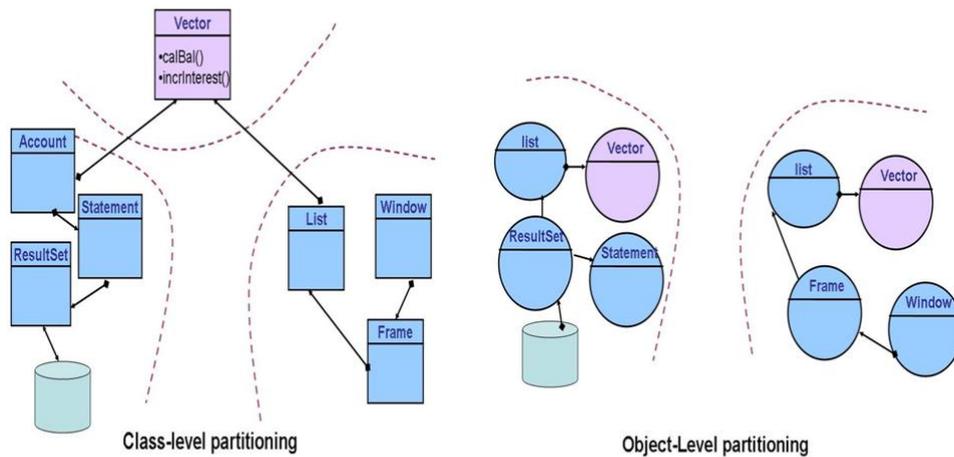


Figure 3.1: Class vs Object Level Partitioning

parameter-passing mode (call-by-visit/call-by-move/copy) for each method. Multiple invocations of the same method however by different classes, may make more sense to have different calling modes at runtime. Specially, if the class can be used in different contexts and its different instances are distributed.

O-L partitioning surely provides greater flexibility as far as the partitioning and distribution is concerned, but is complex to implement. In a large-scale program, however, where there are heavy objects, it certainly makes sense to consider different application scenarios, and object level interactions and then place the object. Section 3.2 describes an overview of architecture partitioning system. Subsequent sections then elaborate on different blocks of the same.

3.2 Architecture for Object Level Partitioning

As discussed above in section 2.4, none of the existing schemes provide much flexibility. We have tried to make this as flexible so that it can be further extended to incorporate as many as factors possible at run-time. The system as shown in figure 3.2 begins with the input of *application source code*, written for standalone environment. This code is analyzed by the dependency analyzer to generate a model representing dependency relationships between the objects. The dependency analyzer also plays the role of identifying different

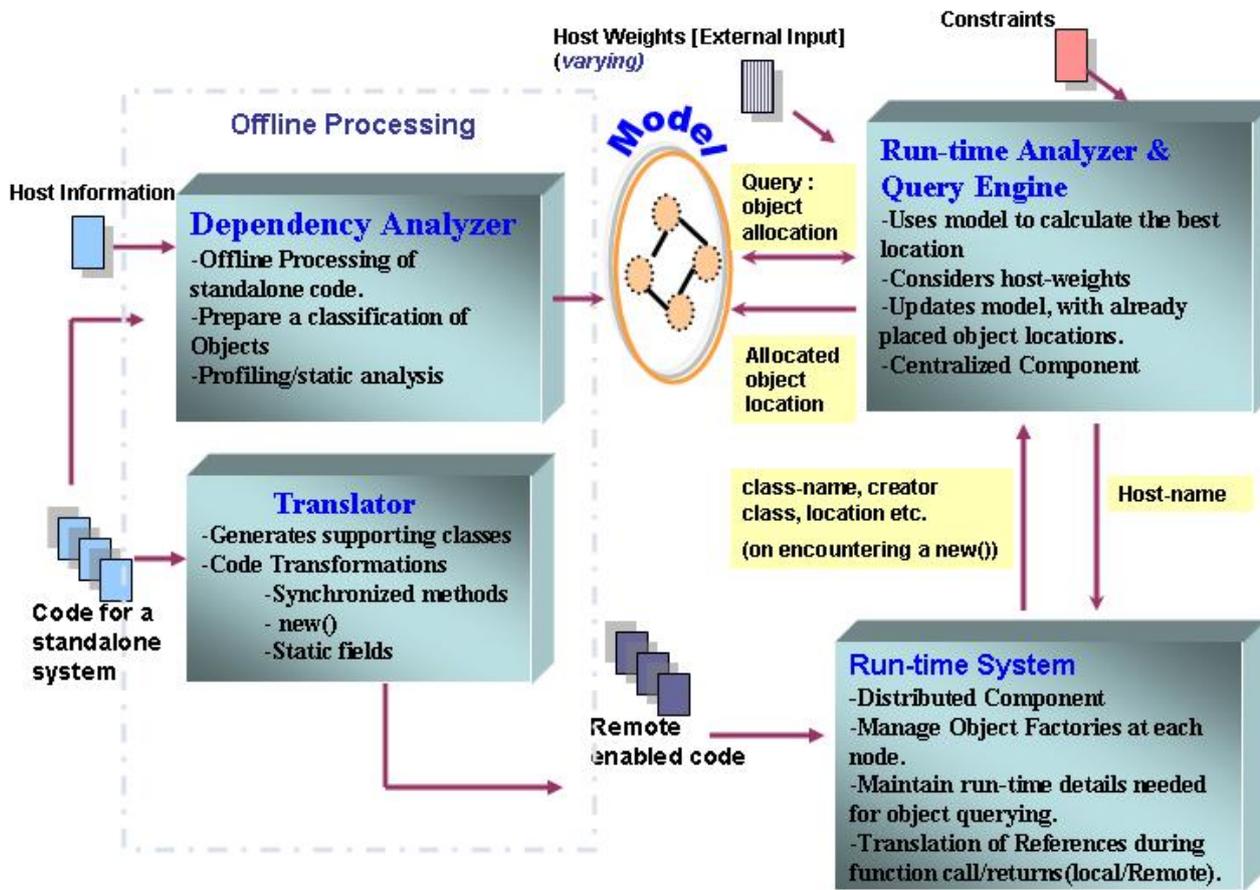


Figure 3.2: System Overview

contexts in which objects may be created. This code is then given to a translator, that uses some code analysis/manipulation library to generate a remote enabled code. At run-time then, as progressively objects are allocated, all the available information is used to *predict* the most suitable location for object allocation. The run-time sits as a layer above RMI. In the next few sections, we discuss each of the blocks of the proposed architectures in detail accompanied with a tested example to illustrate our architecture.

3.3 An example

This is a very simple program as depicted by the class-diagram in figure 3.3. The source code for the example is attached in the appendix A. We assume that there are *three* hosts available for placement. We will walk through this section illustrating this example to explain the functioning of each block.

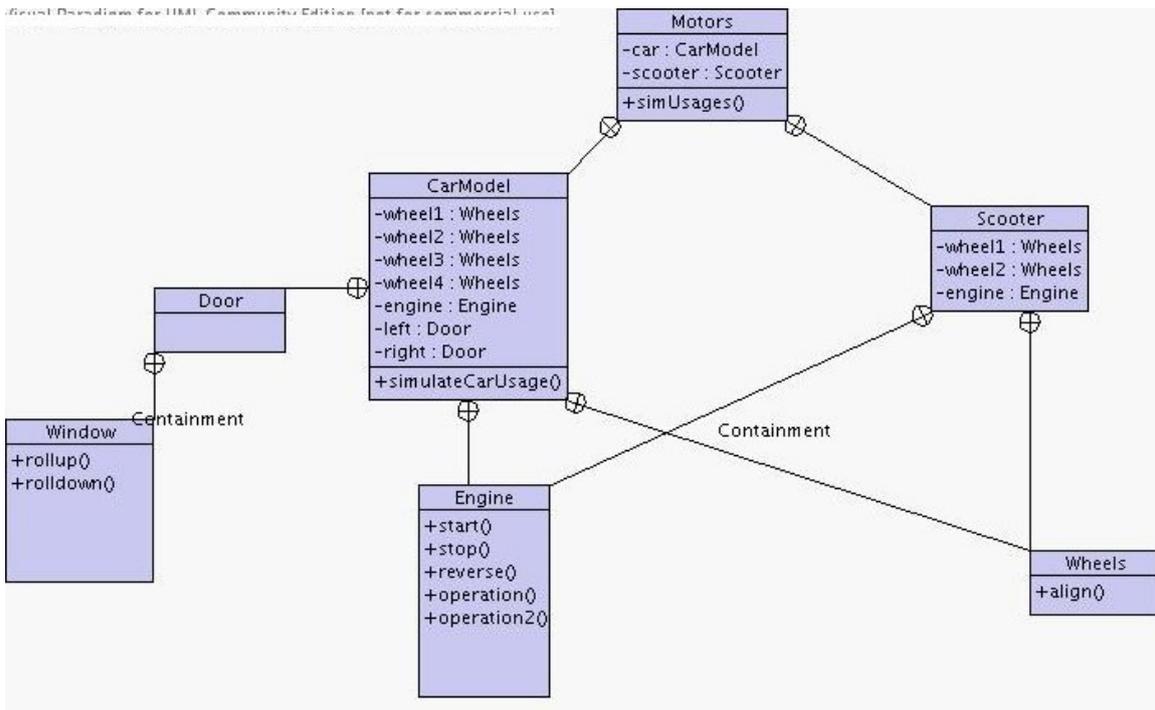


Figure 3.3: An Example

3.4 Dependency Analyzer

The goal of a dependency analyzer is to learn application patterns, dependency relationships between the different components and represent it in a model that can be used at run-time. Figure 3.4 shows the architecture of our dependency analyzer. The standalone code is instrumented and profiled to generate dependency related information. There can be several such runs of the program. These can then be used by a learner to learn object dependencies.

Section 3.4.1 explains the need of a distribution analyzer. Subsequent sections mainly elaborate on the learner, illustrating our analytical model and justifying its applicability.

3.4.1 Significance of initial placement

We feel strongly that there is a need for an optimal partitioning strategy, if partitioning process has to be completely automated. Even if backend provides migration support, migration of heavy objects can often be a network overhead. Thus, a good partitioning software must try to optimize initial placement of object. Now there are several factors that may influence placement of an object O .

- Location of other objects that invoke methods on O
- Resources accessed by O . This is more class-dependent

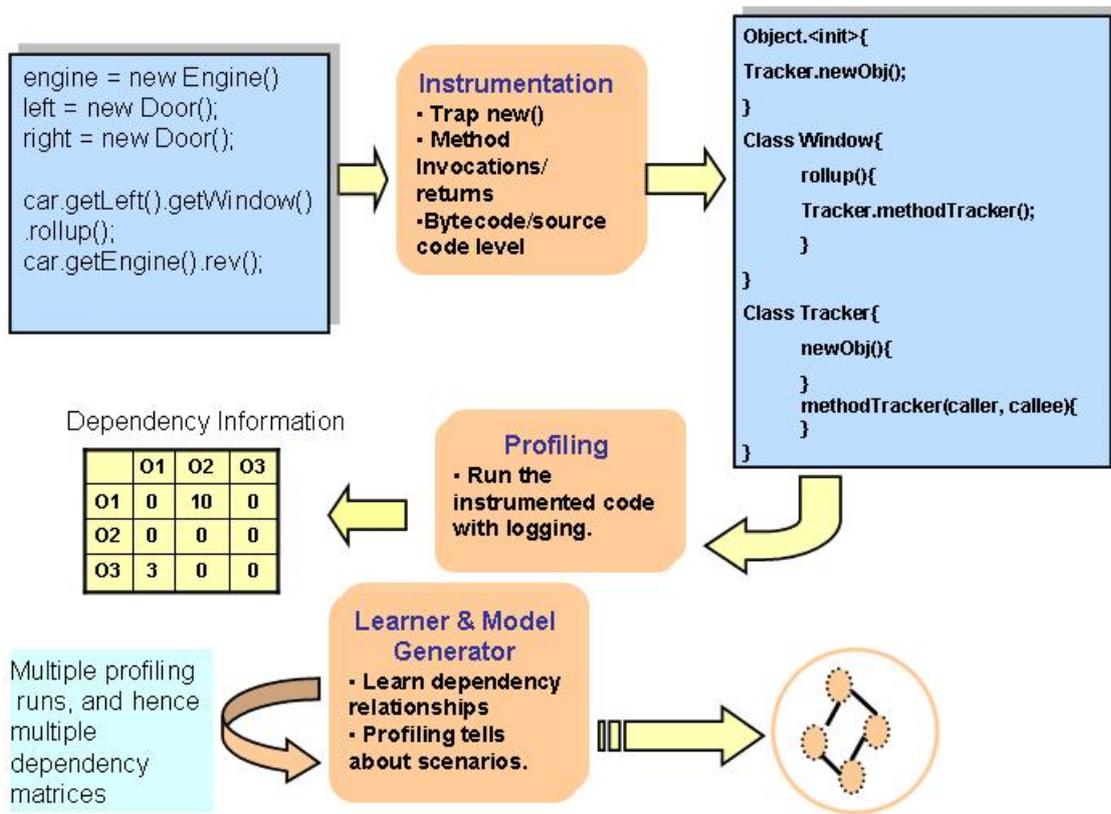


Figure 3.4: Dependency Analyzer

- Control flow of the program, which might influence creation of other objects, that may be accessed by O and thus indirectly the placement of O itself.
- Host-load scenario

3.4.2 Statistical Approach to O-L Partitioning

O-L partitioning aims at optimal placement of components - *instances*, to the most optimal position in the network at run-time, so as to minimize the communication overhead as well as use the distributed environment. Component interactions can be minimized, if highly interacting components are co-located as discussed in Section 2.4.1. We propose a slightly different approach to this problem, based on machine learning. We suggest the use of the *prior knowledge* available from profiling and represent it as a model. A probabilistic model is apt because if we have numerous runs of an application and a large profiling data, a dependency model fitting most of the runs of the application can be built. Thus we suggest the applicability of *dependency-based classification* techniques for this problem.

The idea is to train a classifier from the profiling runs for an application. This classifier can be used to classify the instances at run time to different hosts. We however think, that this should be a case of active learning classification as depending upon the environment,

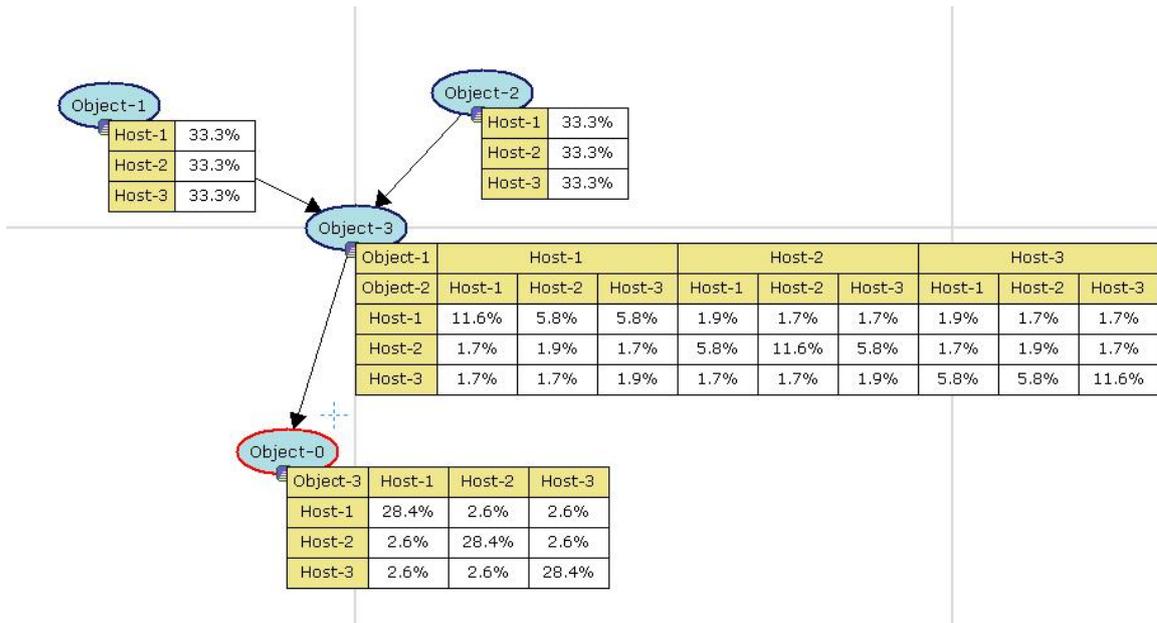


Figure 3.5: BBN - An Example network

the model should keep on adapting.

3.4.3 Bayesian Belief Networks - Applicability

Before explaining the details of the model, we would like to specify clearly what a component denotes in our design. Each component in our system is identified by its unique call-trace (the exact sequence of method calls which led to the creation of the instance). It has a class-type (the class of which it is an instance). Thus one class can have multiple components represented. We are trying to model the relationships between these components, on the basis of references.

Bayesian Belief Networks [BBN98] is a well-known statistical modelling technique. BBNs model relationships between the different variables by representing the conditional dependence as well as independence between them. Structurally, BBN is a set of nodes and arcs. Nodes represent the variables. Each variable can assume a set of values (discrete/continuous). These values are also termed as *class attribute*¹ sometimes. Arcs between the nodes represent dependency relationships. Each node is annotated with a probability table. Each row of this table, denotes a probability - the chance of variable to assume a value conditioned on different values assumed by its parents². Each combination of values of variable and its parents is a separate row. An example BBN could be as shown in Figure 3.5.

The BBN can easily be used to represent the following:

- Conditional *dependence* as well as *independence* of objects on other objects.

¹BBN is a well-known classification technique

²Parents of node A, include nodes having an outgoing edge to A

- Cumulative effect of already allocated objects on other objects.
- Communication Overhead can be incorporated into the probability tables of nodes.
- We assume that objects are always passed-by-reference between components(as against Coign2.4.3). However, the interaction between the arguments passed when componentA invokes a method on componentB, passing componentC as argument, can be accounted as interaction between C and B. Thus DAG very aptly capture the dependency relationships between the objects.
- BBNs can be extended to also handle, different application control flows, and incorporate them using probabilistic BBN learning algorithms.

Dependency information gathered from different profiling runs, can be combined by a learner to develop a model. BBN learners can use this data as training data. Each profiling runs' output suggests a different BBN structure. The BBN learning would therefore be a kind of search amongst these structures, for an appropriate model.

3.4.4 Dependency Model

We model each *component*(as defined in 3.4.3) in our system as a *Bayesian Node*. The set of hosts represent the class attribute for each node. Thus each arc represents dependency between the components. Each row of the probability table denotes one configuration ³ of parent objects and the corresponding node.

According to the approach proposed, if we have large logs indicating instantiation histories over several runs of the program, a BBN learning algorithm [Bou94] could be applied to learn a model that would be true representative of the application behavior for most runs. This would include the following :

- Number of times that component occurs in all runs, and for each such existence, number of times a reference from a particular incoming parent was made.
- Average communication overhead. This overhead would involve :
 - Number of times the two components communicate
 - Cost of exchange of immutable parameters like *String* ⁴ and others that are passed by total copy(if any)
- Location Constraints can be modelled in probability tables. By location constraint we mean that instances of a few classes may be constrained to be anchored on particular hosts.

³object to host mapping

⁴Java.lang.String

3.4.4.1 Simplistic Version of the Model

For the purpose of demo and to maintain simplicity, we have considered a single flow of application. For this single flow we try to develop a model, without using specialized learning algorithm.

As per our simplistic version of the sophisticated dependency model, arcs of BBNs represent the referred to and referred by components for each node(component). We also consider communication overhead between two components as just the number of remote invocations.

3.4.4.2 Mathematical Formulation of the CPT

Let the number of interactions of a component C_i with component C_j be $N_{i,j}$. where $C_j \in \text{parent of}(C_i)$ Number of rows in the CPT,

$$\text{sizeof}(CPT) = \text{num_of_hosts}^{\text{num_parents}+1} \quad (3.1)$$

For a particular configuration(allocation) - $A_{i,k}$, involving C_i and its parents⁵, representing the k^{th} row of the CPT, where $H(C_i)$ represents the host allocated to component C_i

Number of remote invocations,

$$R_{A_{i,k}} = \sum_{H(C_i) \neq H(C_j) \text{ and } C_j \in \text{parent_of}(C_i)} N_{i,j} \quad (3.2)$$

Chance of a configuration k denoted by $\text{chance}_{A_{i,k}}$ is then computed as

$$\text{chance}_{A_{i,k}} = \frac{\sum_{t=1}^{\text{sizeof}(CPT)} R_{A_{i,t}}}{1 + R_{A_{i,k}}} \quad (3.3)$$

The CPT value $P_{i,k}$ for the k^{th} row can then be computed as

$$P_{i,k} = \frac{\text{chance}_{A_{i,k}}}{\sum_{t=1}^{\text{sizeof}(CPT)} \text{chance}_{A_{i,t}}} \quad (3.4)$$

For our example illustration, dependency analyzer takes as input the original source code of appendix A and generates a model as shown in appendix B.4. Further detail will be explained with implementation details in the chapter 4

⁵This represents one row of the CPT

3.5 Translator

The translator transforms the program to incorporate remote capabilities. This component is dependent to a great extent, on the middleware being used. . Following are the main features of a translator.

- Translator makes code remote-enabled. It takes code targetted for a standalone system, and transforms the classes, generating the interfaces and middleware related code. Following code instrumentations are needed:
 - Object creation - Each `new()` has to be transformed to a query to run-time analyzer and followed by call to remote/local run-time system.
 - Method invocation - A function call will have to handle parameter passing modes or may have to make a previously local object remotely enabled etc. Similarly a return may need to create a remote reference to an existing local object
 - Method Return
- Limitations of the middleware e.g. those for RMI, highlighted in section 2.3.1 have to be handled. Code transformations may need to be made for the same. Code replacements Eg. Replace direct field accesses by getters and setters, package-scope access modifiers, handling `==`, `.equals` operator etc. In addition it also involves handling of some specific constructs Eg. `static`, synchronization, threads, inner classes, reflection etc.

Translation can be again done at source code level 2.4.4as well as bytecode level as in 2.4.2. The latter has the advantage that it can handle system classes as application classes except those using machine-dependent constructs. However, the techniques discussed above should be applicable to both. We do not consider machine-dependent classes. J-orchestra2.4.2 exploits all the details for the various combinations of modifiable and unmodifiable code and also discuss the solutions for the same. Code transformations from J-Orchestra can be added to a translator to increase the flexibility of object allocation amongst the hosts. For the purpose of demo, we have only included application class objects for distribution.

3.6 Run-time Analyzer and Query Engine

3.6.1 Object allocation decision

Runtime decision on object allocation has been motivated for us by following reasons:

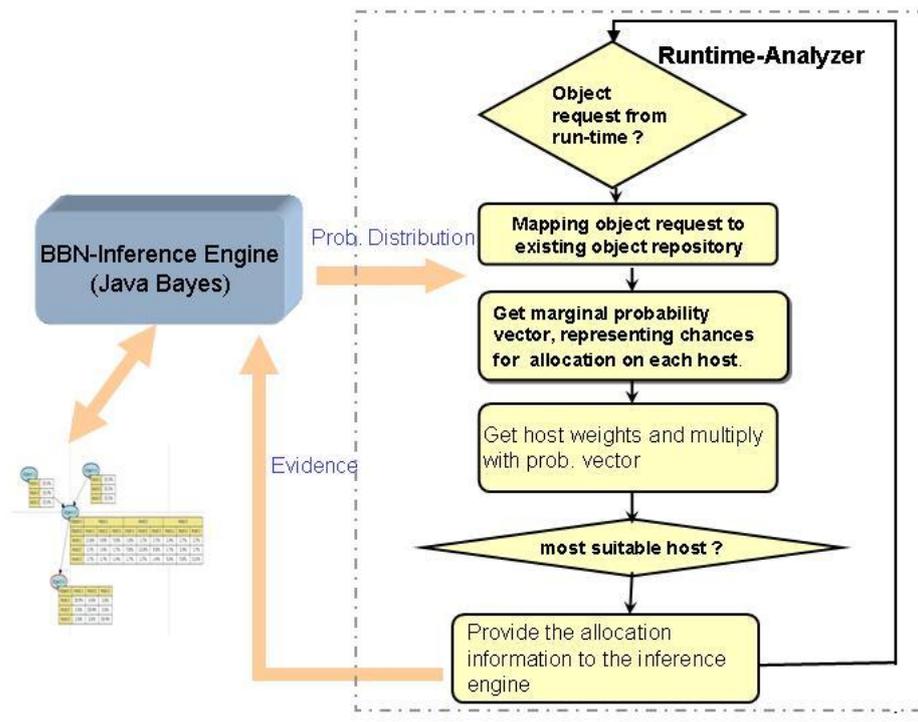


Figure 3.6: Runtime-Analyzer

- Postponing the final allocation decision increases the *flexibility* in allocation. It can help adjusting to varying load scenarios on the host nodes.
- It can also help adapting to situations like those of varying resource locations (E.g. Database Migration).
- Different control flows can lead to different sets of instantiations and hence, a more accurate placement, influenced by already allocated objects will be taken.

At run-time, we would like to use all decisions made regarding the components already instantiated, to be used as evidences. The load balancer feedbacks can be incorporated as *weights* of hosts. This updation can be an external feedback from load-balancer, which obviously would present the actual load condition of the system.

3.6.2 Functioning of Run-time Analyzer

We have used, JavaBayes [jav98]. The functioning of the *run-time analyzer* has been explained in Figure 3.6. This component plays the most important role in making the architecture flexible. The run-time analyzer as in our architecture is like a *central oracle*, for the system. It maintains the model learnt during the offline phase. Runtime analyzer, is queried by the run-time system 3.2 of each host whenever an object has to be created. Run-time analyzer uses the dependency model learnt by analyzer, propagates the current

allocation as evidence in the BBN and updates the structure. We can use a standard BBN Inferencing software. We have used JavaBayes [jav98]

3.7 Run-time System

We briefly explain the working of our runtime architecture with our illustration. As specified in section 3.3 there are three hosts available. Figure 3.7 illustrates the working of runtime. Runtime is a distributed component sitting on each of the hosts. For the purpose of illustration, we fix analyzer - the central oracle at Host-1. Suppose the main() of the program starts at Host-3. When Host-3 gets a request for new object, it invokes the analyzer on Host-1 in step-1, sending the current stack trace of the thread. Host-1 uses the runtime-analyzer and returns with some host say Host-2 in step-2. Following this, Host-3 invokes the runtime object factory at Host2 in step-3 that instantiates a new object of specified type and return a remote reference in step4.

Run-time system offers the necessary support during execution to get the program executed in distributed fashion.

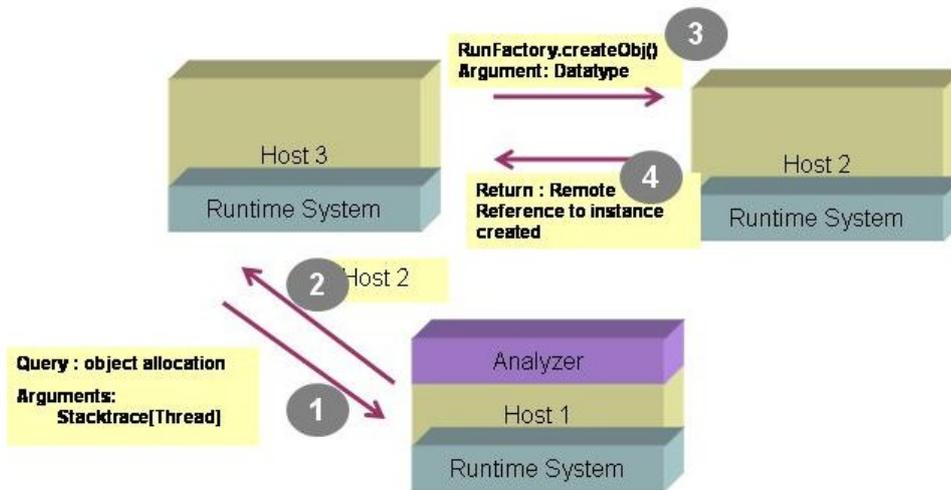


Figure 3.7: Illustration of the runtime

It is a *distributed* component present on each node where objects can be hosted.

- It interacts with the run-time analyzer, passing it necessary parameters like creator class, method, stacktrace of the thread, etc.
- Once the location of an object be decided, run-time provides *object factories*- one per node, binded as a well-known object. These factories handle creation requests for local as well as remote sites.

- It handles translation of references as the objects are passed as arguments to methods.
- Ideal run-time system will support object migration as well and support modes like - call by visit/move and copy
- It maintains the information needed like stacktrace of the thread over the distributed nodes in a distributed fashion.
- Run-time systems at all nodes act as peers, managing object creation and references.

3.8 Comparison with the Related Work

We expect that the above architecture would be more efficient and prove more flexible than the existing softwares. In this section we try to compare our proposed approach to existing mechanisms.

In literature [TS02], [Dah00],[PZ97], much importance has been given to the backend systems, that manage object references at runtime and provide support for migration.

Firstly, the analytical model that we propose using statistical approach is a novel idea. Most of the systems in literature as discussed in Section 2.4 do static analysis. Dynamic analysis has been done in Coign 2.4.3 with DCOM enabled code. We emphasize the significance of initial placement of the component. Most systems namely, [Spi02], [TS02] believe in user specifying the initial position of the component(object/class). Though this provides room for user customization, a truly *automated* partitioning system can actually do some *intelligent* work on this placement. We have identified some factors as discussed above and we incorporate them in our model.

Secondly, none of the existing systems we know, support run-time decision based allocation. The usual design of existing systems has been, to do some analysis (mostly class-based dependency), and generate an *allocation/partitioning* policy *offline*. Pangaea-Doorastha, section 2.4.4 and 2.4.4 for example, would let specification of each object as annotation within the code. However, this means that whenever a particular new instantiation has to be done it will be on a fixed host. J-Orchestra, section 2.4.2 on the other hand, tries best to make each object mobile(system class objects), so that it can be passed around as when instantiated from one host to other when functions are invoked. We argue that relying on migration is not a very good idea.

Most work has been on the backend front, where policy is highly dependent on user specification. The system closest to our design is Coign [Hun98]. However, Coign mainly designed for COM-based systems is mostly targetted for client-server architectures has other shortcomings as highlighted in Section 2.4.3. Our idea of object classification is influenced by Coign's *I3C* classifier.

Chapter 4

Implementation Details

We have shown the feasibility of this architecture, by implementing different blocks. This section highlights a few details of the implementation.

We have implemented the entire architecture of the decision analyzer, as discussed in Section 3.2. The translation algorithm is straightforward. We assume thus hand-coded instrumented code to be the input of our implemented run-time algorithm. The design of translator is discussed in Future extensions at the end of this chapter. The run-time analyzer uses a standard bayesian inferencing software. The run-time infrastructure, is a modification of Doorastha [Dah00]. It provides some features that our architecture proposes as a part of the run-time system. This has been discussed earlier with distributed infrastructures as explained in section 2.3. We have modified Doorastha run-time to provide basic run-time support.

4.1 Standalone Profiling

As discussed in Section 3.4, we use dynamic analysis or profiling for dependency analysis, but do it offline, with standalone code. We have used Java Virtual Machine Tool Interface [jvm04] with JNI support. Profiling works by code *instrumentation* and *event trapping*. In other words, jvmti provides us an API through which we can define our agents and execute necessary code on notification of different events like - JVM start/end. The code has partly been written in *Java* and partly in *C*. The Appendix B details further. Appendix B.1 is the Tracker class. The standalone code for profiling is instrumented to invoke methods of this class. The actual profiling which involves object tracing, method tracing, object-stacktrace maintenance and dependency matrix generation is done in *native* code.

The other options considered for profiling were - JVMPI [jvm], *java.lang.instrument* API [jdk96]. The former is now out-of-date and latter did not provide object tagging which C interface allows. There exist some profiling tools yajp [yaj], [ejp], but most of them provide class based information and not object based. Figure 4.1, shows the main logic used in the profiler. We have used bytecode instrumentation, to record method invocation and object creation events. The output of profiling is :

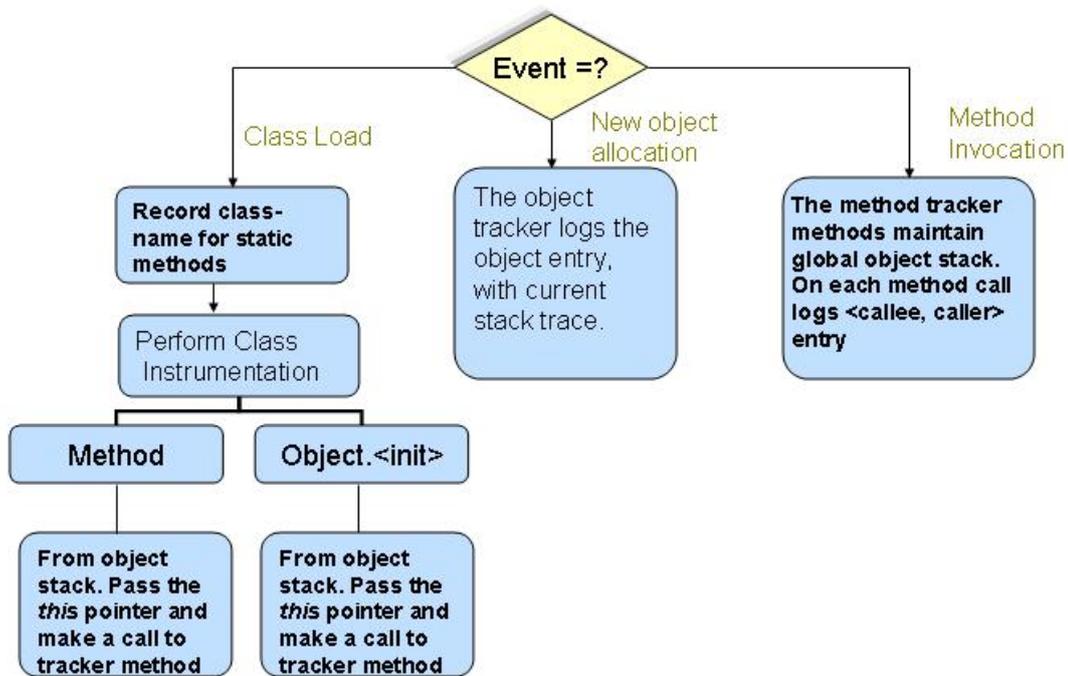


Figure 4.1: Profiling Flow

1. Object Information table - Unique stack traces encountered and their ids. Appendix B.3 shows the output of a sample run on the Motor car example.
2. Dependency Matrix - $M(caller, callee)$ is number of times object id *caller* invoked a method on *callee*. Appendix B.3.2 shows a part of the matrix learnt.

4.2 Coding the object model - BBN

For simple demo of the system, we have demonstrated Model learning for a particular flow of the application. We have implemented a simplistic model as discussed in section 3.4.4.1. The dependency data is just based on standalone profiling runs, and hence is independent of host knowledge.

However, as discussed in section 3.4.4.1, the run-time will query the model for best location. Thus, the learner has to have a small component which we call *BBNgenerator*. The *BBNgenerator*, computes different combinations on object allocation based on each parent and calculates the CPTs for each node. In our system with simplistic model, it implements the algorithm discussed in section 3.4.4.1. This coding is done in *C*. The Bayesian network learnt is represented in standard BIF (Bayesian Interchange Format)

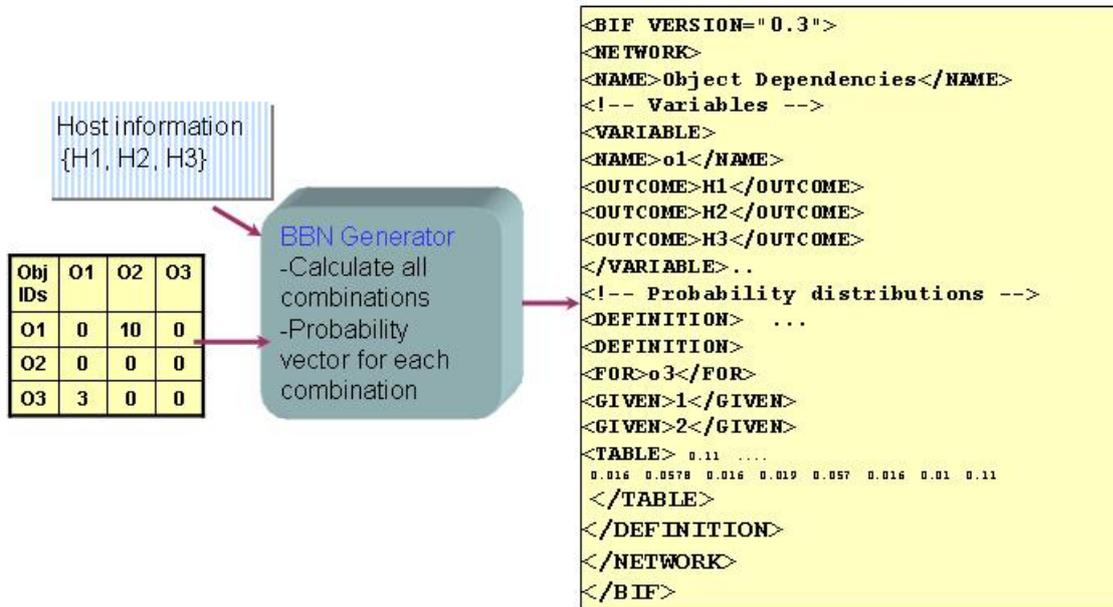


Figure 4.2: BBN Generator

[bif98].

4.3 Run-time Issues

We have modified the Doorastha [Dah00] to incorporate object-level partitioning run-time algorithm. We needed to add support for interfacing with the analyzer engine. We have added support for maintaining the thread stacktrace during the distributed execution, which is needed by the analyzer to index object creation requests in the object repository.

4.4 Extensions to the current Implementation

Here we highlight the designs of the components which could not be implemented, due to constrained time.

4.4.1 Code Translation and Instrumentation for Run-time

Translator has the responsibilities as stated in section 3.5. Translator needs to perform the following functions broadly. Appendix C shows the translated code - supporting

classes and modified original class, which is expected by the runtime system as input.

1. Instrument each class with additional instant fields, to store - stacktrace at the time of creation and host-id representing the host on which instance is hosted.
2. Needs to generate extra supporting classes needed for the implementation of proxy-based approach 2.3.1 and stacktrace maintenance.
3. Each *new()*, has to be replaced, with a call to Analyzer, and then the code for instantiation on required host, for obtaining proxy etc.
4. Method return needs to be modified to check if the reference obtained in return is a remote reference of a local object.
5. Routine checks and transformations, to ensure there is no violation of RMI semantics 2.3.1.

Doorastha [Dah00] handles a few of these. The implementation of translator can be done using a standard bytecode analysis library like *Barat* [BS98]. Barat represents the entire program in the form of an AST¹ structure. Each Class, Method, code blocks (object-allocation, etc), are represented as the subclass of a main base class Node.

Barat implements visitor pattern 4.3 whereby, each program element is represented as a Node, overriding *accept()* method. This *accept()* method in turn invokes all the methods related to sub-nodes (methods, fields etc), implemented in Visitor passed as reference to *accept*. It then provides some abstract visitor classes, which can be extended to write a visitor of own. It provides handle to every programming construct needed, very easily. Using these hooks one can transform the code. It then, also provides support for code generation.

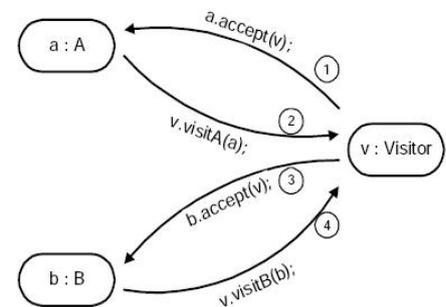


Figure 4.3: An example visitor pattern.

¹Abstract Syntax Tree

Chapter 5

Conclusion and Future Work

Application Partitioning is a promising concept, significantly useful for distributed computing. An automated partitioning software, handling component allocation and relocation transparently, is certainly desirable. With increasingly programs becoming larger, complicated, time-consuming, a highly flexible architecture, like the one proposed by us, would lead to better utilization of idle computer times and load distribution.

We have classified the existing partitioning mechanisms 2.3, and subsequently proposed a flexible partitioning architecture. Our key contributions include

- Detailed study and classification of existing partitioning mechanisms
- Proposing a complete architecture for application partitioning, emphasizing on placement of components.
- We have designed and implemented a Decision Analyzer based on statistics, to model component dependence and independence relationships. At run-time, this model also incorporates already allocated components as evidences and progressively provides dependency based inference for new allocations
- Our dynamic placement strategy, postpones the component allocation decisions to run-time, making this architecture the most flexible of all.
- Our model is flexible enough to incorporate factors like load-balancing when deciding the allocation at run-time.
- We have also modified existing Doorastha source code to use as a distributing infrastructure for our partitioning scheme.

This however, is just a very preliminary implementation of a large system to show the feasibility of the whole architecture.

We would like to suggest a few enhancements to our implementation

- Program analysis can be further researched in detail, and a machine learning algorithm incorporating different profiling runs, based on the belief network can be modelled. This algorithm would use each dependence matrix obtained from profiling as a different BBN structure. The learning algorithm having lots of profiling data would then could be a heuristic structure search.

- Though the run-time analyzer algorithm has been implemented as a centralized entity, if one could propagate the encoded BBN probabilities values(it is an XML File), with function calls, we can surely make the system distributed. However, this has to be implemented carefully and related issues may need to be further handled at run-time.
- Garbage collection mechanisms have to be taken care of.
- In current implementation, though we decide at run-time where the allocation has to be made for the first time when object request for a particular object, associated with stacktrace arrives, subsequent allocations would go to the same node. This limitation can be overcome if we can keep track of live and dead objects, and flush the values in the BBN accordingly.
- A more sophisticated BBN algorithm can also incorporate network topology details and can learn probability vectors accordingly. A more complicated formula can be implemented in BBN generator, considering sizes of copyable objects.
- We have not considered migration in any of the analysis or run-time algorithms. Effect of migration, on the overall placements has to be looked into. Object sizes encountered in profiled runs can be used to decide whether migration has to be made and if so the migration modes.
- A thorough performance analysis could be done, with different load distributions as input and seeing the number of objects created parameter vary on the different hosts.
- A design for translator has been suggested, can be fully implemented

Object-Level Partitioning, emphasizes on careful initial placements, based on the information from existing allocations and dependency characteristics, not leaving the load configurations. On one hand, O-L partitioning gives us benefits of utilizing idle systems on the network and getting a standalone program executed in distributed settings, it certainly also poses a lot of overhead. As the application size and subsequently the size of the components increase, these overheads become insignificant and benefits of O-L partitioning can be realized.

Abbreviations and Notations

Abbreviations

- OO : Object Oriented
- O-L : Object Level
- RMI : Remote Method Invocation
- BBN : Bayesian Belief Network
- CPT : Conditional Probability Table

Bibliography

- [BBN98] An introduction to bayesian networks and their contemporary applications, 1998. <http://www.niedermayer.ca/papers/bayesian/>.
- [bif98] The interchange format for bayesian networks, 1998. <http://www.cs.cmu.edu/~fgcozman/Research/InterchangeFormat/>.
- [Bou94] Remco Bouckaert. Properties of bayesian belief network learning algorithms. In *Proceedings of the 10th Annual Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 102–10, San Francisco, CA, 1994. Morgan Kaufmann.
- [BS98] Boris Bokowski and Andre Spiegel. Barat-a front-end for java. Technical Report tr-b-98-09.pdf, Freie University Berlin, 1998. <ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-98-09.pdf>.
- [Dah00] Markus Dahm. Doorastha: a step towards distribution transparency. In *Proceedings of the Net.Object Days 2000*, 2000.
- [ejp] Extensible java profiler. <http://ejp.sourceforge.net/>.
- [HS99] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *OSDI'99: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 1999. Springer-Verlag.
- [Hun98] Galen C. Hunt. *Automatic Distributed Partitioning of Component-Based Applications*. PhD thesis, University of Rochester, Rochester, New York, 1998.
- [jav98] Javabayes - version 0.346, bayesian networks in java, 1998. <http://www.cs.cmu.edu/~javabayes/Home/>.
- [jdk96] Java 5.0 specification standard, 1996. <http://java.sun.com/j2se/1.5/index.html>.
- [jvm] Jvm profiling interface. <http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html>.
- [jvm04] Jvm tool interface (jvm ti), 2004. <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html>.

- [PZ97] Michael Philippsen and Matthias Zenger. Javaparty transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.
- [RMI96] Rmi standard, 1996. <http://java.sun.com/j2se/1.5/docs/guide/rmi/spec/rmiTOC.html>.
- [Spi02] Andre Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, Freie University Berlin, 2002.
- [TS02] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 178–204, London, UK, 2002. Springer-Verlag.
- [TSCI01] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A byte-code translator for distributed execution of “legacy” Java software. *Lecture Notes in Computer Science*, 2072:236–243, 2001.
- [yaj] Yet another java profiler. <http://yajp.sourceforge.net/>.

Appendix A

Example

```
/* Class Motors: Has two types of vehicles - Car and Scooter*/
public class Motors {
    CarModel car;
    Scooter scooter;
    Motors(){
        scooter = new Scooter ();
        car = new CarModel ();
    }
    public void rollCarWindows(){
        Window w = car.getLeft().getWindow ();
        for(int i =0; i < 100;i++){
            w.rollup ();
            w.rolldown ();
        }
    }
    public void simUsages() {
        car.simulateCarUsage ();
        scooter.simulateScooterUsage ();
    }
    public static void main(String [] args){
        Motors motors=new Motors ();
        motors.rollCarWindows ();
        motors.simUsages ();
    }
}

/* Class Car: Is a vehicle in class Motor*/
public class CarModel{
    /* Required car parts: 1 Engine, 4 wheels, and 2 doors */
    private Engine engine;
    private Wheel wheel1, wheel2, wheel3, wheel4;
    private Door left, right;
```

```

public CarModel_original() {
    engine = new Engine();
    left = new Door();
    right = new Door();
    wheel1 = new Wheel();
    wheel2 = new Wheel();
    wheel3 = new Wheel();
    wheel4 = new Wheel();
}
/* Simulates car usage */
public void simulateCarUsage() {
%    engine.start();
    engine.rev();
    wheel1.align();
    engine.stop();
}
}
/* Class Car: Is a vehicle in class Motor*/
public class Scooter {
    private Engine engine;
    private Wheel wheel1, wheel2;
    public Scooter() {
        engine = new Engine();
        wheel1 = new Wheel();
        wheel2 = new Wheel();
    }
    public void simulateScooterUsage() {
        engine.start();
        engine.rev();
        for(int i=0;i<25;i++)
            wheel1.align();
        wheel2.align();
        engine.stop();
    }
}
}
/* Engine: classes used to model car/scooter parts */
class Engine {
    public void start() {
        System.out.println("Start the car.");
    }
}

```

```
    }  
    public void rev() {  
        System.out.println("Rev_the_engine.");  
    }  
    public void stop() {  
        System.out.println("Car_stopped.");  
    }  
}  
class Door {  
    private Window window = new Window();  
    public Window getWindow() {  
        return window;  
    }  
    public void setWindow(Window window) {  
        this.window = window;  
    }  
    public void open() {  
        System.out.println("Open()");  
    }  
    public void close() {  
        System.out.println("Close()");  
    }  
}  
class Window {  
    public void rollup() {  
        System.out.println("Rollup_the_window.");  
    }  
  
    public void rolldown() {  
        System.out.println("Rolldown_the_window.");  
    }  
}
```

Appendix B

Instrumentation and Profiling

B.1 Tracker Class

```
/** Java class to hold static methods which will be called in byte  
 *code injections of all class files.  
 */  
 */  
public class Tracker {  
    private static int engaged = 0;  
    /* Native method that does profiling, tags the object with  
 *instantiation history(current stacktrace)  
 */  
    private static native void _newobj(Object thread, Object o);  
    /** At the start of constructor of Object class i.e. <init> method  
 * a call to following method is injected.  
 * @param o : reference of object  
 */  
    public static void newobj(Object o)  
    {  
        if ( engaged != 0 ) {  
            _newobj(Thread.currentThread(), o);  
        }  
    }  
    /* This method does the object-stacktrace maintenance in native code,  
 *pushing new stackrace frames  
 */  
    private static native void _method_entry(Object thr, Object obj);  
    /** At the very beginning of every method, a call to method_entry()  
 * is injected  
 * @param o : reference of object  
 */  
    public static void method_entry(Object obj)  
    {
```

```

        if (engaged != 0 ) {
            _method_entry( Thread.currentThread(), obj);
        }
    }
    /* This method does the object-stacktrace maintenance in native code,
     * popping stacktrace frames if needed
     */
    private static native void _method_exit(Object thr);
    public static void method_exit()
    {
        if ( engaged != 0 ) {
            _method_exit( Thread.currentThread());
        }
    }
}

```

B.2 Profiling Implementation

This section just shows some snippets from the native implementation of profiling. We are attaching the most important method-tracing profiling code, which shows object-trace maintenance.

```

#define M_native_entry _method_entry
/* The method entry and exit codes try to maintain a global object
 * stack trace, gdata->objStackw, which keeps on top the current
 * object on which method is invoked. It makes entries in the dependency
 * matrix.
 */
static void M_native_entry(JNIEnv *env, jclass klass,
                           jobject thread, jobject obj) {
    jvmtiError error;
    enterCriticalSection(gdata->jvmti); {
    Stack * objStack=gdata->objStack;
    jvmtiEnv *jvmti;
    jvmti=gdata->jvmti;
    jvmtiError err;
    ObjectTag * invokerTag;
    ObjectTag * invokedTag;
    if( !(gdata->vmInitialized) || (gdata->vmDead) ){
        return;
    }
}

```

```

}
/*Get the stack top to get invoker tag*/
invokerTag = objStack->getTop(objStack);
if(obj != NULL) {
    jlong size_ptr , tag_ptr , caller_tag_ptr;
    TraceInfo * tinfo;

    jvmtiError error;
    /* Retrieve the tag for the object*/
    error= (*jvmti)->GetTag(jvmti , obj , &tag_ptr);
    /*tag_ptr is tagInfo typecasted to long*/
    if(! tag_ptr) {
        return;
    }
    tinfo = (TraceInfo * )(void *)(ptrdiff_t)(tag_ptr);
    /*check for private function/ same class/obj
    *funcion invocation*/
    if(invokerTag !=NULL && invokerTag->id == tinfo->id) {
        incrementStackCount(invokerTag);
        return;
    }
    /* Going further means call is not
    *internal to the object. Create the tag object
    */
    invokedTag=createTag(tinfo->id , OBJECT_CALL);
}
else /*obj == null implies static method invoked*/{
    jvmtiFrameInfo frames[MAXSTACKFRAMES];
    jint count;
    err = (*jvmti)->GetStackTrace(jvmti , thread , 0,
                                MAXSTACKFRAMES, frames , &count);
    if (err == JVMTIERROR_NONE && count >= 2) {
        int i;
        char *methodName;
        char *declaringClassName;
        jclass declaring_class;
        /*Frame Descriptions : frame[0]: method_entry (from Tracker)
        *frame[1]: naive method_entry
        *frame[2]: the invoked method
        *frame[3]: caller method

```

```

*/
err = (*jvmti)->GetMethodName(jvmti, frames[2].method,
                              &methodName, NULL, NULL);
if (err == JVMTLERROR_NONE) {
    ClassInfo * cinfo;
    /*get the class sig of the method being invoked*/
    err = (*jvmti)->GetMethodDeclaringClass(jvmti,
                                              frames[2].method, &declaring_class);
    CHECK_JVMTLERROR(jvmti, err, "cannot_get_the_Method"
                    "Declaring_class");

    /*retrieenve id from hashClasses table*/
    err = (*jvmti)->GetClassSignature(jvmti, declaring_class,
                                      &declaringClassName, NULL);
    CHECK_JVMTLERROR(jvmti, err, "cannot_get_the_class"
                    "_signature");
    cinfo = lookupClassInfo(declaringClassName);
    /*check for private function/ same
     * class/obj function invocation
     */
    if(invokerTag != NULL && invokerTag->id == cinfo->id)
    {
        incrementStackCount(invokerTag);
        return;
    }
    /*Create the tag object*/
    invokedTag=createTag(cinfo->id, STATIC_CALL);
}
else {
    CHECK_JVMTLERROR(jvmti, err, "cannot_get_method_name");
}
deallocate(jvmti, methodName);
deallocate(jvmti, declaringClassName);
}
else{
    CHECK_JVMTLERROR(jvmti, err, "cannot_get_stack_trace");
}
}
/* Push the object on which method was called on the stack*/
incrementStackCount(invokedTag);

```

```
objStack->push(objStack, invokedTag);
if(invokerTag != NULL) {
    /*Make entry in id table for invokerTag,invokedTag*/
    gdata->objectDependencies[invokerTag->id][invokedTag->id]++;
}
} exitCriticalSection(gdata->jvmti);
}
```

B.3 Profiling Output

B.3.1 Object Info

```

Index= 0: Type:Class Name:edu/iitb/olpartitioning/example/Motors
Signature:Ledu/iitb/olpartitioning/example/Motors;

Index= 1: tnum= 1
stack=(Ljava/lang/Object;.<init>@1,Ledu/iitb/olpartitioning/example/Motors;
.<init>@1,Ledu/iitb/olpartitioning/example/Motors;.main@8) nframes=3

Index= 2: Type:Class Name:edu/iitb/olpartitioning/example/Scooter
Signature:Ledu/iitb/olpartitioning/example/Scooter;

Index= 3: tnum= 3
stack=(Ljava/lang/Object;.<init>@1,Ledu/iitb/olpartitioning/example/Scooter;.<init>
>@1,
Ledu/iitb/olpartitioning/example/Motors;.<init>@9,Ledu/iitb/olpartitioning/example
/Motors;.main@8) nframes=4

Index= 4: Type:Class Name:edu/iitb/olpartitioning/example/Engine
Signature:Ledu/iitb/olpartitioning/example/Engine;

Index= 5: tnum= 5
stack=(Ljava/lang/Object;.<init>@1,Ledu/iitb/olpartitioning/example/Engine;.<init>@
@1,Ledu/iitb/olpartitioning/example/Scooter;.<init>@9,Ledu/iitb/olpartitioning/exa
mple/Motors;.<init>@9,Ledu/iitb/olpartitioning/example/Motors;.main@8) nframes=5

Index= 6: Type:Class Name:edu/iitb/olpartitioning/example/Wheel
Signature:Ledu/iitb/olpartitioning/example/Wheel;

Index= 7: tnum= 7
stack=(Ljava/lang/Object;.<init>@1,Ledu/iitb/olpartitioning/example/Wheel;.<init>@
1,Ledu/iitb/olpartitioning/example/Scooter;.<init>@20,Ledu/iitb/olpartitioning/exa
mple/Motors;.<init>@9,Ledu/iitb/olpartitioning/example/Motors;.main@8) nframes=5

Index= 8: tnum= 8
stack=(Ljava/lang/Object;.<init>@1,Ledu/iitb/olpartitioning/example/Wheel;.<init>@
1,Ledu/iitb/olpartitioning/example/Scooter;.<init>@31,Ledu/iitb/olpartitioning/exa
mple/Motors;.<init>@9,Ledu/iitb/olpartitioning/example/Motors;.main@8) nframes=5

Index= 9: Type:Class Name:edu/iitb/olpartitioning/example/CarModel
Signature:Ledu/iitb/olpartitioning/example/CarModel;

Index=10: tnum=10
stack=(Ljava/lang/Object;.<init>@1,Ledu/iitb/olpartitioning/example/CarModel;.<ini
t>@1,Ledu/iitb/olpartitioning/example/Motors;.<init>@20,Ledu/iitb/olpartitioning/e
xample/Motors;.main@8) nframes=4

Index=11: tnum=11
stack=(Ljava/lang/Object;.<init>@1,Ledu/iitb/olpartitioning/example/Engine;.<init>
@1,Ledu/iitb/olpartitioning/example/CarModel;.<init>@9,Ledu/iitb/olpartitioning/ex
ample/Motors;.<init>@20,Ledu/iitb/olpartitioning/example/Motors;.main@8) nframes=5

Index=12: Type:Class Name:edu/iitb/olpartitioning/example/Door
Signature:Ledu/iitb/olpartitioning/example/Door;

Index=13: tnum=13
stack=(Ljava/lang/Object;.<init>@1,Ledu/iitb/olpartitioning/example/Door;.<init>@1
,Ledu/iitb/olpartitioning/example/CarModel;.<init>@20,Ledu/iitb/olpartitioning/exa
mple/Motors;.<init>@20,Ledu/iitb/olpartitioning/example/Motors;.main@8) nframes=5

```

Figure B.1: Tagged object information extract

B.3.2 Dependency Matrix

We just show few rows of the dependency matrix obtained for the Motor example.

```

0-0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1-0,0,0,1,0,0,0,0,0,0,2,0,0,1,0,200,0,0,0,0,0
2-0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
3-0,0,0,0,0,3,0,25,1,0,0,0,0,0,0,0,0,0,0,0,0
4-0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
5-0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

```

Figure B.2: Dependency Matrix

B.4 Model

The BBN model learnt for the Motos example

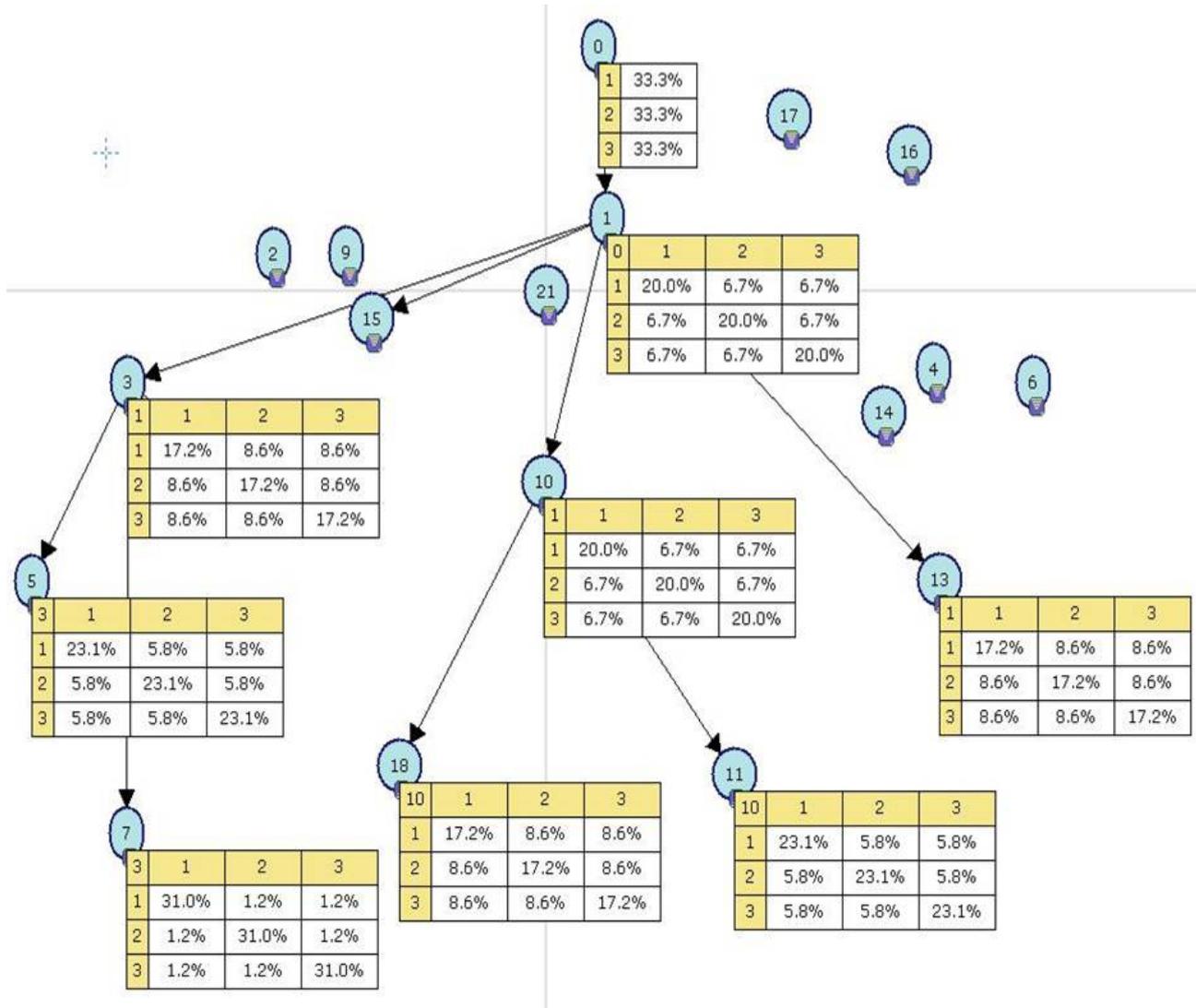


Figure B.3: Dependency Matrix

Appendix C

Example - Translated Code

We are just showing the translated code that was used at run-time, for CarModel class as discussed in chapter A.

C.1 Supporting Proxy class

This is instantiated at the remote caller and maintains a remote reference to a wrapper object hosted at a remote host. For more syntactical details please refer to [Dah00]

```
public class CarModel_Proxy extends CarModel implements ProxyMigrateable {
    /* Required car parts: 1 Engine, 4 wheels, and 2 doors */
    private CarModel_Interface $$instance;
    private Ident    $$id;
    private boolean    $$is_migrateable;
    CarModel_Proxy(CarModel_Interface a, Ident id) {
    super((doorastha.Dummy) null);
    $$instance = a;
    $$id       = id;
    }

    /* Simulates car usage */
    public void simulateCarUsage() {

    do {
        try {
            $$instance.$simulateCarUsage();
            return;
        } catch(RemoteException e) {
            RuntimeSystem.DEBUG(e); return;
        } catch(MovedException e) {
            $$adjustTo(e); }
    } while(true);
}
```

```

    }
}

```

C.2 Supporting Wrapper class

This is instantiated at the remote site where source object is instantiated. It keeps a local reference to source object and exports remote reference to callers at remote sites. For further details refer to [Dah00].

```

public class CarModel_Wrapper extends Object_Wrapper
implements CarModel_Interface , WrapperMigrateable {
    /* Required car parts: 1 Engine, 4 wheels, and 2 doors */
    private CarModel $$carmodel;
    public CarModel_Wrapper(CarModel a) {
        super(a);
        $$carmodel = a;
    }
    public CarModel_Wrapper() {
        // needed when used as a remote factory
        super(null);
    }
    public CarModel $$createCarModel() {
        // Called from remote site for remotenew
        CarModel_Proxy proxy = (CarModel_Proxy)new CarModel().
                                globalize();
        proxy.$$isMigrateable(true);
        // Set flag. Flag is not set by normal creation
        return proxy;
    }
}

```

C.3 Instrumented CarModel class

Original class has its methods and object creation statements re-written. Enclosing the code of the constructor showing how each `new()` object creation gets translated.

```

public CarModel {
public CarModel() {

        Engine engine;

```

```

Door door;
location = AnalyzerEngine.getOptimalLocation
    ("edu.iitb.olpartitioning.example.Engine");
if(location.compareTo(NetworkSystem.host_address)
    == 0 )/*If local host*/
    engine = new edu.iitb.olpartitioning.
        example.Engine();
else /*remote host*/
    engine = edu.iitb.olpartitioning.example.
        Engine_Proxy.$$createEngine(location);

location = AnalyzerEngine.getOptimalLocation
    ("edu.iitb.olpartitioning.example.Door");
if(location.compareTo(NetworkSystem.host_address) == 0 )
    left = new edu.iitb.olpartitioning.
        example.Door();/*If local host*/
else/*remote host*/
    left = edu.iitb.olpartitioning.
        example.Door_Proxy.$$createDoor(location);

}
}

```

Other objects are instantiated in a similar way.

Acknowledgements

I would like to express my gratitude to my advisor **Prof. Sridhar Iyer**. I am indebted to him for his guidance, encouragement, support and trust in me. I would like thank my senior **Vikram Jamwal**, Research Scholar, KReSIT for his valuable feedback and suggestions.

I would also like to thank my family and friends especially my entire M.Tech Batch, who have been a source of encouragement and inspiration throughout the duration of the project.

Last but not the least, I would like to thank the entire KReSIT family for making my stay at IIT Bombay, a memorable one.

Anshu Veda
KReSIT,
IIT Bombay
July 12, 2006

