

A GENERIC FRAMEWORK FOR DISTRIBUTED COMPUTATION

By

Shailesh Kelkar

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

Approved:

Carlos Varela
Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

July 2004
(For Graduation August 2004)

CONTENTS

LIST OF TABLES	v
LIST OF FIGURES	vi
ACKNOWLEDGMENTS	x
ABSTRACT	xi
1. Introduction	1
1.1 Issues in developing distributed applications	1
1.2 A Generic Framework for Distributed Computing	3
1.2.1 The Generic Programming Methodology Approach	5
1.2.2 Programming Language Level	6
1.2.3 Library Level	7
1.2.4 Run-Time Level	8
1.3 Evaluation Metrics	8
2. Generic Framework for Distributed Computing (GFDC)	10
2.1 Overview of GFDC	10
2.2 GFDC Architecture	12
2.2.1 Resource Level	12
2.2.2 Resources Managers Level	14
2.2.3 Work Distribution Level	14
2.2.4 Application Partitioning Level	14
2.2.5 Distributed Algorithms and Data Structures Level	15
2.2.6 User Application Level	15
3. Implementation	16
3.1 Overview	16
3.2 Resources	16
3.2.1 Resource	17
3.2.2 Node	18
3.2.3 Theater	18
3.3 Resource Managers	19
3.3.1 Resource Management	20
3.3.2 Static Resource Manager	20
3.3.3 Dynamic Resource Manager	21

3.4	Resource Allocation	22
3.4.1	ResourceAllocation	23
3.4.2	Round Robin	24
3.4.3	Priority	24
3.4.4	RandomAllocation	25
3.4.5	SemiRandom	25
3.5	Work Distribution	26
3.5.1	Farmer Worker	26
3.5.2	Hierarchical	28
3.6	Application Partitioning	29
3.6.1	Alternate	29
3.6.2	Sub Part	29
4.	Applications	30
4.1	Overview	30
4.2	Sample Application Implementations	32
4.2.1	Most Productive Turing Machine	32
4.2.2	Distributed Search	36
5.	Analysis	40
5.1	Overview	40
5.2	Analysis of SALSA with GFDC	40
5.2.1	SALSA Architecture	40
5.2.2	Enhanced SALSA Architecture with GFDC	42
5.2.3	Comparison of Two Approaches Using Evaluation Criteria	44
6.	Related Work	46
6.1	Overview	46
6.2	Programming Languages	46
6.2.1	C++	46
6.2.2	Java	47
6.2.3	SALSA	48
6.2.4	Discussion	50
6.3	Generic Libraries	50
6.3.1	Standard Template Library (STL)	51
6.3.2	Boost Graph Library (BGL)	51
6.3.3	Loki	51

6.3.4	Discussion	52
6.4	Run-Time Systems	52
6.4.1	Globus	52
6.4.2	Java Virtual Machine	53
6.4.3	World-Wide Computer	54
6.4.4	Internet Operating System	54
6.4.5	Discussion	55
7.	Future Work	56
7.1	Development of Abstractions for GFDC	56
7.2	Implementation Of All Abstractions Of GFDC	56
7.3	Implementation of Generic SALSA	57
7.4	Implementation of GFDC in other Languages	59
8.	Conclusions and Contributions	60
8.1	Conclusion	60
8.2	Contribution	61
	LITERATURE CITED	62

LIST OF TABLES

5.1	Table comparing the relative advantages and disadvantages of implementing the Most Productive Turing Machine (MPTM) in SALSA as opposed to SALSA with Generic Framework for Distributed Computation (GFDC). From the comparisons it is clear that the SALSA with GFDC is easier and faster to code. The program size is small, and programmer does not need to know the details of the distributed computing issues.	44
-----	--	----

LIST OF FIGURES

1.1	Distributed computing issues addressed at three levels. The <i>Programming Language Level</i> addresses the issues related to support for synchronization, migration, and generic programming. The <i>Library Level</i> addresses the work distribution strategies and development of libraries implementing distributed algorithms and data structures. The <i>Run-time Level</i> address the distributed computing specific issues like dynamic reconfiguration, profiling, load balancing and resource management.	4
2.1	Generic Framework for Distributed Computation. In this multi-level framework, various abstractions like resources, resource management and allocation, work distribution, distributed data structures, and algorithms are identified. It also presents the interaction between these levels within the architecture, and also, the interaction of the user application with the framework.	13
3.1	UML class hierarchy of different types of Resources. Here Resource is the abstract base class for various types of resources. Node and Theater are the two sample derived classes.	17
3.2	The Resource class is the abstract base class for classes that are part of the resource level. It has the variables to completely describe the resource and the methods required to act on them.	17
3.3	The Node class is the specialization of the Resource class. It has the overloaded constructors to accept resources that can be partially or fully described.	18
3.4	The Theater class is the specialization of the Resource class. It has an additional variable UAL to store the UAL of the agent in the <i>SALSA/WWC</i> programming paradigm. It also has methods to act on the UAL variable and overloaded constructors, to accept resources that can be partially or fully described.	18
3.5	UML class hierarchy of different types of resource managers. Here <i>Resource Management</i> is the abstract base class for various types of resources managers. <i>Static Resource Manager</i> and <i>Dynamic Resource Manager</i> are the two sample derived classes.	19
3.6	The ResourceManagement class is the abstract base class for the classes that implement the <i>Resource Management</i> level. It is parameterized for the type resources that it manages. It has the variables and methods to keep track of the resource that it is managing.	20
3.7	The StaticResourceManager class extends the abstract ResourceManagement class and is parameterized for the type resources that it manages. It has the constructor that accepts and array of resources that it will manage during the computation.	21

3.8	The <code>DynamicResourceManager</code> class extends the abstract <code>ResourceManagement</code> class and is parameterized for the type resources that it manages. It resources that it manages can change during the computation. It has method to get resources from server at run-time.	21
3.9	UML class hierarchy of different types of <i>Resource Allocations</i> . Here <i>Resource Allocation</i> is the abstract base class for various types of resources allocators. <code>RoundRobin</code> , <code>RandomAllocation</code> , <code>Priority</code> and <code>SemiRandom</code> are the sample derived classes.	22
3.10	The <code>ResourceAllocation</code> class is the abstract base class for the classes that implement the <i>Resource Allocation</i> level. The class has variable to keep track of the total and next resource to be allocated and the methods to act on those variables. It has an abstract method <code>getNext()</code> which must be implemented by all the concrete derived classes.	23
3.11	The <code>RoundRobin</code> class implements the <code>getNext()</code> method so that the resource allocation is based on round-robin manner.	24
3.12	The <code>Priority</code> class implements the <code>getNext()</code> method so that the resource allocation is based on the priorities of the resources.	24
3.13	The <code>RandomAllocation</code> class implements the <code>getNext()</code> method so that the resource allocation is done in a random manner.	25
3.14	The <code>SemiRandom</code> class implements the <code>getNext()</code> method so that the resource allocation is done in a semi-random manner. In such a case, one or more special resources are allocated by special requests and the remaining resources are allocated randomly.	25
3.15	The <code>FarmerWorker</code> class implements the <i>FarmerWorker</i> work distribution strategy. C1: <code>StaticResourceManager</code> is declared for resource type <code>Theater</code> . C2: <code>ResourceAllocation</code> variable <code>ra</code> is declared. C3: An array of <code>Theater</code> is created and each theater is initialized. C4: <code>StaticResourceManager</code> is defined with array of <code>Theater</code> from C3. C5: <code>ResourceAllocation</code> <code>ra</code> is assigned an object of type <code>RoundRobin</code> . C6: According to the <code>ResourceAllocation</code> <code>ra</code> a new resource of the type <code>Theater</code> is requested from <code>StaticResourceManager</code> <code>srm</code> . C7: <i>Worker</i> is added to the list of workers.	27
3.16	The <code>Hierarchical</code> class implements the <i>Hierarchical</i> work distribution strategy. C1: <code>StaticResourceManager</code> is declared for resource type <code>Theater</code> . C2: <code>ResourceAllocation</code> variable <code>ra</code> is declared. It also implements the required methods for managing the resources.	28
3.17	The <code>Alternate</code> class implements the <i>Alternate</i> work distribution policy.	29
3.18	The <code>SubPart</code> class implements the <i>Sub Part</i> work distribution policy.	29

4.1	The <code>MostProductiveTuringMachine</code> class implements the <i>MostProductiveTuringMachine</i> application. C1: An instance of <i>Farmer Worker</i> work distribution strategy is created. C2,C3,C4,C7: Various methods of <code>FarmerWorker</code> are called. C5: <code>FarmerWorker</code> is initialized. C6: <code>FarmerWorker</code> is called to create required number of workers.	34
4.2	Result obtained from the execution of the <i>Most Productive Turing Machine</i> application. The three windows on the right hand side are the worker theaters that display messages about the status of computation and result. The window on the top left hand side shows the computation from the perspective of the farmer. The farmer displays that the results. The naming server is seen in the bottom right window. The application scripts are run by the user in the bottom left window.	35
4.3	The <code>DistributedSearch</code> class implements the <i>DistributedSearch</i> application. C1: An instance of <code>FarmerWorker</code> work distribution strategy is created. C2,C3,C4,C5,C8: Various methods of <code>FarmerWorker</code> class are called. C6: <code>FarmerWorker</code> is initialized. C7: <code>FarmerWorker</code> is called to create required number of workers.	38
4.4	Result obtained from the execution of the <i>Distributed Search</i> application. The three windows on the right hand side are the worker theaters that display messages about the status of computation and result. Here one of the workers found the element while others did not. The window on the top left hand side shows the computation from the perspective of the farmer. The farmer displays that the element was found as one of the workers found it. The naming server is seen in the bottom right window. The application scripts are run by the user in the bottom left window.	39
5.1	SALSA Architecture. In this architecture the application developer has to code all the things required for distributed computation along with the application itself. The application programs are then compiled by the SALSA compiler and combined with the SALSA actor library to give the Java bytecode, which is then executed.	41
5.2	Enhanced SALSA Architecture with GFDC. In this architecture, the application developer only has to code the things required by the application. The application programs are then compiled by the SALSA compiler, and combined with the SALSA actor library and <i>Generic Framework for Distributed Computation</i> implementing libraries, which take care of distributed computing issues. The programmer then executes the generated class files.	43
6.1	An actor upon receiving a message either (1) Changes it's internal state, (2) Migrate to a new location, (3) Send messages to other Actors or (4) Create new Actors.	49

7.1	Programming using Generic SALSA Libraries. In this architecture, the programmer will be able to write generic code in SALSA and use the generic SALSA code from the libraries. The generic SALSA compiler will compile the code into generic Java. The Java compiler will combine this code with the generic Java code developed to give a class file that can be executed. The programs in this architecture will be smaller, simpler, and more efficient, with less development time.	58
-----	---	----

ACKNOWLEDGMENTS

I would like to thank Professor Carlos Varela for advising and guiding me on this thesis. This thesis is the result of the interactions that I had with him in various research meetings, courses and independent study. I would like to thank him for his guidance and encouragement.

I would like to thank the past and present members of the *Worldwide Computing Research* lab in *Computer Science Department* at RPI. It was indeed a pleasure to have been associated with all them. The discussions and suggestions that I had with them were very constructive and helpful for the thesis.

I would like to thank Professor David Musser for his course *Generic Software Design*, which introduced me to *Generic Programming*. I would also like to thank him and Mayuresh for some discussions we had related to this thesis. I would also like to thank Krishna for his expert advice on *LaTeX*.

On a personal note, I would like to thank my parents Dr. Ramesh Kelkar and Roopali Kelkar, sister Kalpana Kelkar and her husband Ameya Agnihotri for their encouragement and motivation. Words can't describe the love and inspiration that they have given to me. This thesis is dedicated to them.

I would like to thank all my friends and relatives for the good times, help and support. I would also like to thank all my teachers, fellow students and former colleagues for the things that I learned from them. I would also like to thank *Computer Science* department and *Rensselaer Polytechnic Institute* for giving me an opportunity to pursue graduate studies.

Finally and most importantly, I would like to thank almighty God for blessings and kindness.

ABSTRACT

With the increase in bandwidth and interconnection of computers, there has been an increasing adoption of the distributed computing paradigm to solve computationally intensive problems that would normally take a long time if solved on a single computer. However, the development of programming languages and libraries specifically developed for distributed programming tasks such as efficient work distribution, network setup, load balancing, and dynamic reconfiguration has been slow. This results in inefficient implementations and considerable time spent in developing distributed computing specific parts, rather than actual application development.

The issues that were identified earlier in distributed application development span programming languages, library support, and run-time systems. This thesis addresses these issues mainly at the library level, with some programming language level and run-time level support.

This thesis identifies and develops abstractions, to enable designing and developing generic components, and libraries for distributed computations. It uses a *Generic Software Design Methodology* to design these abstractions and to enable maximum reuse, flexibility, and efficiency. A *Generic Framework for Distributed Computing - GFDC* developed in this thesis can be used in a broad range of distributed applications implemented using different technologies and programming languages.

Java has emerged as a platform of choice for developing distributed applications. This thesis provides an implementation of GFDC in *Java*, as a set of generic libraries. For a class of problems, which need reconfiguration and migration, researchers in the *Worldwide Computing Laboratory at Rensselaer Polytechnic Institute* are developing the *SALSA* programming language. The generic libraries that are developed in *Java* are transparently integrated with *SALSA*.

GFDC and its implementation handle the distributed computing tasks and allow programmers to concentrate on the actual application development. This results in a flexible and efficient implementation along with reduced development time for the distributed applications. The programmers who have little or no knowledge of distributed computing issues should be able to develop applications using GFDC. At the same time, GFDC and its implementation is extensible to enable advanced programmers to customize and enhance it, to tackle specific distributed computing issues.

CHAPTER 1

Introduction

There is a growing interest in harnessing the power of the Internet, for computationally intensive applications. Applications in different areas like *SETI@Home* [57] in Astronomy, *Folding@home* [23] and *fightAIDS@home* [22] in Life Sciences, and *Great Internet Mersenne Prime Search* [30] in Mathematics are just a few examples of complex and time consuming computations performed using idle resources on the Internet.

1.1 Issues in developing distributed applications

Irrespective of actual problems in different areas solved by these distributed applications, they share common issues related to distributed computing. Some of the issues tackled in this thesis are:

1. *Efficient Work Distribution*: The distributed applications depending upon the scale need to spread the work efficiently across different resources. This involves categorization of resources based on various parameters like processing power, memory, and bandwidth available for the computation. The efficiency of a resource for a particular computation changes during the life of the computation, and is different across various computations, with different efficiency considerations. Once the efficiency parameters are established, the machines or nodes that participate in the computation need to be categorized based on these parameters. The aim of the application is then to make maximum use of the resources that are efficient by allocating more work to these resources. As the run-time environments change from a fixed set of machines to a dynamic one where new machines can join and existing machines can leave the computations any time, the task becomes quite complex to keep track of the efficient resources.

The application programmer needs to work at a higher level to solve the application specific problems, instead of worrying about these issues. This framework does efficient work distribution, which is more or less transparent to the application programmer, depending upon the nature and complexity of the application being developed.

2. *Network Setup*: A distributed computation needs to decide on the resources that it will be using. This can be done either, during initialization, run-time, or a combination of initialization and run-time. For static environments like a cluster, the resources avail-

able for computation need to be provided during setup. Additionally, depending on the application, these resources need to be categorized for efficiency.

For a dynamic environment, the application needs to keep track of resources that might join or leave the computation at any time. As the resources participating in the computation change, their relative efficiency changes, and the application needs to keep track of these resources.

This framework provides a unified presentation of either *Static* environments, where the resources allocated to the computation remain same throughout the computation, or a *Dynamic* one, where they might change during the computation. This results in minimal changes to the user code, when running the application on different execution environment.

3. *Load Balancing*: As the number of resources and their relative efficiencies, change during the lifetime of a distributed computation, the application needs to distribute the work efficiently between the available resources, to make the maximum use of them. Different load distribution strategies are being researched in the Worldwide Computing Laboratory at Rensselaer Polytechnic Institute and at other places. The framework enables the library developer to plug these load balancing strategies in the framework and the application developer, to make use of these strategies for optimum results.

4. *Dynamic Reconfiguration*: Dynamic reconfiguration is critical in the context of distributed computing, as it allows the application to reconfigure itself to make the optimum use of, the available resources. At the same time, it affects the way in which the application is developed, since it is related to work distribution and load balancing. This framework considers the mixing of dynamic reconfiguration with various work distribution and load balancing strategies to get optimal results for the application.

5. *Library Support* Various libraries like STL (Standard Template Library) [58], BGL (Boost Graph Library) [9], and Loki [45] have been successfully implemented. These libraries make complex code available to average users and give expert users flexibility to easily enhance the library. However, these libraries are not applicable in a distributed environment.

Moreover, implementations of distributed algorithms and data structures, in the form of libraries or reusable components, are not widely available. These results in considerable development time spent in developing and implementing them, instead of directly using them in user applications.

6. *Programming Language Support* There is a need for enhanced programming language level support for distributed computing, which can make the development of distributed programs easier and more efficient. Effective programming language level support for distributed applications would enable programmers to have constructs that would result in simpler and smaller programs.

1.2 A Generic Framework for Distributed Computing

This thesis develops a generic framework for distributed computing to address these issues. The various issues that were identified earlier in distributed application development span programming languages, run-time systems, and library support. Hence, this thesis addresses these issues individually or collectively at three different levels:

- *Programming Language Level:* Here we consider the programming languages and language level features, that would make developing distributed applications easier and more efficient.
- *Library Level:* Here we consider the libraries and components that can be developed to enable application developer to use off-the-self code, to reduce development time. We also consider a framework that would enable advanced programmers to develop reusable libraries and components, for distributed applications.
- *Run-Time Level:* Here we develop abstraction and mechanism to hide the complexities of resource management in distributed computing due to different kinds of hardware architectures, operating and run-time systems, and network topologies. This would enable the application developers to efficiently run their applications on different sets of resources and topologies, with minimum changes to their application code.

This three level strategy is good for tackling specific issues at appropriate levels. At the same time, it allows us to account for cross-cutting issues that cannot be handled at a single level. The major goal of this implementation is to address these issues at the library level with some programming language level and run-time level support. Of the three levels identified earlier, the library level is important, since the abstractions developed at library level are independent of any particular run-time system or programming language. This allows for its implementation in different programming languages on different architectures

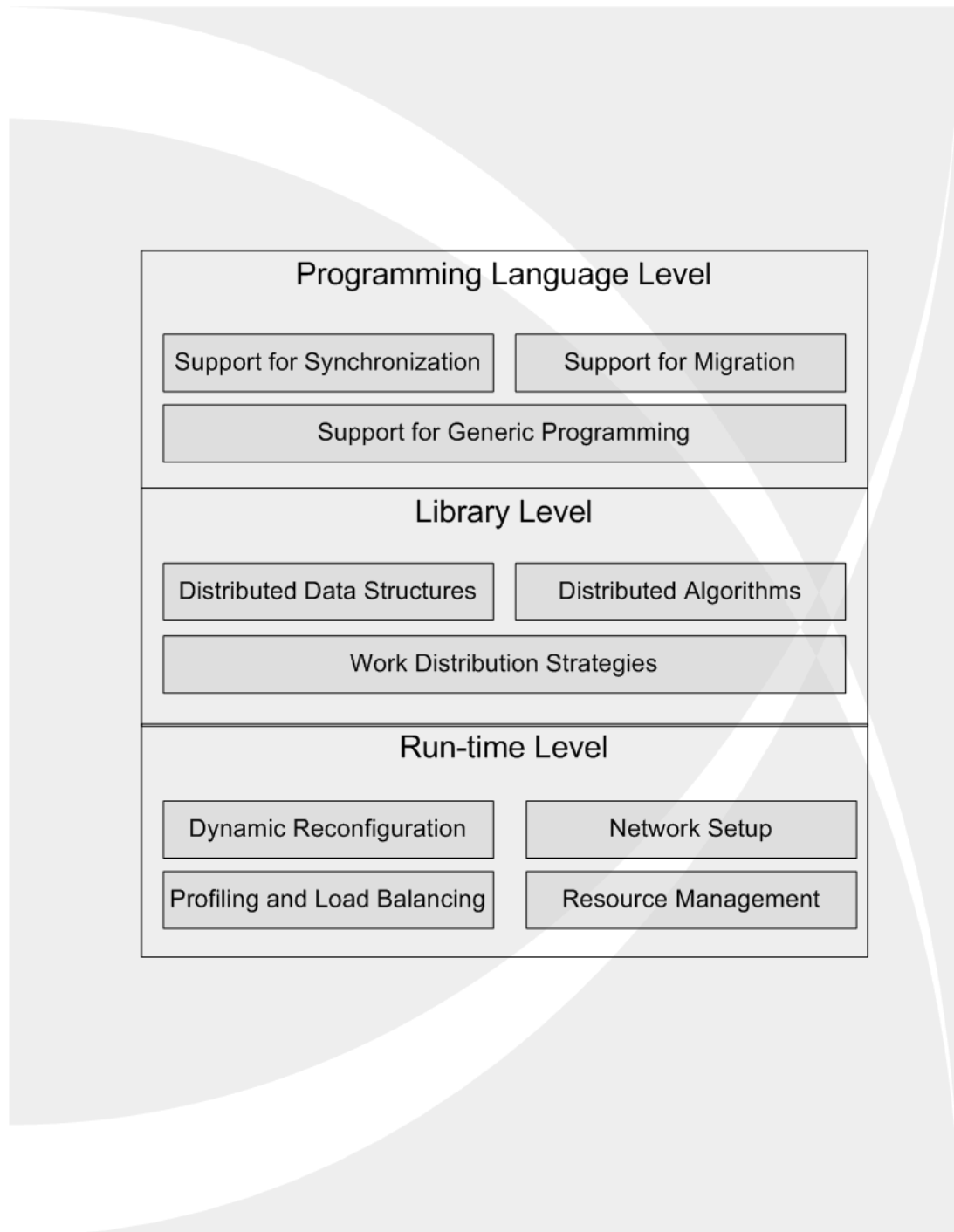


Figure 1.1: Distributed computing issues addressed at three levels. The *Programming Language Level* addresses the issues related to support for synchronization, migration, and generic programming. The *Library Level* addresses the work distribution strategies and development of libraries implementing distributed algorithms and data structures. The *Run-time Level* address the distributed computing specific issues like dynamic reconfiguration, profiling, load balancing and resource management.

and run-time systems. The library level itself consists of various sub-levels to tackle different issues, and the cross-cutting issues can be tackled in different ways specific to programming languages and run-time systems, as long as the abstractions and interfaces that they provide, remain consistent with the framework.

1.2.1 The Generic Programming Methodology Approach

The generic programming methodology refers to the programming language level constructs that support polymorphism. Various programming languages support *Parametric Polymorphism* and/or *Subtype Polymorphism* which enables a polymorphic code to be used across different types in an efficient manner.

In parametric polymorphism a method, class, or interface is parameterized by one or more types. This enables the user to select the actual parameter type when the method, class, or interface is instantiated. Parametric polymorphism is useful for defining generic behavior where the actual parameters need not be related in the type hierarchy. It allows the use of a single abstraction across many types.

Subtype polymorphism on the other hand, is useful for defining generic behavior over a set of related types. Therefore, an entity that acts on a particular type can also work on the types derived from that type.

A major part of generic programming is to identify and define the abstractions needed, to implement generic components and libraries. Efficiency, extensibility, and user friendliness are the major considerations while designing these generic components.

Various libraries like STL, BGL, and Loki have been successfully developed and implemented, using this methodology. These libraries are not applicable to distributed applications, since it was not their intended use. However, the study of their design methodology and identified abstractions can be helpful while developing similar generic libraries for distributed computing.

By having proper abstractions at correct levels, we can address the range of issues that we identified earlier for distributed computing. This would help combine the various abstractions in a generic way, so as to provide maximum number of possible solutions from a limited amount of code. Hence, the issues in distributed computing at the library level are addressed using this methodology.

The following three sections describe the motivation for addressing the issues in distributed computing at the three levels using the generic programming approach.

1.2.2 Programming Language Level

Java [37] has become a programming language of choice for developing distributed applications. Library support and features for distributed technologies like RMI [55], CORBA [16, 42, 53] and serialization have been major reasons for success of Java in distributed computing.

Java does have limitations for certain classes of applications that need dynamic re-configuration and migration due to Java's passive object model. The researchers at the Worldwide Computing Laboratory are developing SALSA [62], an actor-based programming language for mobile and Internet computing as well as run-time support Internet Operating System (IOS) [35] and Worldwide Computer (WWC) [61]. SALSA simplifies programming dynamically-reconfigurable applications using the additional programming language support for migration, reconfiguration, and asynchronous communication using the Actor model [28].

Applications written in the SALSA programming language are compiled to Java and hence preserves many of Java's useful object oriented concepts. It also allows for creating Java objects inside the Actors, which allows the SALSA programmers to use the Java libraries and other existing Java classes. However, as SALSA is ultimately transformed into Java, it has the inherent limitations of Java. These are the motivating factors for adding generics to Java [10] and SALSA:

- Need for casting, which can get complex and complicated.
- Incorrect use of casting can result in run-time exceptions. This is especially important in distributed applications where it is desirable to detect as many errors as possible, at compile time, instead of at run-time.
- Lack of compile-time support for generics leads to less reliable and more complex libraries for developing distributed applications in Java and SALSA.

Thus, adding generics to SALSA and Java increases expressiveness and safety, that is useful while developing generic libraries.

Additionally generics can be used for SALSA to enhance and enable specific features like:

- The ability to specify the type of tokens at compile time.

- Increase the expressiveness and safety of the actor behaviors and interfaces. This can be done by making type parameters explicit in their declaration and definition. That will make the type casts implicit for their use.
- Write libraries of families of similar actor behaviors based on combinations of template arguments.
- Enhance the SALSA interface with IOS and WWC run-time to enable the programmers to describe the run-time actor topology for better profiling and reconfiguration decisions by WWC/IOS.

1.2.3 Library Level

As discussed earlier, while developing distributed applications, the developer has to address issues like efficient work distribution, network setup, load balancing, dynamic reconfiguration, and replication. In addition to these issues, the common algorithms and data structures have to be implemented by the developer. This creates the following problems:

- An application developer who is generally an expert in the application area of the problem, may not understand the issues related to distributed computing. This discourages lot of application developers from successfully developing distributed solutions for their problems.
- Even if the developer understands the various issues, and is able to implement the solution, it may not be the best implementation, and can be improved.
- For successful implementations, there is a replication of similar code, to address the issues related to distributed computing. This results in longer development cycles as there is very little reuse of code. Additionally, with the growing research and use of distributed algorithms and data structures, there is a need to implement these in generic libraries so as to be available for use by average users.

The libraries developed for single machine or cluster computations do not address these issues. Java has some library-level support to enable developing distributed applications. However, it lacks the application oriented libraries to do standard tasks in distributed application development like work distribution and optimum resource utilization.

Salsa also lacks the library level support to enable faster and easier code development. Hence, there is a need to have libraries, that would take advantage of the actor model and SALSA programming language features. Just as the popularity of Java was not only due to its programming language features, but also because of its library support for faster and reliable application development, we believe that further success of Java and SALSA depends upon our ability to provide application oriented libraries and programming language features for development of these libraries.

Hence, there is a need for developing efficient libraries that enable faster and easier distributed application development. The libraries would reduce the inefficient implementations and considerable development time spent in common distributed computing specific tasks across multiple applications. Developing libraries using generic programming methodology would enable us to develop libraries that would enable maximum amount of reuse.

1.2.4 Run-Time Level

The generic libraries ultimately need support and feedback from the run-time to make decisions for aspects like efficient work distribution, network setup, load balancing, dynamic reconfiguration, and replication. These libraries would depend on the profiling and run-time characteristic provided by the run-time to dynamically optimize the application. The run-time itself can be written in a modular way by using generic Java and/or generic SALSA.

1.3 Evaluation Metrics

There are various programming languages and tools developed for general purpose distributed programming. For specific application areas in distributed computing, various programming languages like SALSA are being developed for dynamically reconfigurable systems by enabling efficient migration.

The generic framework for distributed computing encompasses programming level abstractions as well as network and machine level abstractions at lower level for the efficient implementations of the higher level abstractions.

Therefore, the evaluation metrics for this implementation can be categorized as:

- *Programmer level* issues of implementing with or without the use templates, based on following factors:

- Simplicity.
- Ease of application development.
- Support for genericity.
- *Run-Time* level comparisons that consider issues like:
 - Performance in terms of network and individual nodes.
 - Scalability
 - Ease of reconfiguration.

The programmer-level success metrics are discussed in the design section, and the run-time level comparisons are made in the implementation section.

Structure of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents the abstractions and the design of the generic framework for distributed computation. Chapter 3 presents the implementation of the framework described in Chapter 2 in form of generic components and libraries. Chapter 4 discusses the possible applications of the GFDC. It also presents two sample applications developed using GFDC. The success metrics that were discussed in Chapter 1 are applied to the implementations in Chapter 3 and the relative advantages and disadvantages are evaluated in Chapter 5. Chapter 6 presents related work in the area of distributed computing and generic programming. Chapter 7 discusses future work. Finally, Chapter 8 presents conclusions and contributions.

CHAPTER 2

Generic Framework for Distributed Computing (GFDC)

2.1 Overview of GFDC

Single machine abstractions required for *Generic Programming* [56] are very well defined and implemented by STL, BGL and Loki. However, there is a lack of proper identification and implementation of the abstractions required for programming distributed applications.

Generic programming is effective in providing language abstractions and interchangeable components to build generic libraries, which significantly reduces development time without sacrificing efficiency. Generic distributed programming uses the underlying philosophy of generic programming to develop concepts and generic libraries that will result in faster and more efficient distributed application development.

Depending upon the scalability and performance requirements, the distributed applications vary from high performance clusters with low scalability, to Internet scale applications with low performance requirements, but high scalability. Therefore, even though the requirements for developing distributed applications are similar, depending upon the scalability and performance requirements, their relative importance for specific application is different. However, all the applications still require most of efficient work distribution, load balancing, dynamic reconfiguration, and replication strategies. The development time can be further reduced, along with better implementations, if we are able to provide generic distributed algorithms and data structures.

Hence, we follow a multi-level approach in designing the libraries and providing appropriate abstractions at all the levels to handle the specifics. At higher level, we provide concepts of distributed algorithms and data structures. At a lower level, we provide the abstractions for the distributed computing specific issues like work distribution, load balancing, dynamic reconfiguration, and replication. In a way, it is similar to STL, which provides facilities for *Algorithms* and *Containers*, which store collections of other objects. These higher level abstractions need *Allocators* at lower levels, to take care of memory allocation issues specific to different architectures.

This design allows us to combine algorithms and containers at a higher level like STL. At the same time, it also handles the specifics of the way in which these are implemented

in the underlying distributed system at the lower level, which can range from a high performance cluster to Internet. The conflicting nature and the wide variety of possible distributed architectures that can be modeled, create the requirement of highly specific configuration of these generic components depending on the application and execution environment. Another conflicting requirement is to make these components generic and simple enough that they can be used directly by an average programmer, but can be highly customized by an advanced programmer.

We solve these problems by using policy-based design. Policies establish an interface for a particular issue, and can be implemented in a variety of ways, as long as the policy interface is maintained. This allows us to assemble a concept with complex requirements out of multiple smaller concepts each of which takes care of only one behavioral or structural aspect. It involves creating a hierarchy of concepts gradually from the lower-most level to the highest level that the programmer uses directly. However, by modifying or providing different policy classes with consistent interfaces to pre-existing ones, a programmer is able to configure the libraries for his specific use at any and all levels of the policy-based hierarchy.

Thus, a programmer can use the same distributed algorithm, or work distribution design on the cluster and Internet by choosing appropriate network concepts for that distributed implementation. At the same time, if required, he can modify the network concept by creating or modifying a set of policy classes, and using them with the same distributed algorithm or work distribution design to get better performance or scalability out of his specific distributed environment. This allows us to develop a highly flexible architecture by combining the concepts and policies provided by the library, with the new policy classes created by the programmer that have different implementations of the same policies.

For example, if we want to use the *Farmer-Worker* work distribution design strategy to solve the *Busy Beaver*[12] on Internet, *Missing Baryons* on high performance cluster and *Twin Prime*[8] on Intranet. In all these cases, the design strategy at higher level remains the same and solves all the three problems, but the underlying distributed environment on which the application would be run changes. This can be handled by choosing appropriate lower level network category, assembling a specific network by combining existing, or new categories of load balancing and dynamic reconfiguration categories.

The GFDC architecture presents this multi-level abstraction strategy to implement

various distributed algorithms and data structures at higher level. At the same time, distributed computing-specific components like efficient work distribution, network, load balancing, dynamic reconfiguration, and replication can be implemented at middle and lower levels.

The libraries using this architecture can be implemented in languages that have support for Generic Programming. The critical thing is that even though dependent on the features and limitations of the language, the actual implementations would be different, they should follow the similar set of hierarchy and concepts that are developed to maintain consistency. This is similar in spirit to the idea of generic containers and algorithms in C++ and Java representing the same concept even though their implementation is different. Sometimes, this may or may not be possible in all the implementations, or the implementer can add own new concepts due to the unique advantages of that particular implementation.

For example an actor-oriented programming language like SALSA, that is specially designed to develop dynamically re-configurable open distributed applications, would be able to implement additional concepts, not possible in traditional languages like C++. However, its genericity is limited to a great extent by that of Java.

The work is not limited to a theoretical exercise of developing the architecture and concepts, but also to implement it in different programming languages like Java and SALSA, and to refine and improve the architecture from the experience gained during the implementation. An interesting side effect of this would be a comparison of relative advantages and disadvantages of implementations in these languages, and the ways to improve the languages and their libraries so as to enable easier and efficient distributed computing using these languages.

2.2 GFDC Architecture

The Generic Framework for Distributed Computing is a layered set of abstractions as shown in Figure 2.1.

2.2.1 Resource Level

This is the lower level, and represents the various resources that are used in distributed computing. So, in case of Java we have *Node*, and in case of SALSA we have *Theater*. These abstractions are similar as they are the resources where the actual computation takes place.

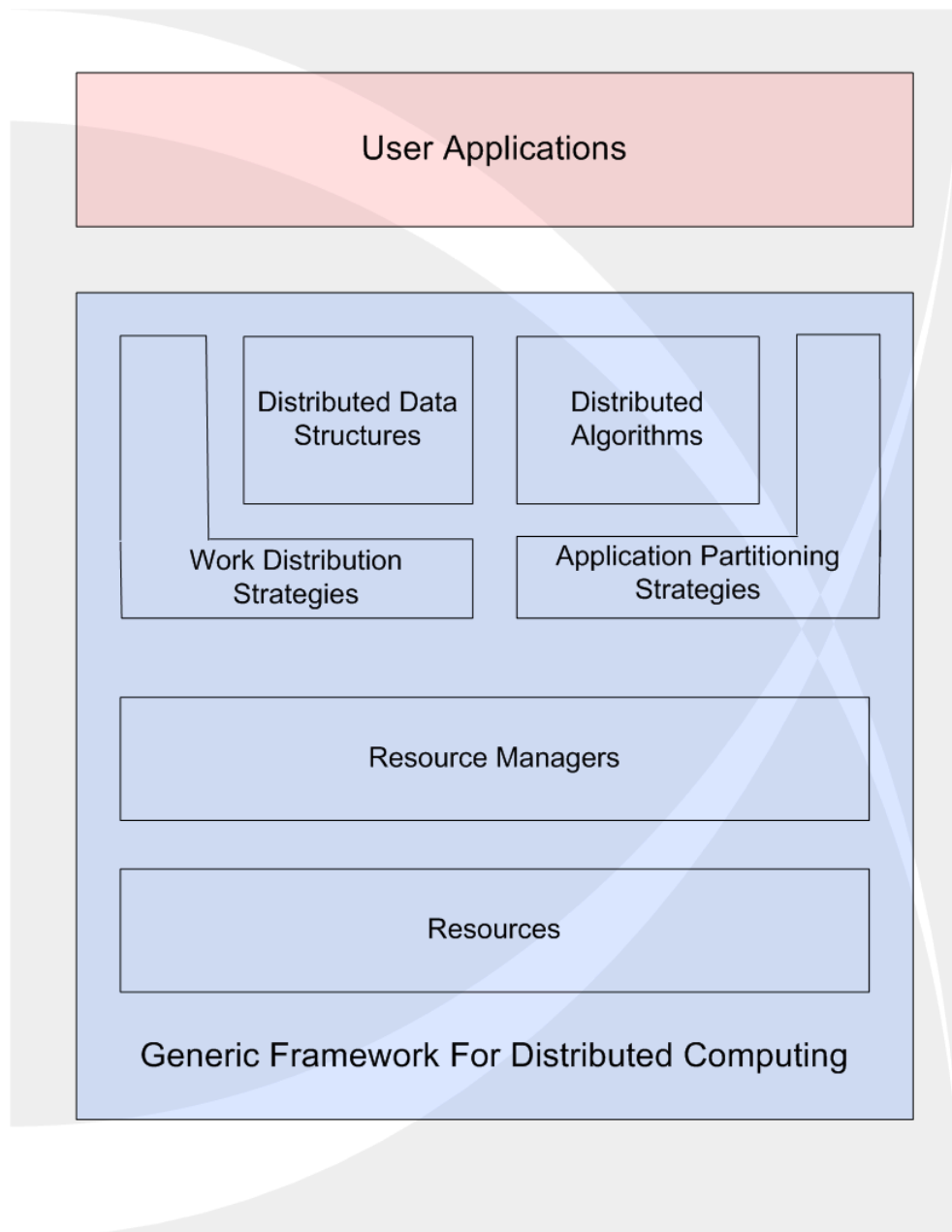


Figure 2.1: Generic Framework for Distributed Computation. In this multi-level framework, various abstractions like resources, resource management and allocation, work distribution, distributed data structures, and algorithms are identified. It also presents the interaction between these levels within the architecture, and also, the interaction of the user application with the framework.

2.2.2 Resources Managers Level

The *Resources Managers* manage the resources which can be any type of lower level resource like Node or Theater which take part in the computation. The *Static Resource Manager* and the *Dynamic Resource Manager* are two sample resource managers. The static resource manager is used in the computations where the resources are allocated for the computation before the computation begins. The dynamic resource manager is responsible for keeping track of the resources in a dynamic environment, where a resource may join or leave the computation at any point of time.

2.2.3 Work Distribution Level

This level requests the resources from the appropriate resource managers at the lower level and uses those resources to distribute the work. *Farmer-Worker* and *Hierarchical* are the two sample implementations of the *Work Distribution*.

In case of the farmer worker work distribution strategy, a single stable resource is considered as the main node of the computation, and is assigned as the farmer. Farmer then distributes the work to multiple workers.

In case of hierarchical work distribution strategy, a single stable resource is considered as the main node of the computation and is assigned as the farmer. The farmer then distributes the work to multiple workers similar to the farmer worker strategy. The main difference is that the workers can be either *Sub Farmers* or *Workers*. Workers are resources that would do the assigned work and report the work back to the farmer. Sub farmers can in addition to doing the work on behalf of the farmer, can themselves act as farmers by creating more workers and delegating some of the work to them. As the workers created by the sub farmer can again be either sub farmer or worker a hierarchy of sub farmers and workers is created depending on the need of the application.

2.2.4 Application Partitioning Level

Application Partitioning Strategies is used at the application level to distribute the state space of the problem. *Alternate* distributes half of the states in the container to the new resource for further computations. *Sub Part* distributes some percentage of the states in the container to the new resource for further computations. Depending on the size of sub part, those many states in the container, are transferred to the new resource and hence the name, sub part.

2.2.5 Distributed Algorithms and Data Structures Level

At this level, various distributed algorithms and data structures can be implemented. This level utilizes the services provided by the lower levels of *Work Distribution* and *Application Partitioning Strategies*.

This is similar in design to STL, where the containers and algorithms use allocators to take care of machine dependent tasks like memory allocation. Similarly, in the context of distributed computations, the network and work distribution tasks are performed by lower levels while at higher level the developer can implement the various distributed algorithms and data structures.

2.2.6 User Application Level

This is the user-level layer that interacts with the generic framework for the distributed computing architecture. Here the application developer utilizes the lower level libraries to rapidly develop a distributed application. Generally, the user need not change the components at various levels, and can use the components available from the library itself. An advanced user can add specific components for the application being developed as required, or modify existing components by mixing various components.

CHAPTER 3

Implementation

3.1 Overview

In earlier chapters, we discussed issues related to distributed computing, and how they can be tackled using a multi-level generic framework. In this chapter, we introduce an actual implementation of the various levels of this framework. At each level, classes that present a minimum interface for that level along with their specialized versions are presented.

The implementation is in generic Java (1.5.0-beta2-b51). This is the latest release of Java that supports generics. All the classes that implement the generic framework for distributed computation are in `gfdc` package. The actual source code for the implementation can be downloaded from www.cs.rpi.edu/wwc/gfdc

In the following section, we will be presenting the classes for each level, and discuss their dependencies and uses by other classes in the framework.

3.2 Resources

This is the lowest level of the framework. At this level, we model the various types of resources that are used in distributed computing. In case of *Java* or *C++* we have `Node`, which represents the machine where the computation take place. In case of *SALSA* we have `Theater` which represents the theater in the context of a SALSA/WWC system.

These resources represent different physical and logical entities. However, their abstractions are similar as they are the resources where the actual computation takes place. The `ResourceManagement` class at the upper level uses these classes.

As shown in Figure 3.1, `Resource` is the abstract base class, and is the root class for various types of resources. Sample resources like `Theater` and `Node` are derived from `Resource`.

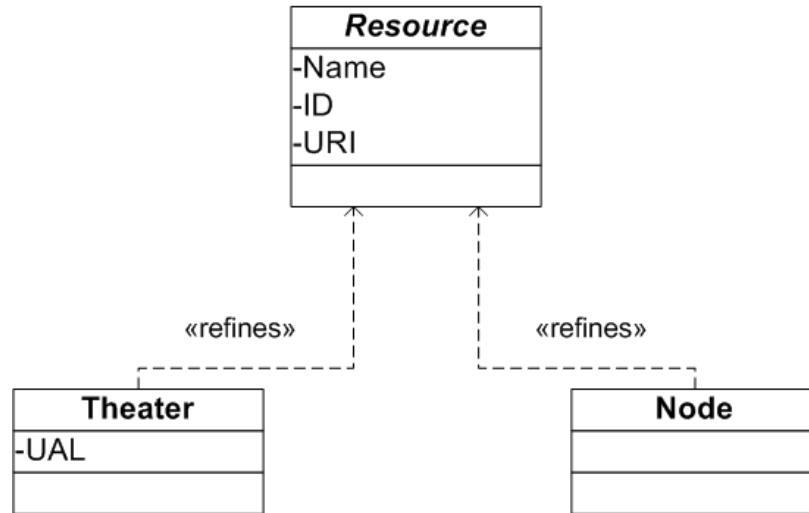


Figure 3.1: UML class hierarchy of different types of Resources. Here Resource is the abstract base class for various types of resources. Node and Theater are the two sample derived classes.

3.2.1 Resource

This class acts as the base class for various types of resources. It has variables like **Name** for name, **ID** for identification, and **URI** for Uniform Resource Identifier [7]. These variables are used to identify and store information of the resources. It also has methods to perform various activities on the resources by the classes in the higher level of the framework. Some of the required methods are implemented in this class, so the derived classes need not implement them unless required.

```

public abstract class Resource implements Serializable {

    protected String Name = "NULL";
    protected int ID = -1;
    protected String URI = "NULL";

    // Implements the methods that act on the member variables.
    // These methods are overridden in the derived classes if required.

}
  
```

Figure 3.2: The Resource class is the abstract base class for classes that are part of the resource level. It has the variables to completely describe the resource and the methods required to act on them.

3.2.2 Node

The `Node` class is used to model an available resource like machines with their *Internet Protocol* IP addresses and port number. These kind of resources are generally used by *C++* and *Java* for distributed network computations. Along with the methods derived from the base class, it has methods specific to its behavior.

```
public class Node extends Resource {
    public Node () { ... }
    public Node (String rURI) { ... }
    public Node (String rName, String rURI) { ... }
    public Node (String rName, String rURI, int rID) { ... }
}
```

Figure 3.3: The `Node` class is the specialization of the `Resource` class. It has the overloaded constructors to accept resources that can be partially or fully described.

3.2.3 Theater

The `Theater` class is used to model an available resource like theater in the context of *SALSA/WWC*. It has additional member variables like *Uniform Actor Locator* UAL, which is specific to the theater. It also has additional methods specific to its behavior, along with the methods derived from the base class.

```
public class Theater extends Resource {
    private String UAL = "NULL";
    // Implements overloaded constructors.
    // Implements methods to act on the new variable UAL.
}
```

Figure 3.4: The `Theater` class is the specialization of the `Resource` class. It has an additional variable UAL to store the UAL of the agent in the *SALSA/WWC* programming paradigm. It also has methods to act on the UAL variable and overloaded constructors to accept resources that can be partially or fully described.

3.3 Resource Managers

The resources managers are responsible for the resources taking part in the computation. The resources, which can be any type of lower level resource like `Node` or `Theater`, are categorized and managed differently by the various Resource Managers.

`ResourceManagement` is the base class for the resource manager level. It is abstract, and implements some common functionality that is needed by all the resource managers. `StaticResourceManager` and `DynamicResourceManager` are the two derived classes of `ResourceManagement`.

The `StaticResourceManager` know the resources allocated for the computation before the computation begins. The `DynamicResourceManager` is responsible for keeping track of the resources in a dynamic environment, where a resource may join or leave the computation at any point of time.

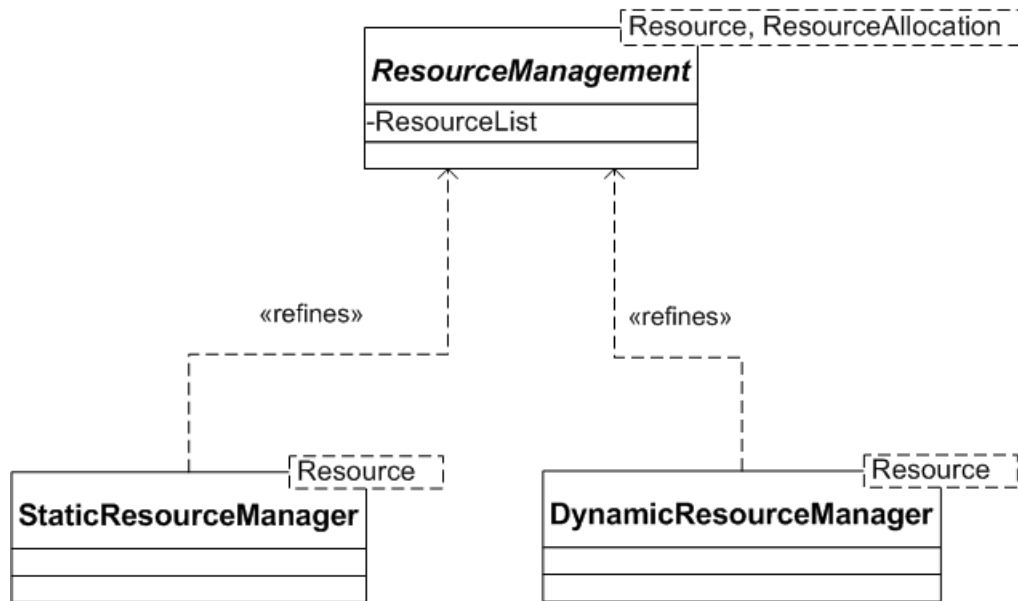


Figure 3.5: UML class hierarchy of different types of resource managers. Here *Resource Management* is the abstract base class for various types of resources managers. *Static Resource Manager* and *Dynamic Resource Manager* are the two sample derived classes.

3.3.1 Resource Management

The `ResourceManagement` is the abstract base class which is the root for various implementations of resource managers. Sample resource managers like `StaticResourceManager` and `DynamicResourceManager` are derived from `ResourceManagement`.

`ResourceManagement` stores a list of resources in `ResourceList` those are available to the computation. It also has the methods that are related to resource managements, which are required by all resource managers. The class is parameterized for different types of resources that it can handle.

```
public abstract class ResourceManagement<Res> implements Serializable {
    private LinkedList <Res> ResourceList = new LinkedList <Res> ();

    // Implements the methods that act on the member variables.
    // These methods are overridden in the derived classes if required.
}
```

Figure 3.6: The `ResourceManagement` class is the abstract base class for the classes that implement the *Resource Management* level. It is parameterized for the type resources that it manages. It has the variables and methods to keep track of the resource that it is managing.

3.3.2 Static Resource Manager

It is used for computations where the resources (machines/theaters) are known initially. In the case of `StaticResourceManager` the resources are expected to be the same during the computation.

```

public class StaticResourceManager<Res>
    extends ResourceManagement<Res> {

    // Resources passed to it are added to the list of resources.
    public StaticResourceManager(Res[] r){

    // Copy the URI of all the Resources into the ResourceList.
    for(int i = 0; i < r.length; i++)
        addResource(r[i]);
    }
}

```

Figure 3.7: The `StaticResourceManager` class extends the abstract `ResourceManagement` class and is parameterized for the type resources that it manages. It has the constructor that accepts and array of resources that it will manage during the computation.

3.3.3 Dynamic Resource Manager

It is used for computations where the resources (machines/theaters) are obtained during run-time. In case of `DynamicResourceManager` the resources can change during the computation. It has `fetchResources()` method to get resource list from server.

```

public class DynamicResourceManager<Resource>
    extends ResourceManagement<Resource> {

    public void fetchResources() {

        // Get available resources from server at run-time.
    }
}

```

Figure 3.8: The `DynamicResourceManager` class extends the abstract `ResourceManagement` class and is parameterized for the type resources that it manages. It resources that it manages can change during the computation. It has method to get resources from server at run-time.

3.4 Resource Allocation

These implementations keep track of the resources that are available for the application at a given point of time. They also keep track of the next resource to be allocated. The higher level classes request resources from any one of these implementations.

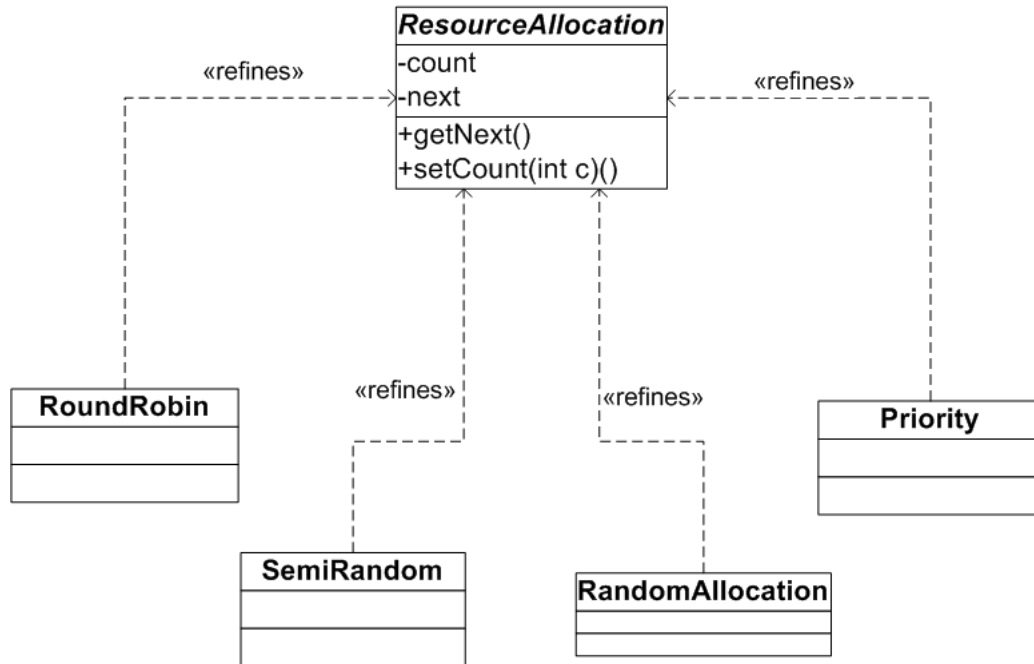


Figure 3.9: UML class hierarchy of different types of *Resource Allocations*. Here *Resource Allocation* is the abstract base class for various types of resources allocators. RoundRobin, RandomAllocation, Priority and SemiRandom are the sample derived classes.

`ResourceAllocation` is the abstract base class from which other classes are derived. RoundRobin, Priority, SemiRandom and RandomAllocation are four sample implementations of the resource allocation which are derived from `ResourceAllocation`. These classes override `getCount()` and `setCount()` methods as required for specific implementation.

As required, a user can add more resource allocation implementations for specific applications.

3.4.1 ResourceAllocation

This is the abstract base class from which other classes are derived. It defines two variables `count` and `next` to keep track of the total resources and the next resource to be allocated respectively. It has `getCount()` and `setCount()` methods to act on these variables.

```
public abstract class ResourceAllocation implements Serializable {  
  
    protected int count = 0; // Initially set to 0 to denote empty.  
  
    protected int next = -1; // Initially set to -1. It WILL be  
                            // appropriately set by the getNext()  
                            // method implemented in the derived class.  
  
    // The derived classes HAVE to implement this method  
    public abstract int getNext();  
  
    public int getCount() {  
        return count;  
    }  
    public void setCount(int c) {  
        count = c;  
        next = 0;  
    }  
}
```

Figure 3.10: The `ResourceAllocation` class is the abstract base class for the classes that implement the *Resource Allocation* level. The class has variable to keep track of the total and next resource to be allocated and the methods to act on those variables. It has an abstract method `getNext()` which must be implemented by all the concrete derived classes.

3.4.2 Round Robin

In case of round-robin allocation of resources, the resources are allocated fairly every time a request for resource allocation is made.

```

public class RoundRobin extends ResourceAllocation {

    public RoundRobin() { }

    public RoundRobin(int c) {

        count = c;

    }

    public int getNext(){

        if(next == count) // Check if next is equal to the limit.
            next = 0; // In that case reset it to zero.
        else // Or increment the next.
            next++; // As next is initially set to -1 in
                // base class. This works in all cases.

        return next;

    }

}

```

Figure 3.11: The RoundRobin class implements the getNext() method so that the resource allocation is based on round-robin manner.

3.4.3 Priority

In case of priority allocation, the resources are allocated based on their priorities. Priority can be assigned based on single or combination of resource attributes like stability, processing speed or memory available.

```

public class Priority extends ResourceAllocation {

    // Methods to allocate resources in a semi-random manner.

}

```

Figure 3.12: The Priority class implements the getNext() method so that the resource allocation is based on the priorities of the resources.

3.4.4 RandomAllocation

In case of `RandomAllocation` resource allocation any resource is allocated randomly. In the `getNext()` function a random number is generated between the range of 0 to `count`.

```

public class RandomAllocation extends ResourceAllocation {

    private Random rnext = new Random();

    public RandomAllocation() {

    }

    public RandomAllocation(int c) {

        count = c;

    }

    public int getNext(){

        // Return a random number within a range specified by count.
        return rnext.nextInt(count);

    }

}

```

Figure 3.13: The `RandomAllocation` class implements the `getNext()` method so that the resource allocation is done in a random manner.

3.4.5 SemiRandom

Some of the resources that have maximum priority are preferentially allocated and other resources are allocated randomly or in a round-robin manner.

```

public class SemiRandom extends ResourceAllocation {

    // Methods to allocate resources in a semi-random manner.

}

```

Figure 3.14: The `SemiRandom` class implements the `getNext()` method so that the resource allocation is done in a semi-random manner. In such a case, one or more special resources are allocated by special requests and the remaining resources are allocated randomly.

3.5 Work Distribution

Work distribution combines any implementation of `ResourceManagement` with any implementation of `ResourceAllocation` to effectively allocate work among the resources available to the computation.

3.5.1 Farmer Worker

In case of *FarmerWorker* work distribution strategy, a single stable resource is considered as the main node of the computation and is assigned as the farmer. The farmer then distributes the work to multiple workers according to the work allocation strategy been used for the resources that are made available by the resource manager.

```

public class FarmerWorker implements Serializable {

    // StaticResourceManager used to manage Resources.
    C1: private StaticResourceManager<Theater> srm;

    // ResourceAllocation used to decide resource allocation
    C2: private ResourceAllocation ra;

    public void initializeFarmer(String[] args ) {

        // These theaters are available for computation.
    C3: Theater t[] = new Theater[args.length-2];

        // Initialize all the Theaters

        // Store these theaters to create workers required
    C4: srm = new StaticResourceManager<Theater>(t);

        // Initialize the resource allocator
    C5: ra = new RoundRobin(args.length - 3);
    }

    public void createWorkers(int count) {
        for(int i = 0; i< count; i++)
        {
    C6: Theater th = srm.getResource(ra.getNext());

            // store the UAN of the worker in the list
    C7: sub_farmers.addLast(th.getURI() + "worker" + (i+1)) ;

            ...
        }
    }

    // Other methods of the FarmerWorker class
}

```

Figure 3.15: The `FarmerWorker` class implements the *FarmerWorker* work distribution strategy. **C1:** `StaticResourceManager` is declared for resource type `Theater`. **C2:** `ResourceAllocation` variable `ra` is declared. **C3:** An array of `Theater` is created and each theater is initialized. **C4:** `StaticResourceManager` is defined with array of `Theater` from **C3**. **C5:** `ResourceAllocation` `ra` is assigned an object of type `RoundRobin`. **C6:** According to the `ResourceAllocation` `ra` a new resource of the type `Theater` is requested from `StaticResourceManager` `srm`. **C7:** *Worker* is added to the list of workers.

3.5.2 Hierarchical

In case of *Hierarchical* work distribution strategy a single stable resource is considered as the main node of the computation and is assigned as the farmer. The farmer then distributes the work to multiple workers according to the `WorkAllocation` strategy been used and the resources made available by the `ResourceManager`. These workers can be either sub farmers or workers. Workers are resources that would do the assigned work and report the work back to the farmer. Sub farmers can in addition to doing the work on behalf of the farmer, themselves act as farmers by creating more workers and delegating some work to them. As the workers created by the Sub Farmer can again be either sub farmer or worker, a hierarchy of sub farmers and workers is created depending on the need of the application.

```
public class Hierarchical implements Serializable {  
  
    // StaticResourceManager used to manage Resources.  
    C1: private StaticResourceManager<Theater> srm;  
  
    // ResourceAllocation used to decide resource allocation  
    C2: private ResourceAllocation ra;  
  
    // Other methods of the Hierarchical class  
  
}
```

Figure 3.16: The `Hierarchical` class implements the *Hierarchical* work distribution strategy. **C1:** `StaticResourceManager` is declared for resource type `Theater`. **C2:** `ResourceAllocation` variable `ra` is declared. It also implements the required methods for managing the resources.

3.6 Application Partitioning

Application partitioning is used at the application level to distribute the state space of the problem.

3.6.1 Alternate

It uses the work distribution policy to distribute half of the states to the new resource for further computations. The alternate states are transferred to the new resource and hence, the name *Alternate*.

```
public class Alternate <List, WorkDistribution> { ... }
```

Figure 3.17: The `Alternate` class implements the *Alternate* work distribution policy.

3.6.2 Sub Part

It uses the work distribution policy to distribute some portion of the work to the new `Resource` for further computations. Depending on the size of sub part, that much work is transferred to the new resource and hence, the name *Sub Part*.

```
public class SubPart <List, WorkDistribution> { ... }
```

Figure 3.18: The `SubPart` class implements the *Sub Part* work distribution policy.

CHAPTER 4

Applications

4.1 Overview

The Generic Framework for Distributed Computing provides a hierarchical framework of abstractions. These abstractions and their implementations can be combined in different ways to get a large number of execution frameworks for distributed computations.

The flexibility of the architecture allows the creation of additional abstractions augmenting existing abstractions, thus enabling the development a wide range of distributed computing applications.

At the same time, by choosing the correct set of abstractions and interchanging them, the programmer can scale and configure the application as it evolves.

GFDC essentially allows the programmer to:

- Execute the same application in a host of different environments and scale, by choosing different set of abstractions. For example, a data intensive application that relies on distributing the data over a large number of nodes can be implemented at a cluster level or at an Internet level by choosing appropriate Resource Managers and Work Distribution Strategies.
- Execute similar application on the same set of abstractions that have been proved effective by earlier applications. For example, various applications like Folding@home and fightAIDS@home are similar applications that are developed using the computational model of SETI@Home. Therefore, once the combination of a set of abstractions for a particular type of problem is established, other similar application can be easily and efficiently developed.

Some of the sample applications that can be implemented using GFDC are:

- Applications in physics and biology that need high performing clusters with low latencies, for studying the data from live experiments would be easily developed using GFDC. They can use the `StaticResourceManager` and `FarmerWorker` or `Hierarchical` work distribution strategies that are part of GFDC.

- Applications that distribute streaming audio or video signals can be easily developed using GFDC. They can use the `StaticResourceManager` or `DynamicResourceManager` and `FarmerWorker` work distribution strategies that are part of GFDC.
- Distributed storage system can be implemented using GFDC by using existing lower level abstractions and newer higher level abstractions.
- Applications that need to compute the solution by working on huge data sets. Such application are present in number of areas discussed earlier like *SETI@Home* in Astronomy, *Folding@home* and *fightAIDS@home* in Life Sciences, and *Great Internet Mersenne Prime Search* in Mathematics. The developers of these applications can use GFDC to simplify the application development by relying on GFDC to implement distributed computing tasks. This would allow them to concentrate on the actual application development. They can use the `DynamicResourceManager` and `FarmerWorker` or `Hierarchical` work distribution strategies that are part of GFDC.

In the following section, we describe the actual implementation of two different applications on same set of abstractions.

4.2 Sample Application Implementations

In this section, we present two applications that are developed using the framework. The *Most Productive Turing Machine* and a *Distributed Search* application. Both these applications distribute the work than needs to be done over multiple resources. A `FarmerWorker` work distribution strategy with a `StaticResourceManager` resource manager is used. As the applications are developed in SALSA, the resource of type `Theater` is used in both the applications. `RoundRobin` resource allocation is used in both the cases.

The following two subsections explain the application and the results in detail.

4.2.1 Most Productive Turing Machine

The *Most Productive Turing Machine* problem also called as *Busy Beaver*, is defined as follows:

Consider an n -state binary alphabet turing machine with an infinite tape that is initially blank. The n -state machine which will leave maximum number of 1's on the tape before coming to a halt is called the busy beaver and the number of 1's left on tape is called the productivity of that machine. Then depending on the combinations of explicit or implicit halt, position of head in halt position, quadruple or quintuple formalism, relative position of 1's on the tape at the halt state various variants of the busy beaver are defined.

For smaller values of n (number of states of Turing Machine) like 1 or 2 the Busy Beaver can be found. But for higher values of n , the search tree of the machines to be considered even after all the optimizations in the search process becomes huge and each machine take enormous number of steps before coming to halt.

The `MostProductiveTuringMachine` program developed in SALSA language is used to distribute the work among multiple computers to find the busy beaver champion. We are using *Farmer-Worker* work distribution strategy to a distributed computation on different theaters.

The `MostProductiveTuringMachine` has other files that are required for the computation are not shown below. What is shown below is the `MostProductiveTuringMachine` SALSA file which uses the generic framework for distributed computing for faster and efficient application development.

Initially the `MostProductiveTuringMachine` calls the `finit()` method of the `FarmerWorker` for initialization during which it passes the list of resources available for computation to the `FarmerWorker`.

Then it calls the `createWorkers()` method with appropriate argument to indicate

the number of workers that would be needed by the application. It then creates those workers and the farmer and calls the `initializeFarmer()` method of itself to start the computation.

The `initialize()` method then distributed the work using `FarmerWorker` work distribution strategy and calls its `compute()` method in various workers. The workers perform computation and report the results back to the farmer.

In the `collector()` method of the farmer a track is kept using the `FarmerWorker` of the workers that have finished the computation. Once all the workers report the results back, the farmer publishes the final results and ends the computation.

In the Figure 4.2, you can the result of running the most productive turing machine application. The three windows on the right hand side are the worker theaters that display messages about the status of computation and result. The window on the top left hand side shows the computation from the perspective of the farmer. The farmer displays that the results. The naming server is seen in the bottom right window. The application scripts are run by the user in the bottom left window.

```

behavior MostProductiveTuringMachine {

C1: private FarmerWorker fw = new FarmerWorker();

void Collector(String workerId)
{
C2:     fw.decWorkerCount();
    ...
}
void initialize( ... )
{
    ...
C3:     for(int i=0; i < fw.getWorkerLimit()
        && machines.size() > 0; i++ ) {
C4:         MostProductiveTuringMachine machine =
            new MostProductiveTuringMachine(new UAN(fw.getWorker(i)));
            fw.incrementWorkerCount();
            machine<-compute( ... );
            ...
        }
}
void act(String[] args )
{
C5:     fw.initializeFarmer(args);

C6:     fw.createWorkers(3); // Only three workers needed.

    for(int i = 0; i < 3; i++) {
        MostProductiveTuringMachine machine =
            new MostProductiveTuringMachine();
C7:     machine<-bind( fw.getWorker(i), fw.getWorkerUAL(i))@
            machine<-print(" Worker " + (i+1) + " created");
    }
    ...
}

// Other methods of MostProductiveTuringMachine class
}

```

Figure 4.1: The `MostProductiveTuringMachine` class implements the *MostProductiveTuringMachine* application. C1: An instance of *Farmer Worker* work distribution strategy is created. C2,C3,C4,C7: Various methods of `FarmerWorker` are called. C5: `FarmerWorker` is initialized. C6: `FarmerWorker` is called to create required number of workers.

```

C:\WINDOWS\system32\java.exe
Theater started on port: 4040
Running simulation of BCS
Transition Limit: 1

Farmer activated for Computation.
Creating and calling 3 Workers.
Worker 1 : uan://jawaan:3030/w1 invoked for Busy Beaver Computation

Execution Time for Worker 1 for 0 machines
Execution Time for 0 machines
Time in msec 0
0 hour(s) 0 minute(s) 0 second(s)
Holdouts=0
Candidates=0
Halters=0
Worker 2 : uan://jawaan:3030/w2 invoked for Busy Beaver Computation

Worker 1 Finished
Execution Time for 1 machines
Time in msec 150
0 hour(s) 0 minute(s) 0 second(s)
Holdouts=1
Candidates=0
Halters=0

Execution Time for Worker 2 for 0 machines
Execution Time for 0 machines
Time in msec 0
0 hour(s) 0 minute(s) 0 second(s)
Holdouts=0
Candidates=0
Halters=0

Worker 2 Finished
Execution Time for 1 machines
Time in msec 20
0 hour(s) 0 minute(s) 0 second(s)
Holdouts=1
Candidates=0
Halters=0
Worker 3 : uan://jawaan:3030/w3 invoked for Busy Beaver Computation

C:\WINDOWS\system32\cmd.exe - run
C:\ggjava>start java wuc.naming.WUCNamingServer
C:\ggjava>start java wuc.messaging.Theater 4040
C:\ggjava>start java wuc.messaging.Theater 4041
C:\ggjava>start java wuc.messaging.Theater 4042
C:\ggjava>start java wuc.messaging.Theater 4043
C:\ggjava>compile
C:\ggjava>java salsa.SalsaCompiler bhp/*.salsa
Salsa Compiler Version 0.5: Reading from file bhp/BB.salsa . . .
Salsa Compiler Version 0.5: Salsa program parsed successfully.
Salsa Compiler Version 0.5: Salsa program compiled successfully.
C:\ggjava>javac bhp/*.java -source 1.5
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
C:\ggjava>run
C:\ggjava>java bhp.BB uan://jawaan:3030/ rmsp://jawaan:4040/ rmsp://jawaan:4042/ rmsp://jawaan:4043/

C:\WINDOWS\system32\java.exe
Theater started on port: 4041
Worker 1 created
Worker 1 : uan://jawaan:3030/w1 invoked for Busy Beaver Computation
Worker 1 performing Busy Beaver Computations ...
Worker 1 completed performing Busy Beaver Computations.

C:\WINDOWS\system32\java.exe
Theater started on port: 4042
Worker 2 created
Worker 2 : uan://jawaan:3030/w2 invoked for Busy Beaver Computation
Worker 2 performing Busy Beaver Computations ...
Worker 2 completed performing Busy Beaver Computations.

C:\WINDOWS\system32\java.exe
Theater started on port: 4043
Worker 3 created
Worker 3 : uan://jawaan:3030/w3 invoked for Busy Beaver Computation
Worker 3 performing Busy Beaver Computations ...
Worker 3 completed performing Busy Beaver Computations.

C:\WINDOWS\system32\java.exe
Request: GET /w1 URNP/1.0
Response : URNP/0.1.200 Location Found
rmsp://jawaan:4041/w1

Request: GET /w2 URNP/1.0
Response : URNP/0.1.200 Location Found
rmsp://jawaan:4042/w2

Request: GET /f URNP/1.0
Response : URNP/0.1.200 Location Found
rmsp://jawaan:4040/f

Request: GET /w3 URNP/1.0
Response : URNP/0.1.200 Location Found
rmsp://jawaan:4043/w3

Request: GET /f URNP/1.0
Response : URNP/0.1.200 Location Found
rmsp://jawaan:4040/f

Request: GET /f URNP/1.0
Response : URNP/0.1.200 Location Found
rmsp://jawaan:4040/f

```

Figure 4.2: Result obtained from the execution of the *Most Productive Turing Machine* application. The three windows on the right hand side are the worker theaters that display messages about the status of computation and result. The window on the top left hand side shows the computation from the perspective of the farmer. The farmer displays that the results. The naming server is seen in the bottom right window. The application scripts are run by the user in the bottom left window.

4.2.2 Distributed Search

The *Distributed Search* problem involves finding an element across a given set of elements. The problem involves distributing the set across multiple resources so that each resource looks for the number in the subset assigned to it.

As each of the multiple resources look for a number in the assigned subset, distributed approach is generally faster as compared to a single machine looking through an entire set. A single machine takes a linear time that depends upon the size of the data set. As the data set increases in size, the single machine will take correspondingly more time to find the solution. However, by adding n resources in a distributed search, the search time is reduced by a factor n . For distributed search, some time is spent on initial setup and final reporting due to network latencies. Hence,

Search time for distributed search = Search time for single resource search / n + Time required for setup and reporting.

For searching over large data sets, time spent on initial setup and final reporting due to network latencies is small as compared to the time saved by distributing the search over multiple machines.

Hence, the distributed search approach is efficient that a single resource approach when the data sets are substantially large. It should not be used for small data sets, where a single resource search would be more efficient.

Initially the `DistributedSearch` calls the `initializeFarmer()` method of the `FarmerWorker` for initialization during which it passes the list of resources available for computation to the `FarmerWorker`.

Then it calls the `createWorkers()` method with appropriate argument to indicate the number of workers that would be needed by the application. It then creates those workers and the farmer and calls the `initialize()` method of itself to start the computation.

The `initialize()` method then distributes the work using `FarmerWorker` work distribution strategy and calls its `compute()` method in various workers. The workers perform computation and report the results back to the farmer.

In the `collector()` method of the farmer a track is kept using the `FarmerWorker` of the workers that have finished the computation. Once all the workers report the results back, the farmer publishes the final results and ends the computation.

In the Figure 4.4, you can see the result of running the search application. The three

windows on the right hand side are the worker theaters that display messages about the status of computation and result. Here, one of the workers found the element while others did not. The window on the top left hand side shows the computation from the perspective of the farmer. The farmer displays that the element was found as one of the workers found it. The naming server is seen in the bottom right window. The application scripts are run by the user in the bottom left window.


```

behavior DistributedSearch {

C1: private FarmerWorker fw = new FarmerWorker();

void Collector(Boolean found) {

C2: fw.decWorkerCount();

if(found.booleanValue() == true) {
    standardOutput<-println(" The number was found");
}

C3:     if(fw.getWorkerCount() == 0)
        ...
}

void init( ... ) {
    ...
C4:     DistributedSearch search =
        new DistributedSearch(new UAN(fw.getWorker(i)));
C5:     fw.incWorkerCount();
        search<-compute( ... );
        ...
}

void act(String[] args ) {
C6:     fw.finit(args);
C7:     fw.createWorkers(3);

        for(i = 0; i < 3; i++) {
            DistributedSearch search = new DistributedSearch();
C8:             search<-bind( fw.getWorker(i), fw.getWorkerUAL(i))@
                ...
        }
        ...
}

// Other methods of DistributedSearch class
}

```

Figure 4.3: The DistributedSearch class implements the *DistributedSearch* application. C1: An instance of FarmerWorker work distribution strategy is created. C2,C3,C4,C5,C8: Various methods of FarmerWorker class are called. C6: FarmerWorker is initialized. C7: FarmerWorker is called to create required number of workers.

The figure displays a distributed search application with four terminal windows. The top-left window shows the farmer's perspective, receiving requests and responses from three workers (w1, w2, w3) and reporting 'Location Found' for each. The top-right window shows a worker theater (port 4041) where Worker 1 successfully finds the number 133. The middle-left window shows the farmer's perspective after 3 workers are created, with Worker 1 reporting the number was found and the computation finished. The middle-right window shows a worker theater (port 4042) where Worker 2 did not find the number 133. The bottom-left window shows the user's command-line execution, including compiling and running the Salsa compiler and naming server. The bottom-right window shows a worker theater (port 4043) where Worker 3 did not find the number 133.

```

C:\WINDOWS\system32\java.exe
Request: GET /w1 UANP/1.0
Response : UANP/0.1 200 Location Found
rmsp://javaan:4041/w1

Request: GET /w2 UANP/1.0
Response : UANP/0.1 200 Location Found
rmsp://javaan:4042/w2

Request: GET /f UANP/1.0
Response : UANP/0.1 200 Location Found
rmsp://javaan:4040/f

Request: GET /w3 UANP/1.0
Response : UANP/0.1 200 Location Found
rmsp://javaan:4043/w3

C:\WINDOWS\system32\java.exe
Theater started on port: 4041
Worker 1 created
Worker 1 : uan://javaan:3030/w1 invoked for Search Computation
Worker found the number 133

C:\WINDOWS\system32\java.exe
Theater started on port: 4042
Worker 2 created
Worker 2 : uan://javaan:3030/w2 invoked for Search Computation
Worker did not find the number 133

C:\WINDOWS\system32\java.exe
Theater started on port: 4043
Worker 3 created
Worker 3 : uan://javaan:3030/w3 invoked for Search Computation
Worker did not find the number 133

C:\WINDOWS\system32\cmd.exe - strun
C:\ggjava>compile
C:\ggjava>java salsac.SalsaCompiler Search/*.salsa
Salsa Compiler Version 0.5: Reading From file Search/Search.sals
Salsa Compiler Version 0.5: Salsa program parsed successfully.
Salsa Compiler Version 0.5: Salsa program compiled successfully.
C:\ggjava>javac Search/*.java -source 1.5
C:\ggjava>setup
C:\ggjava>start java wuc.naming.WUCNamingServer
C:\ggjava>start java wuc.messaging.Theater 4040
C:\ggjava>start java wuc.messaging.Theater 4041
C:\ggjava>start java wuc.messaging.Theater 4042
C:\ggjava>start java wuc.messaging.Theater 4043
C:\ggjava>run
C:\ggjava>java Search.Search uan://javaan:3030/ rmsp://javaan:404
n:4041/ rmsp://javaan:4042/ rmsp://javaan:4043/

```

Figure 4.4: Result obtained from the execution of the *Distributed Search* application. The three windows on the right hand side are the worker theaters that display messages about the status of computation and result. Here one of the workers found the element while others did not. The window on the top left hand side shows the computation from the perspective of the farmer. The farmer displays that the element was found as one of the workers found it. The naming server is seen in the bottom right window. The application scripts are run by the user in the bottom left window.

CHAPTER 5

Analysis

5.1 Overview

The Generic Framework for Distributed Computation is implemented in Java, and can be used by Java as well as SALSA applications. With the development of the Generic Framework for Distributed Computation, we present the analysis of application development with SALSA and Java.

In the following sections, we present the SALSA programming language and architecture. We present the enhanced SALSA architecture with Generic Framework for Distributed Computation. We then present the analysis based on the key evaluation metrics established earlier.

5.2 Analysis of SALSA with GFDC

In this section, we will describe the current SALSA architecture and the enhanced architecture, by adding a generic framework for distributed computing, to SALSA. To compare the advantages and disadvantages, we will consider the **Most Productive Turing Machine** example, presented in the earlier section.

5.2.1 SALSA Architecture

SALSA is specifically developed for programming dynamically reconfigurable distributed applications by the researchers at the *Worldwide Computing Laboratory at Rensselaer Polytechnic Institute*. In the context of SALSA and WWC the current architecture for application development is as follows:

An application programmer develops the application in SALSA. The programmer can also use Java classes in SALSA. The SALSA part is compiled into Java and combined with other Java code if present along with Actor Libraries to get class files. The class file is then executed.

In this architecture there is SALSA language and Java library level support for distributed application development. Therefore, every time the developer needs to write code for distributed computation specific tasks. This result in inefficient implementations and considerable development time spent in distributed computing specific tasks such as

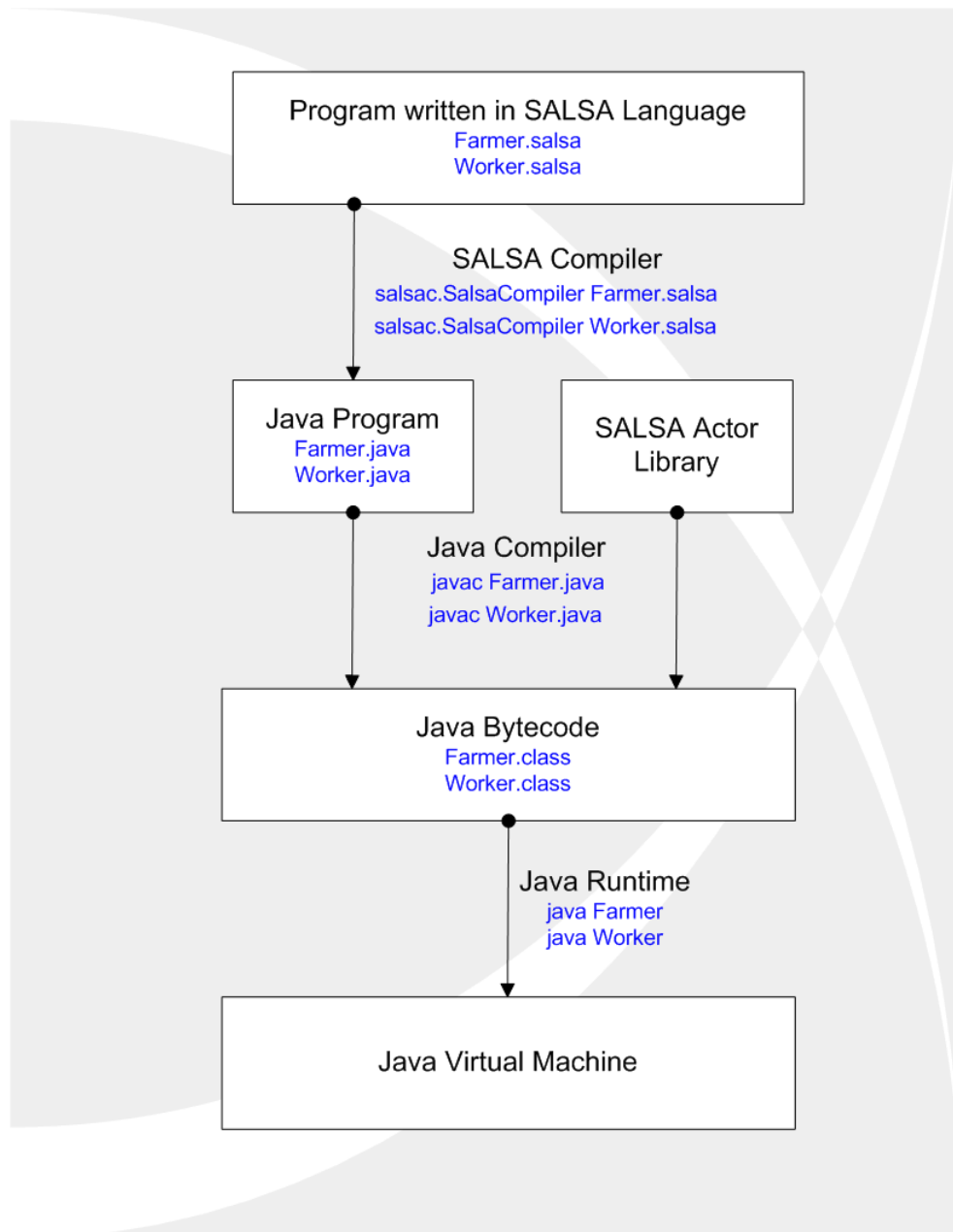


Figure 5.1: SALSA Architecture. In this architecture the application developer has to code all the things required for distributed computation along with the application itself. The application programs are then compiled by the SALSA compiler and combined with the SALSA actor library to give the Java byte-code, which is then executed.

efficient work distribution, network setup, load balancing, dynamic reconfiguration, and replication, rather than actual application development.

For example, in the case of `MostProductiveTuringMachine` the application programmer had to develop two source files, `Farmer.salsa` and `Worker.salsa`. The programmer had to take care of proper setup of resources. The programmer also had to manage these resources along with creating new workers for the computation. The programmer had to keep track of resources and allocation and do some load balancing. This resulted in lot of time spent on distributed computing specific tasks instead of actual application development.

In the next section, we will consider how the application development was made easier by adding generic framework for distributed computing to SALSA.

5.2.2 Enhanced SALSA Architecture with GFDC

In the context of SALSA and WWC, the enhanced architecture for application development is as follows:

The programmer develops the application in SALSA. Then the SALSA part is compiled into Java and combined with other Java code if present along with actor libraries to get Java byte-code as was the case earlier.

The major difference in this architecture is the addition of the *Generic Framework for Distributed Computation*, as shown in Figure 5.2. This gives the application developer efficient and tested code, to perform distributed computing specific tasks, such as efficient work distribution, network setup, load balancing, dynamic reconfiguration, and replication.

This allows the application developer to concentrate on actual application development instead of distributed computing issues. Finally, as was the case earlier, the Java byte-code is then executed. Therefore, there are no run-time changes required in the architecture.

With this architecture in the case of `MostProductiveTuringMachine` the application programmer has to develop only one source file, `MostProductiveTuringMachine.salsa`. The programmer does not have to take care of proper setup of resources. The programmer also does not need to manage these resources, along with creating new workers for the computation. Keeping track of resources and allocation is taken care by the architecture. This allows the programmer to spend more time on actual application development.

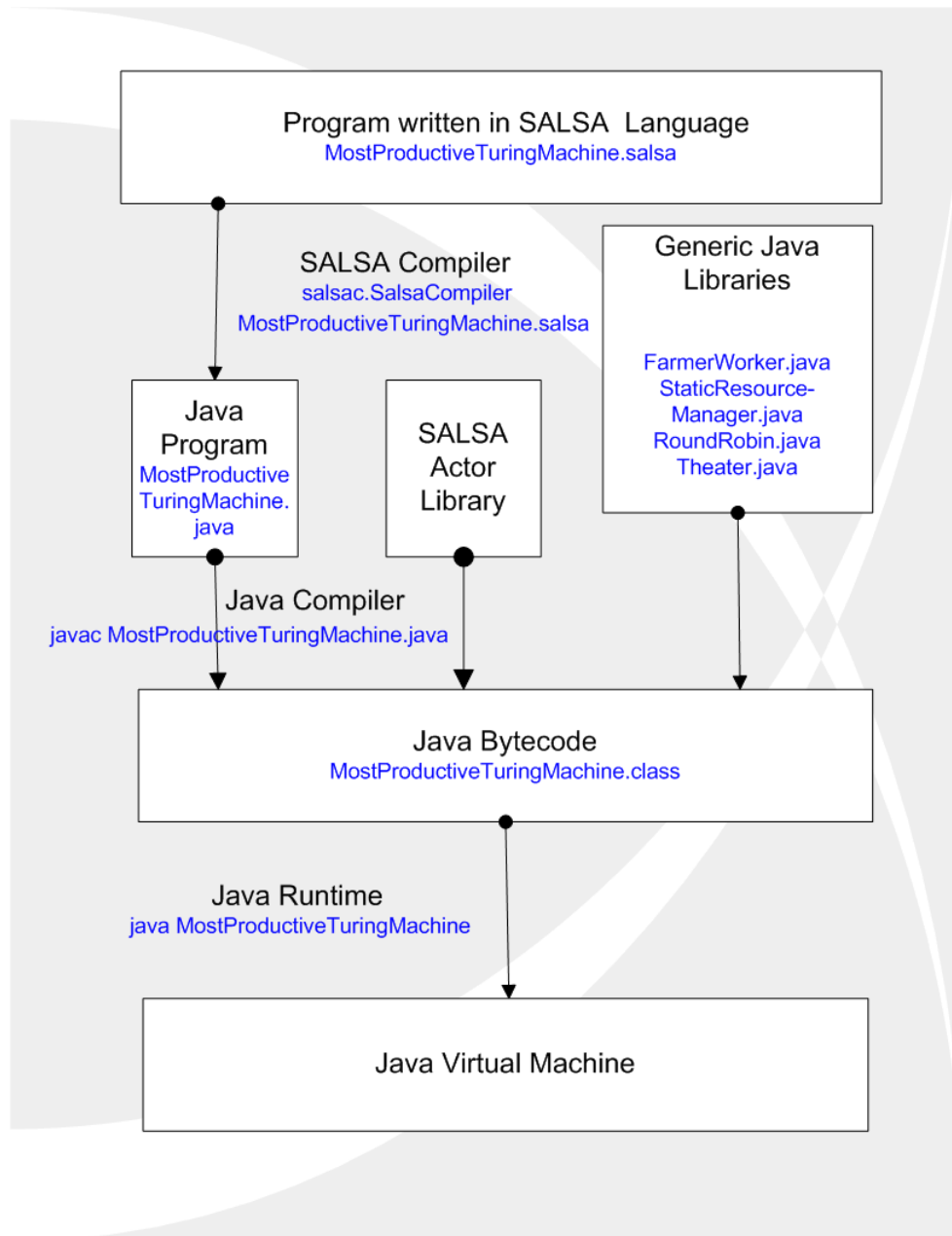


Figure 5.2: Enhanced SALSA Architecture with GFDC. In this architecture, the application developer only has to code the things required by the application. The application programs are then compiled by the SALSA compiler, and combined with the SALSA actor library and *Generic Framework for Distributed Computation* implementing libraries, which take care of distributed computing issues. The programmer then executes the generated class files.

5.2.3 Comparison of Two Approaches Using Evaluation Criteria

The earlier sections described the existing SALSA architecture and the enhanced SALSA architecture with GFDC. In the following section, we discuss the advantages and disadvantages of the two approaches.

	MPTM in SALSA	MPTM in SALSA with GFDC
Number of User Files	2	1
Lines of Code	388	299
Code Simplicity	Difficult	Medium
Implicit Support for Genericity	No	Limited to Java Code
Ease of Application Development	Difficult	Medium
Ease of Change/Scalability	Major Modifications	Minor Modifications
Ease of Reconfiguration	Medium	Medium with current implementation

Table 5.1: Table comparing the relative advantages and disadvantages of implementing the Most Productive Turing Machine (MPTM) in SALSA as opposed to SALSA with Generic Framework for Distributed Computation (GFDC). From the comparisons it is clear that the SALSA with GFDC is easier and faster to code. The program size is small, and programmer does not need to know the details of the distributed computing issues.

As shown in Table 5.1, the application using SALSA with GFDC is easier to develop. It has less number of source files and code to be written by the programmer. The lines of code shown in the table does not include the application specific code in Java that is common to both the architectures. The lines of code refer to the SALSA code that is used for distributed computation. The code that the programmer no longer has to write, is the critical code for aspects related to distributed computing which he/she may or may not be familiar with.

The GFDC implementation also has implicit support for genericity at Java level of the libraries. Hence, it is easy to change around components and scale well. As the critical code required for distributed computing is already supplied by the library application development is easy.

The implementation does not give any significant performance improvements. With the implementation of `DynamicResourceManager`, the application will give a better performance due to better reconfiguration and profiling that would be available through the

`DynamicResourceManager`. Further improvements to the GFDC framework will make distributed application development more easier and faster.

CHAPTER 6

Related Work

6.1 Overview

In this chapter, we present the work related to our *Generic Framework For Distributed Computing*. The framework addresses the distributed computing issues at language, library, and run-time levels. The related work chapter is also organized with the same structure. It presents and compares related work at the same three levels:

- *Programming Language Level:* Related programming languages and programming language features are presented and compared. The languages are discussed in the context of the support for generic programming [38] and distributed computing.
- *Library Level:* Generic libraries and components have been developed to enable application developers to use off-the-self code, to reduce development time. We consider the design philosophies and the applications areas of these libraries.
- *Run-Time Level:* Related run-time systems that hide the complexities of distributed network computations from the user applications are presented.

6.2 Programming Languages

In this section, we discuss various languages for their support for generics and distributed computing. We present the enhanced language level support for distributed computing which can make the development of distributed programs easier and more efficient. For the languages that are presented, we discuss the support for both *Parametric Polymorphism* and/or *Subtype Polymorphism*, which enables development of generic code. In the following subsections, we will discuss some of these languages and their relevance to the generic framework for distributed computing is discussed at the end.

6.2.1 C++

C++ [19] provides one of the most comprehensive sets of language features for generic programming. In particular, it provides support for operator overloading and templates. The template mechanism in C++ is very advanced. C++ templates support partial specialization and template meta-programming. These features are helpful and

essential to write efficient and reusable generic code. C++ also provides *type alias* using `typedef` declarations. C++ also provides *implicit instantiation* in which type parameters can be deduced without requiring explicit syntax for instantiation.

C++ support for generics does have some limitations. C++ does not contain any explicit mechanism for constraining templates. This facility is present in Java and C# [18, 33, 41]. Additionally, C++ generics does not allow for *separate compilation* which is allowed in Java. Separate compilation enables generic functions to be type-checked and compiled independently from their use. A very good comparison of C++ and Java support for generics is present in [29]. Overall, C++ support for generics is very comprehensive and better than Java.

However, with respect to distributed computation, Java provides more functionality than C++. It provides various facilities like *Remote Method Invocation* (RMI) [55], *Serialization* and *Reflection* [34]. Both C++ and Java provide *Common Object Request Broker Architecture* (CORBA) [16, 42, 42] libraries for distributed application development.

6.2.2 Java

Java [32] supports subtype polymorphism as all the objects are derived from the root class called `Object`. This allowed the development of generic libraries for *Containers* and *Algorithms* by assuming the argument object of the type `Object` and then casting the actual object as per the user requirements. This works fine but has two major drawbacks:

- Casting can get complex and complicated.
- Incorrect use of casting can result in run-time exceptions.

There have been numerous proposals [1, 54, 50, 13, 10] for adding genericity to Java. Adding genericity to the Java programming language gives expressiveness and safety, by making type parameters explicit, and making type casts implicit. This is crucial for using libraries such as collections, implementing generic utility classes in a flexible and safe way by removing the need to cast, and allowing more errors to be caught at compile time.

In a way, this is similar to C++ template implementation without the support for features like partial specialization and template meta-programming. However, a new language construct called wildcards, is one of the features that is present in Java but not in C++. This is possible due to the use of `Object` as super class of all classes in Java, which is not possible in C++.

Wildcards [11] increase the flexibility of object-oriented type systems with parameterized classes. Based on the notion of use-site variance, wildcards provide a type-safe abstraction over different instantiations of parameterized classes by using `?` to denote unspecified type arguments. Thus, they essentially unify the distinct families of classes often introduced by parametric polymorphism.

6.2.3 SALSA

SALSA (Simple Actor Language System and Architecture) is an actor-oriented programming language that is useful for developing distributed applications that need dynamic reconfiguration. Some of the major features of SALSA are:

- First class support for unbounded concurrency.
- Communication amongst actors using asynchronous message passing.
- Universal naming model.
- Support for actor migration.
- Location-transparent message sending using UAN (Universal Actor Names).
- High level abstractions for programmers to facilitate coordination of concurrent activities like:
 - Token-passing continuations.
 - Join blocks.
 - Named tokens.
 - First-class continuations.

SALSA is based on an Actor model [2] of concurrent computation for distributed systems, which provides a unit of encapsulation for both the state and a thread of control, that manipulates that state. Actors communicate amongst each other by exchanging messages asynchronously.

In response to a message, an actor may perform one of the following actions:

- Change its current state and possibly also change its behavior.
- Migrate to a new location.

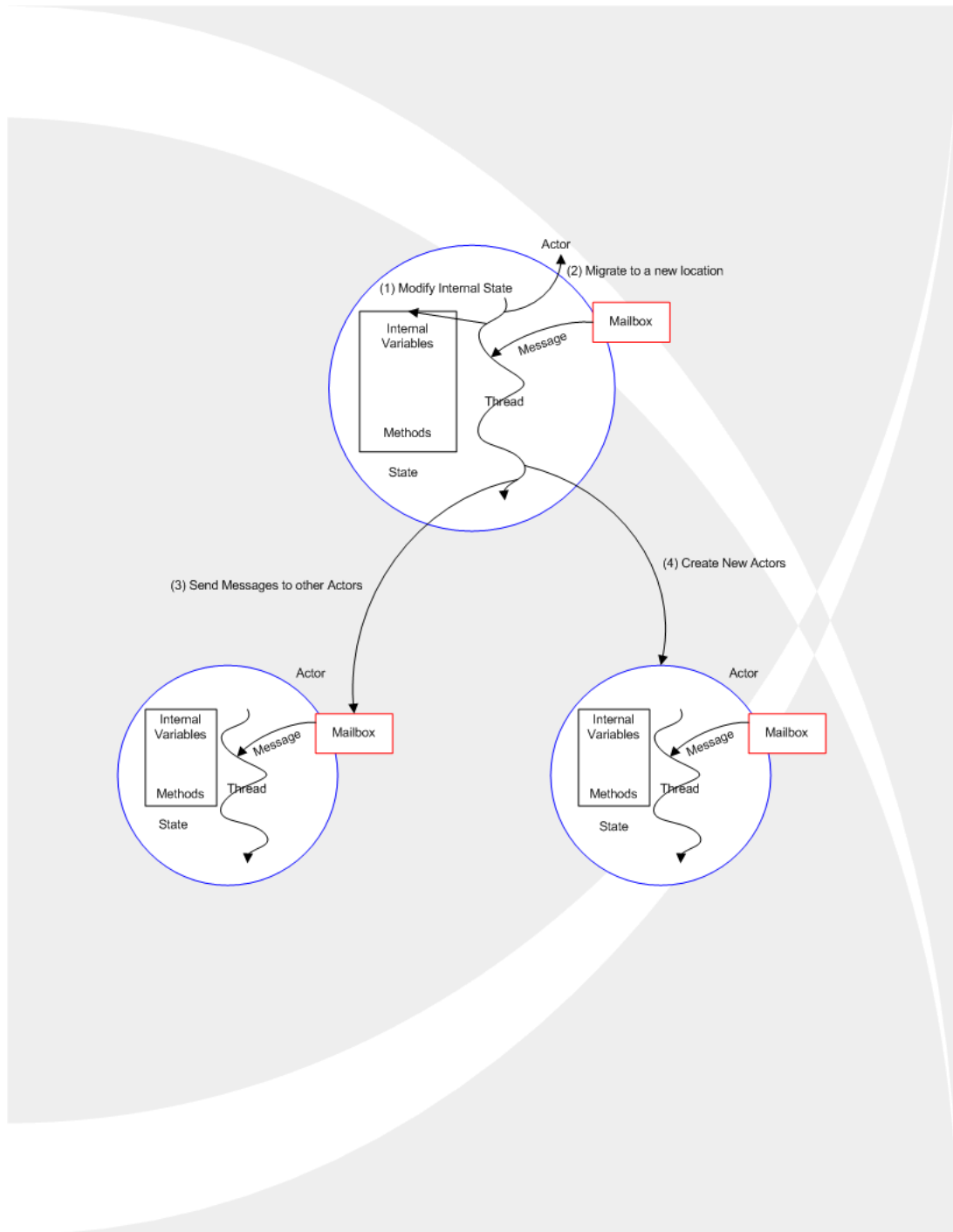


Figure 6.1: An actor upon receiving a message either (1) Changes it's internal state, (2) Migrate to a new location, (3) Send messages to other Actors or (4) Create new Actors.

- Send messages asynchronously to other actors.
- Create new actors with a specified behavior.

6.2.4 Discussion

In addition to the three languages discussed earlier, there are other programming languages that support distributed computing and generics. Programming languages like *Eiffel* [47], *Haskell* [40] and *ML* [49] support generics to varying degrees. Programming languages like *Erlang* [20], *Nomadic Pict* [63, 52], and *JoCaml* [39], are specifically designed to support distributed and concurrent computations.

Chapter 2 presented a generic framework for distributed computing, whose implementation in generic Java was shown in Chapter 3. However, the implementation is not limited to generic Java and is implementable in these languages. This would enable the implementation to make use of the relative strong points of various languages with respect to genericity and support for distributed computation to effectively implement the framework.

The generic Java implementation was limited by the support for genericity that Java has as compared to C++. However, Java's better support for distributed computations was the motivating factor in deciding for implementation in generic Java. C++ implementation of this framework would be more comprehensive and effective due to its support for genericity. However, much code would have to be written for distributed computing-specific tasks, due to the limited support of those tasks, in C++. SALSA was used to develop the applications that use the framework used in generic Java. With further enhancement to SALSA discussed in the future work, implementation of this framework in SALSA should be possible. It is also desirable as SALSA has extensive support for distributed computing and with addition of generics, it would be one of the ideal languages to develop distributed applications.

6.3 Generic Libraries

Here we consider the generic libraries and components that have enabled application developer to use off-the-self code to reduce development time. For designing the generic components the developer has to identify and define a family of abstractions that are all related by a common set of requirements [56]. This is the critical part of developing generic components. These sets of requirements should be as general as possible but still

restrictive enough that programs can be written efficiently for different abstractions that satisfy the requirements. In the following subsections, we will discuss some of the widely used generic libraries.

6.3.1 Standard Template Library (STL)

STL [58, 6, 3] is a framework built on the foundations of generic programming. The framework involves the actual library as well as the definitions of abstractions like containers, iterators, algorithms, function objects, adaptors, and allocators. STL enables faster program development by providing commonly required data structures like list, vector, and hash table in the form of various containers. Common data structures can be derived from the basic containers using various adaptors. At the same time, it provides commonly used algorithms for sorting and searching. The functionality of the algorithms can be customized by using function objects. The containers and algorithms can work on each other through various iterators. Due to the extensive use of templates in STL, the algorithms and containers can be used on any type, which satisfies their requirements.

6.3.2 Boost Graph Library (BGL)

BGL [9, 60, 59] is a collection of containers and algorithms for performing common graph related operations. BGL provides a generic interface to the user, to access the structure of the graph but at the same time, hides the details of the graph data structure implementation. It presents an open interface, so that any graph library that implements this interface will be interoperable with the BGL generic algorithms, and with other algorithms that also use this interface. Towards this goal, BGL provides some general purpose graph classes that conform to this interface and allows the user to add newer classes that are better for certain situations. The only restriction on these classes is that they should conform to the interface defined in the BGL.

6.3.3 Loki

Loki [45] is a C++ library that contains flexible implementations of common design patterns. The patterns are implemented using a generic programming methodology. Some of the common components implemented in Loki are generalized functors, singletons, smart pointers, object factories, abstract factories, and visitors.

In Loki, C++ templates are used to implement a policy-based design. A policy establishes an interface for a specific issue. The actual implementation of the policy can

vary as long as the policy interface is maintained. A class with complex behavior can be constructed by using multiple smaller and simpler classes. These smaller classes take care of only one behavioral or structural aspect. By combining smaller policies in different ways, a large number of complex behaviors can be generated. This makes Loki a flexible and highly reusable library.

6.3.4 Discussion

Libraries such as STL, BGL, and Loki can be used internally by the distributed applications. However, they do not provide abstractions necessary for distributed computing-specific tasks. They also do not implement any distributed algorithms or data structures [46, 5], that can be used directly. These libraries are important, as the understanding of their design is useful while identifying the abstractions in the distributed context.

6.4 Run-Time Systems

In the context of distributed applications development, run-time systems are important, as they hide the complexities of distributed computations from user applications. Various distributed computing issues like efficient work distribution, load balancing, dynamic reconfiguration, and profiling, are handled at this level. Moreover, some of the run-time systems hide the differences in the hardware, operating system and network setup, and present a consistent virtual computer which is easy to program. In the following subsections, we will discuss some of these run-time systems and their relevance to the generic framework for distributed computing is discussed at the end.

6.4.1 Globus

Globus [31, 25, 26] enables the application of *Grid* [27, 24] concepts to scientific and engineering computing. Grid enables securely and effectively exposing high-end computers, networks, databases, and scientific instruments owned and managed by multiple organizations for collaborative use. Grid applications are data and/or computation-intensive. In addition, they often require secure resource-sharing across organizational boundaries. These issues are currently not easily handled by Internet and Web infrastructures. The *Globus Toolkit* addresses these problems. It is a collection of services and components than can be used independently or together to develop applications in this framework. Globus identifies three types of users:

- *Grid Builders* These users actually create the production grid computing environments which are directly or indirectly used by the other two users. NASA's Information Power Grid [51], the European DataGrid Project [21] and the ASCI Distributed Resource Management Project [4] are examples of grid builders.
- *Application Framework Developers* These developers build the software frameworks using the Globus services. The frameworks facilitate the development and execution of different kinds of applications. CAVERNsoft [14] framework for tele-immersive applications, Condor-G [15] and Linear Systems Analyzer [44] are the examples of frameworks developed by application framework developers.
- *Application Developers* Application developers use Globus services to develop innovative Grid-based applications. These applications are developed either directly or by using grid-enabled tools. Remote supercomputing, distributed supercomputing, and supercomputer-enhanced scientific instruments are examples of these applications.

6.4.2 Java Virtual Machine

Java Virtual Machine (JVM)[43] is a run-time environment required for running Java programs. Java programs are interpreted by the interpreter that is built into the JVM. The Java source file is compiled into a class file that is then interpreted by the JVM during execution. The source code is compiled into a Java byte-code class file that is independent of any machine architecture or operating system. Thus, the same byte code can be used across multiple machines with different hardware, software architectures, and systems, as long as they have the required JVM.

This is critical in distributed systems where the applications use large number of resources to perform computations. As the JVM presents a consistent virtual machine, the distributed application that is coded in Java does not have to deal with the different machine architecture or operating systems. This allows the distributed application to use maximum number of available resources.

The JVM runs on the *Operating System*, and has *Security Manager* built into it. Therefore, an application running in JVM can rarely crash or attack the system. The most that it can do is to crash the JVM, which would never be catastrophic unlike the crashing of the operating system itself. This is also critical for distributed applications as they use the resources that are volunteered by the different users. So unless the users are

sure about the safety of the applications that are running, they would not be willing to volunteer their resources.

JVM is one of the important facilitator of creating distributed applications in the Java programming language. It enables using resources with different types of hardware and software in a safe manner.

6.4.3 World-Wide Computer

The *World-Wide Computer* (WWC) is the run-time system for programs written in the SALSA programming language. Just like Java programs run on the JVM abstraction, SALSA programs run on the *Theater* abstraction. In Java, JVM presents a homogeneous view of the underlying heterogeneous hardware architectures and operating systems. Similarly, in SALSA, an agent-based language for distributed programming, the theaters provide a run-time layer between actors and JVMs. A theater hosts multiple actors, and provides support for message passing, remote communication, and other features specific to SALSA.

As the theater is ultimately running on a JVM, an agent can run on any machine that has a theater running on it, irrespective of its hardware and software architecture. This enables a computation to utilize potentially large number of computers throughout the world as long as they have supporting JVM and WWC Theater running on them. This enables SALSA/WWC, to harness the power of a large number of computers over the Internet to perform complex distributed computations.

6.4.4 Internet Operating System

As the application's size and complexity increases, it becomes difficult for the programmer to deal with issues like profiling, efficient resource allocation, dynamic reconfiguration, and automatic migration, to make the best use of the available resources. Just as the Operating System provides the functionality for process and threads on a single machine, the Internet Operating System (IOS)[35, 17] is used to deal with these issues in the context of SALSA/WWC.

The IOS is implemented in a decentralized way on top of the Java Virtual Machines running on different physical machines. It efficiently manages resources (CPU, memory, and network) available to applications running on WWC. It enables efficient re-configuration of applications during run-time to take maximum advantage of available

resources, and at the same time does load balancing to make sure that all theaters get optimal number of actors, and no actor starves for execution environment and resources.

Some of the major issues that IOS addresses are similar to what a *Operating System* does in the context of single computer, and what a *Network Operating System* does in case of a network:

- IOS is responsible for different sets of resources like procession power, memory, and network.
- Provide fair set of resources to competing set of actors or computations.
- Deal with issues unique to the Internet-scale applications like:
 - Security.
 - Soft and hard failure of resources.
 - Profiling and scheduling dynamic resources.

6.4.5 Discussion

Along with the run-time systems discussed above there are other significant systems like the *Microsoft .NET Framework* [48] with its *Microsoft Common Language Runtime* (CLR). This framework supports development of distributed applications and web services using a variety of languages. The C# and C++ languages also have support for genericity, which makes .NET an easy framework to develop distributed applications. A C# or C++ implementation of the generic framework for distributed computing on the .NET framework can be useful.

However, .NET is specific to Microsoft framework and is relatively new. Java using the *J2EE* [36] framework is a widely used and stable implementation. As discussed earlier, it has extensive support for distributed computing and has recently been enhanced with the addition of generics. Hence, Java on J2EE framework was the choice for the sample implementation of the generic framework for distributed computing.

Globus is fast gaining popularity, and the lower level abstractions of the generic framework for distributed computing can be employed to use the facilities provided by Globus. This would give the application developers using the generic framework for distributed computing, a wider choice of the target execution environments, for their distributed applications.

CHAPTER 7

Future Work

7.1 Development of Abstractions for GFDC

The Generic Framework for Distributed Computing identifies the abstractions at multiple levels as well as the interdependencies between these abstractions. In the sequential computation paradigm, different types of abstraction were incrementally identified in generic libraries such as STL, BGL, and Loki.

Similarly, due to the intrinsic complexities of distributed computation, more abstractions will be required to define certain class of concepts. These abstractions may or may not be related to the abstractions defined in GFDC. However, the Generic Framework for Distributed Computing can act as a starting point for identifying these abstractions.

7.2 Implementation Of All Abstractions Of GFDC

The Generic Framework for Distributed Computing is a comprehensive set of abstractions, that can be used to model a range of problems, by correct combination of different abstractions.

This thesis has presented the abstractions and a sub-set of implementation of these abstractions. For the effective and widespread use of this framework, it is necessary to comprehensively implement all the abstractions.

When all the sets of abstractions are available, the programmer would be able to easily mix and match the combinations of the implementations of different abstractions. This would enable the programmer to get the best possible implementation of the distributed application.

7.3 Implementation of Generic SALSA

As discussed in Chapter 1, applications written in the SALSA programming language are compiled to Java, which enables creation of Java objects inside the SALSA behaviors @ not clear @. This allows the SALSA programmers to use the Java libraries and other existing Java classes.

The thesis presented two applications, *Most Productive Turing Machine* and *Distributed Search* that were developed in SALSA. As SALSA actors can create and use Java objects, these applications used the generic libraries and components developed in generic Java. However, even though SALSA programs are translated in Java, it still does not have support for generics. This limits the use by SALSA applications of libraries developed in generic Java using GFDC.

A generic SALSA implementation that will support templates would allow SALSA programmers to fully utilize the generic components and libraries developed using GFDC. Moreover, it would enable the SALSA programmers to create generic libraries in SALSA itself. It would also enable programmers to develop libraries leveraging SALSA-specific features like continuations, support for migration, and dynamic reconfiguration. These libraries can also use the functionality provided by generic Java libraries developed using GFDC, as the SALSA libraries written in SALSA code will eventually be translated to Java.

Figure 7.1 shows the architecture of the generic SALSA implementation.

The difference between Figure 5.1, which shows the current architecture of SALSA and Figure 5.2, which shows the enhanced SALSA architecture, is the addition of the generic framework for distributed computing. This enables the Java and to some extent, the SALSA applications to make use of the generic libraries made available using GFDC.

Now with the proposed generic SALSA architecture, programmers will be able to write generic SALSA libraries. They will also be able to fully utilize the functionality provided by the generic Java code, as now they will be able to write generic Java code in SALSA.

As shown in the Figure 7.1, the generic SALSA code and libraries will be compiled by the generic SALSA compiler to generate generic Java code. Then this code along with the components and libraries developed in generic Java using the framework will be combined by the generic Java compiler to produce the Java byte-code that can be executed.

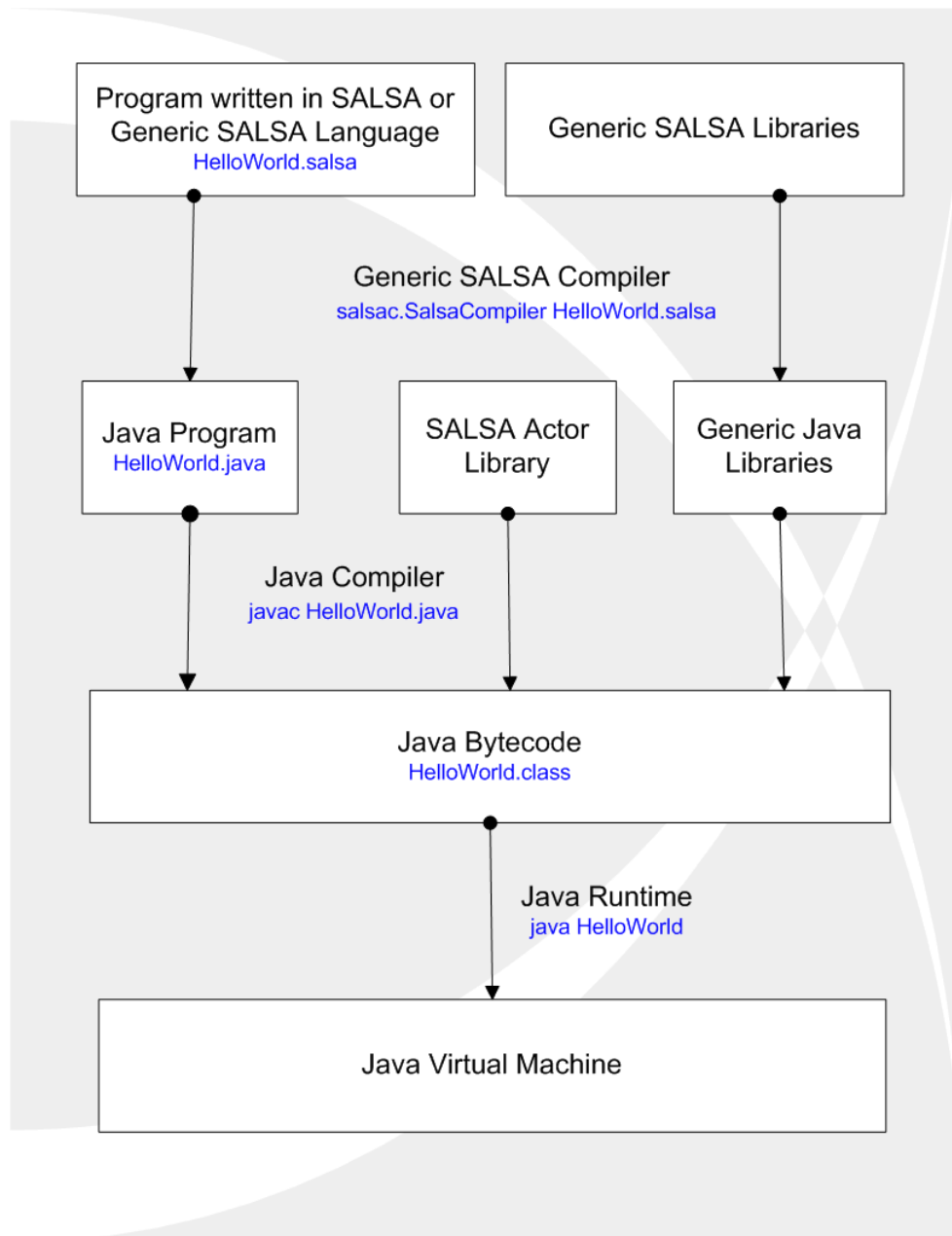


Figure 7.1: Programming using Generic SALSALSA Libraries. In this architecture, the programmer will be able to write generic code in SALSALSA and use the generic SALSALSA code from the libraries. The generic SALSALSA compiler will compile the code into generic Java. The Java compiler will combine this code with the generic Java code developed to give a class file that can be executed. The programs in this architecture will be smaller, simpler, and more efficient, with less development time.

7.4 Implementation of GFDC in other Languages

This thesis has implemented the generic framework for distributed computing in Java. However, the abstractions that are developed as part of the framework are language-independent.

Hence, the framework should be implemented in different languages like C++ and C#. This will allow application developers in those languages to make use of this framework. At the same time, the experiences gained while implementing the framework in different languages will be useful in validating and refining the framework.

CHAPTER 8

Conclusions and Contributions

8.1 Conclusion

There is an increasing adoption of the distributed computing paradigm, to solve computationally intensive problems that would otherwise take a long time to solve on a single computer. This thesis identified a major problem being faced in programming distributed applications, that considerable time is being spent in developing distributed computing specific tasks, rather than actual application development. The users who may not familiar with the distributed computing specific tasks, find it difficult to develop such applications, or develop inefficient applications even after a long development time.

This thesis identified and developed abstractions to enable designing and developing generic components and libraries for distributed computations. The *Generic Framework for Distributed Computations* that was developed is applicable to a broad range of distributed applications implemented using different technologies and programming languages. The thesis used a *Generic Software Design Methodology* to design these abstractions to allow for maximum reuse, flexibility, and efficiency.

This thesis provides an implementation of this framework in Java for developing distributed applications in Java. The SALSA programming language is used for a class of problems that need dynamic reconfiguration and migration. The generic libraries developed in Java were integrated with SALSA transparent to the application developer, to enable the SALSA developers make use of these libraries. The framework and its implementation make development of distributed applications easier, faster, and less error-prone.

The framework enables an average programmer with little knowledge of distributed computing issues, to develop efficient distributed applications easily and quickly. The framework and libraries allow advanced programmers, to further extend the framework and libraries, and to tackle distributed computing issues specific to their applications.

The thesis has identified the key evaluation criteria for success of the framework. These criteria were satisfied as discusses in the analysis chapter. The future work described the current implementation and possible extensions of the architecture. It also presented the case for the development of a Generic SALSA language to make maximum use of this

framework and its libraries.

8.2 Contribution

Following are the contributions of this thesis:

- Identification and design of abstractions to enable designing and developing generic components and libraries for distributed computations.
- Extension of use of generic software design methodology to distributed computing.
- Implementation in Java of the generic framework for distributed computation. This validates the framework.
- Presentation of sample applications to enable average programmers to use the framework.
- Discussion of further architecture and implementation improvements to the framework.
- Motivation of the development of *Generic SALSA* programming language to effectively address a class of problems that need dynamic reconfiguration and migration.

Additionally, applications using these libraries were presented, and advantages and disadvantages of developing these applications with or without generic libraries, were discussed.

The generic framework for distributed computation will serve as a sample implementation as well as a template for future language additions, and newer library development.

LITERATURE CITED

- [1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the java programming language. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 1997.
- [2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [3] Stepanov Alexander and M Lee. The standard template library. tech. rep. Technical report, 1995.
- [4] ASCI Distributed Resource Management Project Home Page. <http://www.llnl.gov/asci/pse/hpcs/drm/>.
- [5] Hagit Attiya and Jennifer Welch. *Distributed Computing, Fundamentals, Simulations and Advanced Topics*. The McGraw-Hill Companies, 1998.
- [6] M. H Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.
- [7] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. IETF Internet Draft Standard RFC 2396, August 1998. <http://www.ietf.org/rfc/rfc2396.txt>.
- [8] Boleslaw K. Szymanski, Patrick H. Fry, Jeffrey Nesheiwat. Twin Primes Enumeration. <http://www.cs.rpi.edu/research/twinp/>.
- [9] Boost Graph Library Home Page. <http://www.boost.org/libs/graph/doc/>.
- [10] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1998.
- [11] Gilad Bracha, Neal Gafter, and Erik Ernst. Adding wildcards to the java programming language. In *SAC04*, 2004.
- [12] Busy Beaver Project Home Page. <http://www.cs.rpi.edu/wwc/bbp>.

- [13] R. Cartwright and G. L. Steele. Compatible genericity with runtime-types for the java programming. In *Proceedings of the 1998 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 1998.
- [14] CAVERNsoft Home Page. <http://www.openchannelsoftware.org/projects/>.
- [15] Condor-G Home Page. <http://www.cs.wisc.edu/condor/condorg/>.
- [16] Corba Home Page. <http://www.omg.org>.
- [17] T. Desell, K. El Maghraoui, and C. Varela. Load balancing of autonomous actors over dynamic networks. In *Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track*, January 2004.
- [18] ECMA. C-Shaer language specification.
<http://www.ecma-international.org/publications/standards/Ecma-334.htm>.
- [19] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 2001.
- [20] Erlang Home Page. <http://www.erlang.org/>.
- [21] European DataGrid Project Home Page.
<http://eu-datagrid.web.cern.ch/eu-datagrid/>.
- [22] FightAIDS@Home Home Page. <http://fightaidsathome.scripps.edu/>.
- [23] Folding@Home Home Page. <http://www.stanford.edu/group/pandegroup/folding/>.
- [24] I. Foster. The grid: A new infrastructure for 21st century science. *Physics Today*, 5:42–47, 2002.
- [25] I. Foster and C. Kesselman. The globus project: A status report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop, pp. 4-18, 1998*, 1998.
- [26] I. Foster, C. Kesselman., and Intl J. Globus: A metacomputing infrastructure toolkit. In *Supercomputer Applications*, pages 115–128, 1997.
- [27] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. In *International J. Supercomputer Applications*, 20011.

- [28] Agha. G., I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
- [29] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the 2003 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'03)*, October 2003.
- [30] GIMPS Home Page. <http://www.mersenne.org/prime.htm>.
- [31] Globus Home Page. <http://www.globus.org/ogsa/>.
- [32] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison Wesley, 1997.
- [33] A. Hejlsberg. The C# programming language. In *Invited talk at Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2002.
- [34] Cay S. Horstmann and Gary Cornell. *Core Java Volume 1 and 2*. Pearson Education, 2001.
- [35] Internet Operating System Home Page.
<http://www.cs.rpi.edu/research/groups/wwc/io/index.html>.
- [36] J2EE Home Page. <http://java.sun.com/j2ee/>.
- [37] Java Technology Home Page. <http://java.sun.com/>.
- [38] M. Jazayeri, R. Loos, D. Musser, and A. Stepanov. Generic programming. in report of the dagstuhl seminar on generic programming. Technical report, April 1998.
- [39] JoCaml Home Page. <http://pauillac.inria.fr/jocaml/>.
- [40] S. P. Jones, J. Hughes, and Lennart Augustsson. Haskell 98: A non-strict, purely functional language. Technical report, February 1999.
<http://www.haskell.org/onlinereport/>.
- [41] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN 01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 1–12. ACM Press, 2001.

- [42] Sean Landis and Silvano Maffei. Building reliable distributed systems corba. In *Theory and Practice of Object Systems*, pages 31–43, 1997.
- [43] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.
- [44] Linear Systems Analyzer Home Page.
<http://www.extreme.indiana.edu/pseware/LSA/>.
- [45] Loki Library Home Page. <http://sourceforge.net/projects/loki-lib/>.
- [46] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc, 1997.
- [47] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
- [48] Microsoft .NET Framework Home Page.
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/netstart/html/cpframeworkrefstart.asp>.
- [49] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (Revised)*. MIT Press, 1997.
- [50] A. Myers, J. Bank, and B. Liskov. Parameterized types for java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1997.
- [51] NASA’s Information Power Grid Home Page. <http://www.ipg.nasa.gov/>.
- [52] Nomadic Pict Home Page. <http://lsewww.epfl.ch/>
- [53] Object Management Group. CORBA services: Common object services specification version 2. Technical report, Object Management Group, June 1997.
<http://www.omg.org/corba/>.
- [54] M. Odersky and P. Wadler. Pizza into java: Translating theory into practice. In *POPL 97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, 1997.
- [55] RMI technology Home Page. <http://java.sun.com/products/jdk/rmi/>.
- [56] RPI Generic Programming group Home Page. <http://www.cs.rpi.edu/~musser/gp/>.

- [57] SETI@Home Home Page. <http://setiathome.ssl.berkeley.edu>.
- [58] SGI Standard Template Library Home Page. <http://www.sgi.com/tech/stl>.
- [59] Jeremy Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
- [60] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. The generic graph component library. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 399–414. ACM Press, 1999.
- [61] C. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, April 2001.
- [62] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001. <http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf>.
- [63] Pawel Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *ASA/MA'99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents)*, 1999.