DESIGN OF A WORKFLOW MANAGEMENT SYSTEM FOR BLAST SERVERS USING JINI AND JAVASPACES

YANG LI

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2003

0-612-77715-4

Abstract

Design of A Workflow Management System for BLAST Servers Using JINI and JavaSpaces

Yang Li

BLAST (Basic Local Alignment Search Tool) is an important DNA and protein similarity tool. It can give evolution and functionality information about unknown genes by comparing them with well studied genes in a database. BLAST search may query many DNA or protein sequences against a huge collection of databases. Today, it is normal that one biological laboratory performs BLAST searches for thousands of sequences in its daily research activities. The performance of the BLAST search implementation is a bottleneck of such activities. In this thesis, we present a design of a new distributed BLAST cluster system using Jini and JavaSpaces technology. This system contains a set of BLAST servers. Each server has a partition of a database against which BLAST algorithms may be executed. This thesis focuses on designing a workflow management system which coordinates the work of distributed jobs. Jini and JavaSpaces provide an easy way to exchange program code and data over a network. We found it is easy to implement a workflow management application in a distributed computing environment using Jini technology. Although this system is for BLAST search, we found it is not difficult to modify it to fit other distributed computing tasks.

Acknowledgments

I thank my supervisor Dr Gregory Butler for his guidance, encouragement and support during my studying and finishing of this thesis.

Contents

List of Figures **ix**

List of Tables **xii**

1 Introduction **1**

1.1 Context of Thesis 1

1.2 Previous Work and Purpose of This Thesis 3

1.3 Contribution of The Thesis 4

1.4 Thesis Organization 5

2 Background **6**

2.1 BLAST - Basic Local Alignment Search Tool 6

2.1.1 Local and Global Alignment 6

2.1.2 Alignment Algorithms and Similarity Scores 7

2.1.3 BLAST Search Database and Algorithms Options 9

2.1.4 High Performance BLAST 9

2.2 Jini - A Distributed Computing Infrastructure on Java 10

2.2.1 What is Jini 10

2.2.2 How Jini Works 11

2.2.3 Architecture of Jini Service 12

2.2.4 JavaSpaces 13

2.2.5 Jini and Distributed Computing 14

2.3 Workflow and Workflow Management System 15

2.3.1 Workflow 15
2.3.2 Workflow Management System 15
2.3.3 Workflow Management System Based on Jini 16

3 Design **18**

3.1 Problem and Solution 18
3.2 System Specification 19
3.2.1 BLAST Client 20
3.2.2 Task Manager 22
3.2.3 BLAST Server 27
3.2.4 Lookup Service 28
3.2.5 Administrator 28
3.2.6 Results and Error Space 29
3.3 System Architecture 29
3.3.1 System Components and Functionalities 29
3.3.2 Administrator and Jini Adminservices 30
3.3.3 Communication between the Components 31
3.4 Module Design 34
3.4.1 BLAST Client 36
3.4.2 Task Manager 38
3.4.3 BLAST Server 38
3.4.4 Administrator 39
3.4.5 Task Manager Database 40
3.4.6 Results and Error Space 40
3.5 XML Data Format 40
3.5.1 Request 40
3.5.2 Task 40
3.5.3 Subtask 42
3.5.4 Result 42

3.6 Class Diagrams 42
3.6.1 Blastbasic Package 44
3.6.2 Blastserver Package 48
3.6.3 Blastclient Package 51
3.6.4 Taskmanager Package 55
3.6.5 Administrator Package 57
3.7 How the System Works 60
3.7.1 Register and Download Services 63
3.7.2 Client Send Request 69
3.7.3 Creating and Dispatching Task 71
3.7.4 Write Results to ResultSpace 71
3.7.5 Read Results from ResultSpace 76
3.7.6 Failure Handling 80

4 Conclusion and Future Work **86**
4.1 System Requirements and Project Status 86
4.1.1 System Requirement and Our Design 86
4.1.2 Project Status 87
4.2 Conclusion 87
4.3 Future Work 88
4.3.1 System Implementation and Integration 88
4.3.2 Load Balancing 89
4.3.3 Job Parallelism 89
4.3.4 Job Priority 90
4.3.5 Jini Administrator Service Bean 90
4.3.6 Framework for High Performance Computing Workflow in Biology 90

Bibliography **92**

Appendix **95**

A BLAST Databases **95**

A.1 General Database 95

A.2 Human Genome Blast Databases 97

B BLAST Search Algorithms **98**

List of Figures

1 An Example of Global and Local Alignment. 7
2 Two Dimensional Table of Smith-Waterman Algorithm. 8
3 Jini Protocols. 12
4 Fat proxy Class Diagram and JVM Objects 13
5 RMI Proxy Classes and JVM Objects. 17
6 Architecture of Administrator and Admin Services. 32
7 System Architecture and Interaction with Jini Lookup Service. 33
8 System Communication without Lookup Services. 35
9 Module Design. 37
10 XML Example for BLAST Request. 41
11 XML Example for BLAST Task. 41
12 XML Example for BLAST Subtask. 42
13 XML Example for BLAST Result. 43
14 Class `BasicData`. 45
15 Class `Infor` Lists All Service Properties. 46
16 `BasicService` Interface 47
17 Class Diagram for BLAST Basic Package 48
18 `BLASTEngine` with JNI Function Call 49
19 `ServerInfor` Contains BLAST Service Properties. 50
20 Class Diagram for the BLAST Server Package 52
21 Class Diagram for the BLAST Client Package 53
22 Class Diagram for BLAST Client GUI 54

23 Class Diagram for the BLAST Client Admin Service 54
24 Class TaskMangerInfor Contains BLAST Services Information 55
25 Class Diagram for the Task Manager Package 57
26 Class Diagram for the Data Classes 58
27 Class Diagram for the Message classes and Task Manager Sub-packages 59
28 Class ServiceNode Creates a Representative for each Node. 61
29 Class Diagram for the Administrator Package 62
30 Sequence Diagram for Jini Services Registration and Downloading. . . 64
31 Class JiniNode Provides Basic Jini Service and Client APIs. 65
32 The *discovered()* and *registerService()* Functions in Class JiniNode . 66
33 Class JiniTaskManger Implements the Function *createService()* to Create JiniTaskManagerService Instance. 67
34 Class JiniTaskManger Implements the *getService()* Function to Download All Available JiniBLASTServices. 68
35 Sequence Diagram for Sending a Request to the JiniTaskManager . . 69
36 JiniBLASTClient Sends Requests to the JiniTaskMangerServiceImp. 70
37 Sequence Diagram for Creating and Dispatching Tasks 72
38 JiniTaskManger Sends Tasks to JiniBLASTServiceImp via Proxy. . 73
39 Class TaskDispatcher Begins a Thread to Create and Distribute Subtasks. 74
40 Class JiniBLASTServiceImp Manages a BLASTEnginge Thread Group and Limits the Number of Processes Running on BLAST Server. . . . 75
41 Sequence Diagram for Getting Result and Writing it to ResultSpace . 76
42 Class JiniBLASTServiceImp Adds the Result to the List and Notify JiniTaskManager. 77
43 Class TaskManager Gets Results from the BLAST Service and Writes Results to the ResultSpace. 78
44 Class JiniTaskManagerServiceImp Finds an Instance of the JiniClient and Notifies it that Result is Ready. 79

45 JiniBLASTClient Get Notify Message and Call method *takeResults()*. 80
46 ResultFrame Gets Results from ResultSpace for a Client. 81
47 Sequence Diagram of a Client Reading a Result from ResultSpace . . 82
48 *wroteErrorMsgToSpace()* in JiniNode. 83
49 JiniBLASTClient Creates an Instance of ErrorMessage and Writes it to the Error Space When an Exception Occurs. 84
50 FailureManager Thread Continually Checks for the Potential Failure of the System Node. 85

List of Tables

1 Comparison of High Performance BLAST Systems 10

Chapter 1

Introduction

1.1 Context of Thesis

Along with the explosive burgeoning of biological data. the application of high performance computing in biology is gaining more and more attention. The commonly used technology for high-throughput computing is a distributed system. A heavy computing task can be split into a set of smaller subtasks. Then these subtasks can be distributed to different servers to be executed concurrently. After subtasks are finished. their results are returned and reassembled into the final result for the original task. In this way. the total execution time decreases dramatically. In such a system. the key issue is how to coordinate the working of all server nodes. balance the work load among these nodes and handle the network exceptions. The ideal system should be flexible. robust and need minimal administration.

BLAST (Basic Local Alignment Search Tool) is one of the most common DNA (Deoxyribonucleic acid) and protein sequence similarity search tools. The purpose of BLAST is to predict the functionality and biological role of unknown DNA or protein sequences through comparison with those well-understood sequences in a database [1]. We assign similarity to sequences by the score of an alignment locally or globally. Due to the huge size of databases with millions of sequence segments. the throughput of the BLAST system is a bottleneck of the analysis process. The current solution is to split the BLAST task into several small tasks. Then these small tasks are performed concurrently to improve the performance. Generally we can select a job partition. a data partition or mixed methods. In job partition. the requested task is split into several small requests. In data partition, the database is split into several smaller parts and the task is executed against these data partitions concurrently [4, 13]. If we use both job partition and data partition. we can get even smaller subtasks and may get better performance. However. there is an overhead of partitioning. It is not always true that the smaller the task the better the performance. In this thesis, we design a high performance distributed BLAST cluster system using the data partition algorithm. Our solution is a simple system containing a cluster of BLAST servers, each performing a BLAST search against a partition of the database.

Jini is a new distributed computation infrastructure from Sun Microsystems, which was released in 1999. According to the Jini specification [9], it is an ideal tool for implementation of dynamic distributed computing systems. In such systems, there are several clients, several services provided by servers, as well as at least one lookup service. The lookup service is an object repository, from which clients can download the service proxy object. All services are registered on a lookup service. Clients can find lookup services by either a multicast or unicast protocol. The client can automatically find the current available service with some desired service properties. This way the whole system can dynamically self-configure without disturbing services. For instance, we can add more servers or remove some servers from the system without shutting down the whole system.

Based on Jini technology, Sun Microsystems developed JavaSpaces [10] which provides a cross-network shared memory. The network nodes can write and read objects from that shared memory. As Migliardi [12] shows, the JavaSpaces model is a simple unified mechanism for dynamic communication, coordination, and sharing of objects between Java technology-based network resources.

Currently there are several implementations of high performance BLAST programs, such as TurboBLAST [13], PowerCloud BLAST [18] and the high throughput distributed DNA sequence analysis system from National Center for Genome Resources (NCGR) [19]. None of them is implemented using Jini technology. In this thesis. we present the design of a new high performance BLAST server based on Jini technology. It not only provides high throughput. but also provides dynamic self-configuration properties [12]. Another advantage of Jini is that it is an open source technology based on Java. It can be used by anyone and is platform independent. From the development perspective. it is very easy to use. Sun Microsystems provides a Jini starter kit [10]. which contains all the essential classes and APIs for Jini development.

In a distributed computing environment. the main problem is how to coordinate the jobs on distributed nodes and maintain a reasonable load balance. Developers usually use workflow management to control the running of a distributed computing system. Workflow is the procedure to coordinate the work of all participants to perform a predefined business goal [7]. A workflow management system is a type of application managing the sequence of work activities and the invocation of appropriate human and/or IT resources in each activity step. In this thesis. we design a workflow management system based on Jini and JavaSpaces to coordinate the task dispatching and data flow of a distributed BLAST search pipeline.

1.2 Previous Work and Purpose of This Thesis

As mentioned above. we can benefit from implementing a BLAST server using Jini technology. Liusong Yang has evaluated the possibility of implementing a single standalone Jini based BLAST server in her master thesis [27]. In such a BLAST server. the BLAST algorithm is implemented using C++. The Jini service API enables it to join a Jini environment. The client can send the BLAST request directly to the server and the server returns the results directly back to the client. Each server performs the whole BLAST search task. The performance of such a system is not good. Another problem of such a system is it may waste a lot of computing resources.

Since even for one big task, only one server can execute the BLAST algorithm.

In order to improve the performance of BLAST search, we plan to design a high performance BLAST server using Jini and JavaSpaces technologies. This project is partitioned into three parts: BLAST server implementation using C++, the design of a workflow management system, and a design of a client GUI. At the time of this writing, the BLAST engine and client GUI are not completed. Thus, we cannot provide a working system for testing or evalution. This thesis will focuses on workflow management design. We will provide a detailed design for workflow related classes and API signatures for client GUI related classes.

Although this paper focuses on a BLAST cluster, from my personal opinion it is easy to adapt the system to other application servers. In other words, it can be applied to other high throughput computations by changing the task definition, server implementation and client interface implementations. The current design of this system is for our local use. However, it is easy to extend it globally by adding security control.

1.3 Contribution of The Thesis

The principal contribution is a system architecture for Jini-based distributed BLAST cluster server system. The multiple BLAST Servers allow this system to do high performance BLAST searches. The Jini and JNI technologies allow this system to have dynamic configuration and platform independency properties. The multiple Task Managers decentralize the system control and workflow management, and the cross network shared error space improves system tolerance.

This thesis also includes detailed design and some implementation for our system. The detail design contains module design, Java class diagrams and sequence diagrams. We specified the Java packages and classes used in our system in class diagrams. The sequence diagrams present the major system activities and runtime components interactions. The implementation is done in JBuilder 4.0 with JDK 1.3.1

and all classes are compiled without error. For some uncertain classes, we also provide the detailed API and function signatures. Although these detailed designs and implementation may be modified in the final version, they will help us implement a working system.

1.4 Thesis Organization

This thesis contains four chapters. Chapter 1 (current chapter) gives a brief description of the context of this thesis. Chapter 2 introduces the background knowledge of BLAST, Jini, JavaSpaces and workflow. Chapter 3 presents the system design including architecture design, detailed module design, Java class diagrams and dynamic sequence diagrams. Chapter 4 discusses the achievement of this thesis, the status of this project and possible future work.

Chapter 2

Background

2.1 BLAST - Basic Local Alignment Search Tool

BLAST is a sequence alignment algorithm. The main purpose of this algorithm is to compare the similarity of unknown sequences against a collection of well studied sequences in a database [1]. Such a comparison can help us predict the biological function and structure of unknown genes or proteins, and predict the evolution of species.

2.1.1 Local and Global Alignment

Sequence alignment means the procedure of comparing two or more sequences by looking for a series of individual characters or character patterns that are in the same order in the sequences. The quality of the alignment is evaluated using an alignment score. We can use global and local alignment to compare the sequences. Global alignment is an attempt to align the entire sequences, as many characters as possible. In order to get the maximal score of alignment, non-identical characters or gaps are placed to bring as many identical or similar characters as possible. In contrast to global alignment, local alignment searches for regions of local similarity and need not include the entire length of the sequences. Local alignment methods are very useful for scanning databases or when you do not know that the sequences are similar over their entire lengths (Fig 1) [11].

```
A. Global Alignment           B. Local Alignmnet

LGPSTKQFGKGSSSRIWDN           -------FGKG--------
|      ||||   |  |                   ||||
LNOIERSFGKGAIMRLGDA           -------FGKG--------
```

Figure 1: An Example of Global and Local Alignment.

2.1.2 Alignment Algorithms and Similarity Scores

How can we perform exact similarity comparisons between sequences? In artificial intelligence (AI) . the Dynamic Programming Algorithm (DPA) is a heuristic method to find the shortest paths in a search graph [15]. It is an ideal algorithm to solve optimization problems. It is also used in sequence alignment. In this method, the problem is represented as different states. The solution of the problem is the destination state. The algorithm is to find an optimal path from initial (original) state to the destination state [16]. This algorithm is based on recursion from each current state, selecting some/or one correct (optimal) next states to be added into the solution path. The most important step for this algorithm is how to select the next state (or node in graph) from the current state. The method is to calculate and assign a score to each child node of the current node, and then select the one with highest score in the set of nodes. The sequence alignment algorithm is based on this algorithm. This algorithm is trying to find a set of small length alignments and then extend these alignments, but only keeps those with higher alignment scores (optimal sets).

The Smith-Waterman algorithm is an example of such an algorithm. As shown in Fig 2, this algorithm constructs a two-dimensional table of partial alignment scores. The table has one dimension or axis for each sequence. Each cell in the table contains the score for the best partial alignment that terminates with the pair of sequence residues (one from each sequence) that correspond to that cell in the table. The best scoring partial alignment is extended to subsequent cells in the table only when it is the prior cell that results in the best scoring partial alignment for the subsequent cell. The final alignment is thus the best scoring alignment possible [17].

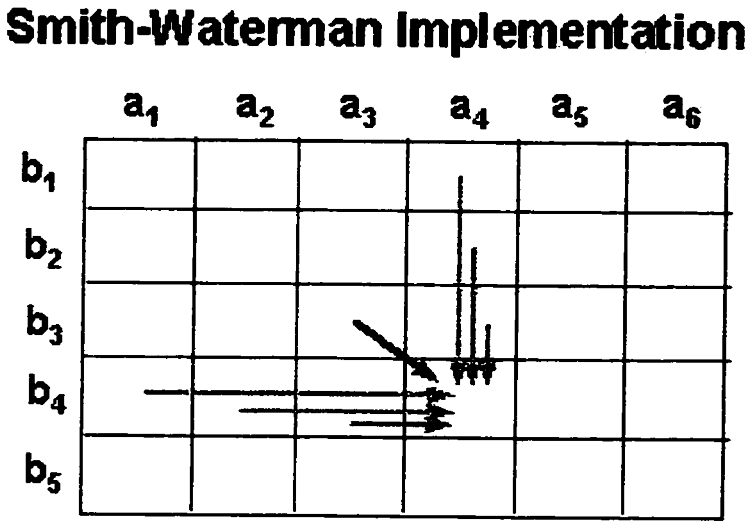

Figure 2: Two Dimensional Table of Smith-Waterman Algorithm.

However. this algorithm may take a long time to complete one comparison. In fact. there is a set of different sequence search algorithms. These algorithms use heuristic algorithms to improve performance and selectivity without losing too much sensitivity. BLAST uses a word-based heuristic to approximate a simplification of the Smith-Waterman algorithm. It is also known as the maximal segment pairs algorithm. Recent advances in maximal segment pairs statistics allow the use of several independent segment alignments to evaluate the significance probability. The BLAST algorithm is less sensitive than Smith-Waterman, but more selective in appropriate circumstances. Recent versions of BLAST can handle gaps and exhibit higher sensitivity [17. 14]. Usually, the BLAST search is more sensitive for protein sequences, since the nucleotide sequence has only four characters.

Detailed information for each algorithm and all available algorithms is beyond the scope of this thesis.

2.1.3 BLAST Search Database and Algorithms Options

The National Center for Biotechnology Information (NCBI) Genebank database contains a set of DNA and protein sequences. When doing a BLAST search, we usually select a database for the closest species first. Different databases also provide different biology evolution information. For more information about databases, please refer to Appendix A.

As mentioned above, when doing a BLAST search, we can select different algorithms to obtain different sensitivity and selectivity. Some algorithms have higher sensitivity on proteins over nucleotides. We also may need to convert a gene sequence into a protein sequence to compare results from different algorithms. The NCBI BLAST search web page provides a collection of search parameters. For example, blastn means standard nucleotide BLAST; blastp means standard protein-protein BLAST; blastx means nucleotide query against protein database BLAST. For more information about the BLAST search algorithm options, please see Appendix B.

2.1.4 High Performance BLAST

For short sequence alignment, a BLAST search is almost as simple as the examples mentioned in Section 2.1.1. However, in the real world, user usually need to compare hundreds or thousands of sequences against a database with millions of sequences. The computation task may take hours or weeks. Hence a high throughput system is very useful for such CPU-intensive tasks. Currently, we use data parallelism and job parallelism to split such big tasks into smaller ones, which are distributed to different nodes to be executed concurrently. The results produced by individual nodes are assembled into a final result. Such a system is usually referred to as a computing farm

Features	Our System	TurboBLAST	PowerCloud	NCGR System
High performance	Yes	Yes	Yes	Yes
Dynamic Configuration	Yes	Yes	N/A	No
Robust	Yes	Yes	N/A	N/A
Heterogonous Platform	Yes	Yes	No	No
Remote Access	Yes	N/A	N/A	Yes
Scalability	Yes	Yes	N/A	N/A
Technology	Jini. JavaSpaces	Java RMI	N/A	CORBA

Table 1: Comparison of High Performance BLAST Systems

[4]. Currently there are several implementations of high performance BLAST programs, such as TurboBLAST [13], PowerCloud BLAST [18] and the high throughput distributed DNA sequence analysis system from National Center for Genome Resources (NCGR) [19]. These systems are based on CORBA or other distributed computing infrastructures. Table 1 displays a comparison of our system with these other systems.

2.2 Jini - A Distributed Computing Infrastructure on Java

2.2.1 What is Jini

Jini (pronounced Jee-nee, is not an acronym and does not stand for anything) means "spontaneous networking". It is a new distributed computation infrastructure from Sun Microsystems. In a network based on this technology, users are able to plug any Jini-enabled devices such as printers, storage devices, speakers, etc., into a network. All other devices and computers in the same system can be notified that the new device has been added and is available. When the clients need to use some resources, it can look up related "services" from the network and download the proxy of the service to complete the tasks. This way, there is no longer any need for the special device support software known as a device driver in a user's local operating system [8. 9].

2.2.2 How Jini Works

Basically. the Jini system consists of three major components: clients, services and lookup services. These three components cooperate with each other using the following protocols [2], as shown in Fig 3:

- **Services discover and join:** each Jini enabled service can discover lookup services using a unicast or multicast protocol and then register itself into the found lookup services.
- **Clients lookup and download:** the Jini client finds a service via a lookup service and download the proxy for the required service to its local Java virtual machine (JVM) to perform desired tasks. During computation, the proxy may communicate with the service provider directly if it is necessary.
- **Entry publish and service attributes match:** each service has one or more entries to list its attributes, including the name, the service it can provide and the quality of the service (e.g. printing quality of the printer). These entries are published on the network when the service is registered. The client can find the required service by matching these fields.

In other words, in Jini-enabled systems, the lookup service serves as a proxy repository. Each service has to register itself, and upload its proxy to at least one lookup service. Clients can search for service proxies by matching entries. Then clients can download the proxy and use it as a local object. Since a lookup service can broadcast its existence, the services and clients can communicate to each other with minimal knowledge about each other.

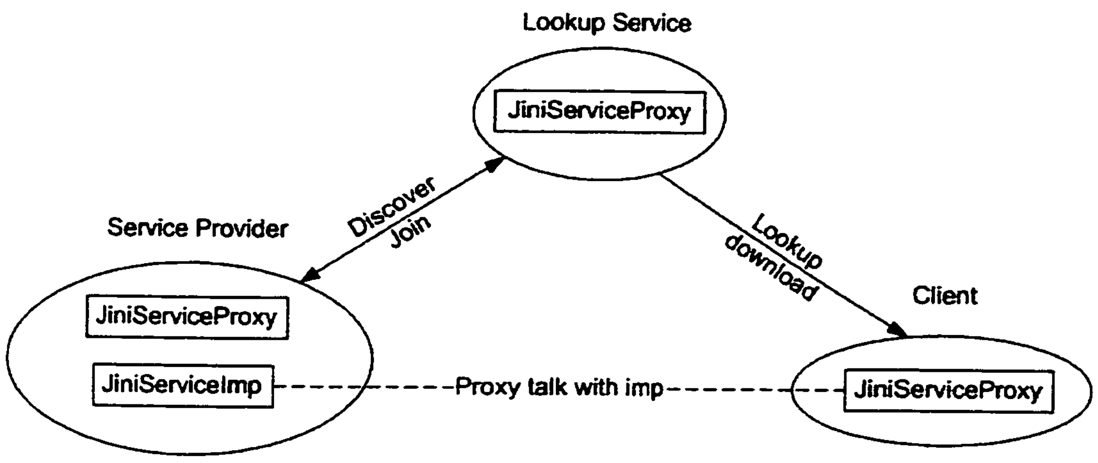

Figure 3: Jini Protocols.

2.2.3 Architecture of Jini Service

According to the implementation method, the Jini service proxy can fall into two categories: fat and thin proxies.

A fat proxy usually does most of the jobs to complete a given task. In extreme cases, it contains all of the code to perform the service. It is downloaded to the client and performs all the computing tasks. As shown in Fig 4, the service includes an interface and an implementation. The client uses the interface as a template to search for an implementation proxy. At runtime, the implementation of a proxy resides in the client JVM. It executes all of the computing tasks on the client's side. For some heavy computing tasks and those tasks which need to communicate with databases, the thin proxy is more reasonable. A thin proxy does no more than provide an interface to enable the client to communicate with the service providers. It has to exchange data and/or code with the service provider to complete its task. Although a thin proxy can be implemented using other protocols, usually it is implemented using Java RMI (Remote Method Invocation) technology [2]. The classes and JVM objects are shown in Fig 5. At runtime, the service implementation stub will be downloaded into the client JVM and will communicate with the server. We also can implement Jini services using a socket client and server model. The client downloads a TCP/IP socket proxy to talk with the server.

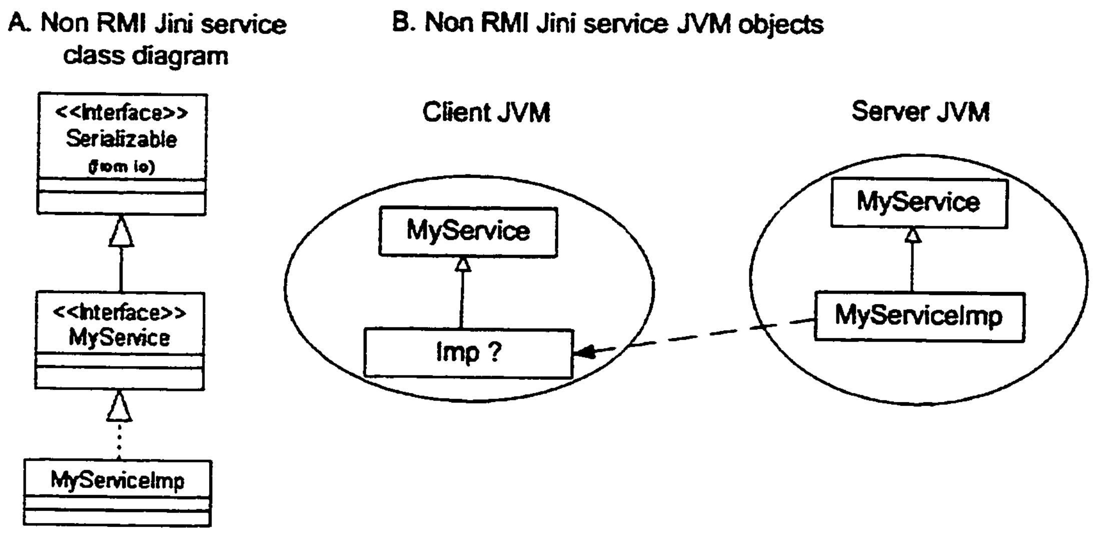
A. Non RMI Jini service class diagram
B. Non RMI Jini service JVM objects
<<Interface>> Serializable (from io)
<<Interface>> MyService
MyServiceImp
Client JVM
Server JVM
MyService
Imp ?
MyService
MyServiceImp

Figure 4: Fat proxy Class Diagram and JVM Objects

2.2.4 JavaSpaces

Based on Jini technology, Sun Microsystems developed JavaSpaces and released the first implementation of JavaSpaces with the Jini starter kit. JavaSpaces is a Jini service that provides an object database service. It can be used as a shared memory in a network. A typical JavaSpaces implementation provides the following interfaces:

- Write: write an object into JavaSpaces.
- Read: read an object from JavaSpaces.
- Take: read and remove an object from JavaSpaces.

Those objects written into JavaSpaces have to implement the `Entry` interface. In this way, a client can use Jini `Entry` field matching to find specific objects.

2.2.5 Jini and Distributed Computing

Jini and JavaSpaces provide a variety of features, which are necessary for implementing distributed systems. These features include: discovering resources, dynamic federations, distributed leasing, distributed event management, and over network shared objects memory, allowing construction of a self-healing system that has no single point-of-failure [20]. In a distributed system, different nodes need to exchange data and messages from each other to establish communication. Some technologies can allow remote code execution. However, the client can only create a reference of remote objects. In contrast, Jini not only allows data and messages to travel over the network, but also allows code to travel to the client's Java virtual machine (JVM). Moreover, JavaSpaces provides a cross network shared memory as an object repository. All participants can write and read objects from JavaSpaces. These shared objects include both data and program code.

In general, Jini technology has the following features and properties that make it easy to implement more robust distributed systems [9]:

- **Scalability:** Jini provides an ideal environment to create flexible networked systems scalable from the small device to the enterprise.
- **Open source:** The Java and Jini community are free and open source, making it easy to set up a creative collaboration.
- **Code mobility:** Not only data, but also programming code can travel over a network in Java objects.
- **Robustness:** Since Jini services use the leasing method to register service proxy, Jini-enabled network can do self-configuration and improve failure tolerance.
- **Integration and compatibility:** the Java language properties and the Java native interface (JNI) allow fast, easy incorporation of legacy, current, and future network components.

2.3 Workflow and Workflow Management System

2.3.1 Workflow

Workflow refers to the procedure of coordinating work among all participants to finish a predefined business goal. The participants can be people. machines and computation devices. In such a procedure. the information and tasks are passed between participants serially or concurrently in a predefined order. Now the workflow is normally organized within the context of an information technology system to provide computerized support for procedural automation [7].

2.3.2 Workflow Management System

A workflow management system is a type of application for managing the sequence of work activities and the invocation of appropriate human and/or IT resources in each activity step. The process definition and task list management is the main functionality of such a system. The task list is a job queue. in which the tasks are performed in a predefined order concurrently or serially. A typical workflow management system has the following five interfaces [7]:

- **Process definition:** specifications for process definition data and its interchange with the Workflow Execution environment.
- **Workflow interoperability:** interfaces to support interoperability between different workflow systems.
- **Invoked applications:** interfaces to support interaction with a variety of IT applications.
- **Client applications:** interfaces to support interaction with human users.
- **Administration and monitoring:** interfaces to provide system monitoring and metric functions to facilitate the management of composite workflow application environments.

2.3.3 Workflow Management System Based on Jini

As mentioned in Section 2.2.5. Jini can make distributed computing implementation easier. Jini and JavaSpaces can also make a workflow management system much easier to implement and much more robust. As Schaefer has shown [22], it is easy to treat all workflow worker participants as service providers and all tasks, processes and activities as objects traveling over the network. We also can write all tasks into shared JavaSpaces. The workflow application components can read or take their interested tasks to perform their own computation job.

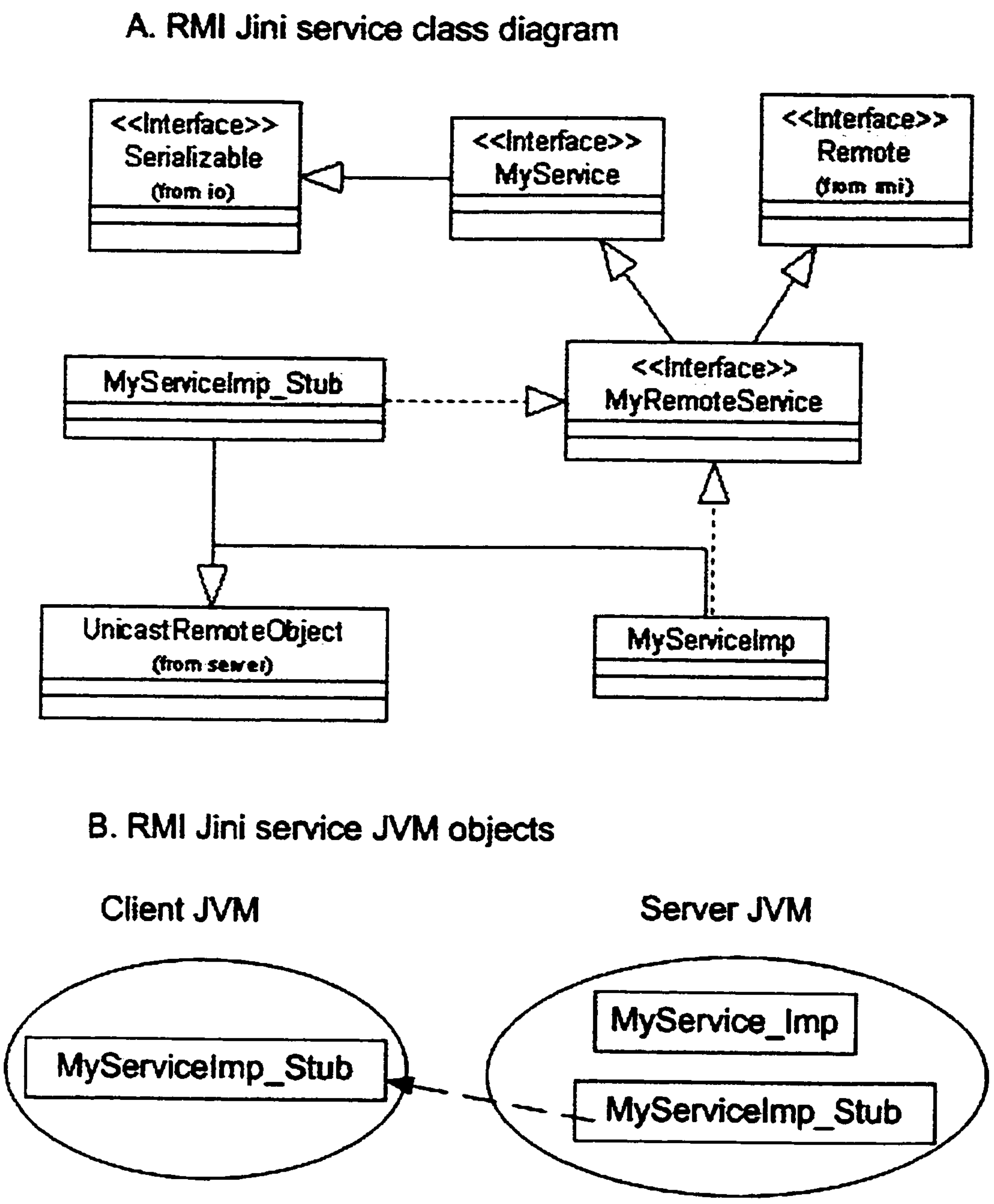
A. RMI Jini service class diagram
<<Interface>> Serializable (from io)
<<Interface>> MyService
<<Interface>> Remote (from rmi)
MyServiceImp_Stub
<<Interface>> MyRemoteService
UnicastRemoteObject (from server)
MyServiceImp
B. RMI Jini service JVM objects
Client JVM
Server JVM
MyServiceImp_Stub
MyService_Imp
MyServiceImp_Stub

Figure 5: RMI Proxy Classes and JVM Objects.

Chapter 3

Design

3.1 Problem and Solution

The purpose of this thesis is to design a high performance BLAST search system. Compared with the existing systems. it should be more robust. more flexible. and it should be able to dynamically extend or contract with minimal administration effort.

We also apply a distributed computing architecture for our system design. Our system consists of a cluster of BLAST Servers. These servers can have diverse hardware and/or OS platforms. Each server handles BLAST searches against one small database partition. In other words. in the current version of our system. we use a data parallelism algorithm to split BLAST tasks (See Section 2.1.4). In order to improve system stability and have true dynamic self-configuring properties. we design a Jini-based workflow management system to handle network exceptions, coordinate the distribution of tasks and maintain load balance among BLAST Servers.

We can get some benefits from Jini technology. First, the users can have a transparent view of the system. They do not need to know the IP address or domain of a BLAST Server before submitting a BLAST search request. Second. we can add or remove any BLAST Server to or from the system without disturbing the service. Third, it is easy to mix heterogeneous BLAST Servers in different platforms into the system. Each server may have a different implementation of BLAST algorithms and/or databases. This can provide an easy way to partition a huge database set into several small parts.

In our system. the workflow management system is a software component to manage data and task flow between BLAST clients and BLAST Servers. The major functionality of the system is providing BLAST search pipeline management and failure handling.

3.2 System Specification

It is simple for the end user to use this system. The user provides some sequences and related parameters. The system executes a BLAST search and returns the results to the user. Another type of user is the system administrator who monitors the system status and controls some services. However. from an internal view. this is a distributed BLAST cluster system controlled by a workflow management application. The Jini-based BLAST cluster system contains the following components (Fig 7 and Fig 8):

- BLAST Clients: receive user's input: generate BLAST Request: send requests to service and retrieve results.
- Task Manager: workflow management engine to coordinate the task and data flow between BLAST Client and BLAST Server. Backup all data exchanged among system nodes during BLAST search execution.
- Task Manager Database: storage to back up user's input, results and intermediate data.
- BLAST Server: execute BLAST algorithms and produce the BLAST search results.
- Lookup Services: Jini service proxy repository providing lookup, discovery, registration and downloading service.
- Administrator: provide GUI to monitor system runtime architecture and status.

- Result Space: shared network memory providing storage for result objects exchanged between nodes.
- Error Space: shared network memory providing storage for error message objects for failure handling. May be combined with the result space.

3.2.1 BLAST Client

The major function of a BLAST Client is providing a user interface for the end user. From a BLAST Client. the user can send a request and view query results. In our system. there is more than one BLAST Client and each has a unique name. The Client supports following functionalities:

- **Send a request:** user can input the query sequences and BLAST parameters then issue a BLAST search request.
- **Display results:** user can view the BLAST results in graphic format.
- **Administration function:** report client status to administrator.

Send a request

In addition to allowing the user to select all BLAST search parameters. as those on the NCBI web blast interface [1]. this system also allows the user to view the server list and select appropriate servers to perform the BLAST search. The user can also submit a batch of sequences each time. The user follows these steps to submit a BLAST request:

- **Input query sequences:** The user can input more than one query sequence directly in a GUI text box or upload a pre-edited text file. All sequence data should be in FASTA [1] format.
- **Select query databases:** The user can select desired DNA or protein databases to run the query against. More than one database can be selected at each time. See Appendix A for available databases.

- **Select BLAST algorithm:** The user can select one of the available algorithms. See Appendix B for available algorithms.

- **Select alignment view option:** The user can select a favorite display format for results. Current available formats include: Pair wise, query-anchored with identities, query-anchored without identities, flat query-anchored with identities, flat query-anchored without identities and hit table.

- **Browse and select desired** BLAST Server**:** The user can use a GUI-based BLAST service browser to find desired services (optional). In the browser, each BLAST service lists its database, available algorithms, current status (idle or busy) and computation resources (such as number of processors, speed of processor and total RAM in megabytes or gigabytes).

- **Submit request:** After the user has collected all the necessary information, he can click a button to submit a request.

- **Request validation:** Once the user has confirmed the request, the BLAST Client program validates the request based on the user's input. The validation is based on the following criteria:

  - **Query sequences quality:** percentage of unknown nucleotide pair or protein amino acid residues, such as user input error due to typing mistakes.

  - **Request parameters:** some input data cannot be applied to specific algorithms or databases.

  All requests that do not pass the validation are blocked, and the user is prompted to correct the input data.

Display Results

The BLAST Client GUI presents the results graphically to the user. This functionality includes the following scenarios:

- **Display request progress status:** After the user submits the request, the GUI displays request progress information.
- **Display instant results:** When a request is being processed, the BLAST Client is notified when partial results or final results are available. The user can select to view these results now or later.
- **Retrieve backup results:** If a request is done, the user can retrieve the backup results from the database at any time.

Administration function

In order to allow an administrator to monitor the BLAST Client, each client must include the following functionalities.

- Report the current status to the administrator.
- Allow the administrator to change the status.

3.2.2 Task Manager

The Task Manager is the main workflow component in our system. Its main functionality is coordinating the communication, task distribution and failure handling. Another important functionality is backing up all the data used in this system including requests, tasks and final results. It supports the following functionalities:

- **Receiving requests:** receive request from BLAST Client.
- **Searching for old results:** in order to save system resources, Task Manager searches backup results for request with the same contents. The search is performed on request sequences and BLAST parameters.
- **Creating tasks:** Task Manager creates tasks from a request.
- **Split task:** those tasks with multiple database partitions are split into subtasks using data parallelism.

- **Dispatching tasks:** dispatching subtasks to an appropriate BLAST Server by matching the request database and preferred server.
- **Receiving results:** receive results from BLAST Server.
- **Preparing final results:** assemble results to final results if it is necessary.
- **Notify** BLAST Client: when a final result is ready. Task Manager notifies the client.
- **Failure management:** continually check for potential system node failure or network exception.
- **Administration function:** report status to administrator.

Receiving requests

The Task Manager receives requests from BLAST Clients. Two requests may have the same contents. but if they come from different BLAST Clients or from the same BLAST Client with different IDs (serial number). we treat them as different requests since the results for these requests are sent back to different clients or the sequence database is updated.

Searching for old results

In order to reduce the work load for the system. the Task Manager searches among previous results with the same request contents and parameters in the Task Manager Database. If such a result is found. and all related BLAST Databases have not been updated. the result is retrieved and sent to the client. Otherwise. the Task Manager continues to process the BLAST search.

Creating tasks

The Task Manager generates tasks based on request contents and parameters. Each task has the following fields:

- **Algorithm:** selection of BLAST algorithm. For more information please refer to Appendix B.
- **Databases:** selection of databases against which to run the query.
- **Sequences:** the user's query sequences.
- **Alignment view option:** the final user preferred display format of results.
- **Preferred BLAST Server:** the user preferred BLAST Server name or ID to execute the request. This is an optional parameter.

After each task is created, it is added into the task queue.

Split task

Based on data parallelism, if there is more than one BLAST database listed in the task, the task is split into subtasks. Each subtask only contains one query database. Depending on the database partition and the subtask database entry, these subtasks may or may not be sent to different BLAST Servers.

Dispatching tasks

At any time, if the task list is not empty, the Task Manager picks a subtask and sends it to one of the appropriate available BLAST Servers. The task dispatching is based on the following criteria:

- **BLAST database matching:** The request query database is matched with one of the BLAST Server databases.
- **First-in, first-out:** In case there is more than one task matched with the same BLAST service, the first-in and first-out rule is used.
- **Preselected server:** In case more than one BLAST Server is matched with the same task, the server whose name is matched with task's preferred services name

is selected. If no BLAST Server is preferred by the task, the server with more computational resources is selected. Otherwise, an arbitrary server is selected.

Receiving results

After a BLAST Result is created on the BLAST Server, the Task Manager is notified and attempts to get the results from the BLAST Server.

Preparing final results

After the Task Manager has received the results, it prepares the result into the final format and sends it back to the client. This procedure includes the following steps:

- **Assemble results:** all results based on subtasks are assembled into one result for the original task.
- **Convert results format:** convert the result data into the user-preferred view format. For available formats please refer to Section 3.2.1.
- **Write results to the shared result space:** Task Manager writes the final results into a result space and notifies the BLAST Client that the results are available. See beginning of this section.
- **Store results into the backup database:** One copy of the final results are stored in the backup database to let users retrieve their results later or from a different BLAST Client machine.

Notify BLAST Client

While each request is being processed, the Task Manager notifies the BLAST Client for several events:

- **Request received event:** After the Task Manager creates each new task from a request, it generates a request received event to notify the BLAST Client. This event contains a request ID. After the BLAST search is done, the user can use this ID to retrieve the results from the database on any machine where the BLAST Client program is running.

- BLAST Server **failure event:** In case one or more BLAST Servers fail, the Task Manager notifies the related BLAST Clients and resends the failed Task to an alternative BLAST Server.
- **Results ready event:** When the final result is ready, the Task Manager notifies the related BLAST Client where the request came from. If the BLAST Client is still available, the user can chose to view the results now or later.
- **BLAST progress event:** The BLAST Server sends back progress information about each individual task. The Task Manager notifies the related client about these events.

Failure management

Since this system is a distributed system, nodes and network connections may fail at any time and any place. Thus we use dynamic configuration properties of Jini to improve the stability of our system. Generally, we use the following methods to handle the failure of one or more nodes:

- **Multiple** BLAST Servers**:** more than one BLAST Server provides the same services with the same database partition. This can ensure that no database partition server totally fails. We also can add and remove servers without stopping the BLAST search service.
- **Multiple** Task Managers**:** having more than one Task Manager helps to decentralize the workflow management.
- **Backup** Task Manager database**:** provide storage of intermediate data, including the original request, partial and final results.

- **Shared error space:** one instance of JavaSpaces functions as a shared memory for all error messages. This decentralizes the failure handling. Each Task Manager can retrieve any error message and execute an exception handler.

Administration functions

The Task Manager reports its status to the administrator and allows the Administrator to change its status. In any situation, changing status must not result in lost data.

3.2.3 BLAST Server

The BLAST Server is a Java based program running on a workstation or a Beowulf cluster. It provides the software interfaces to invoke a native program, written in C++, to execute the BLAST algorithm and return the result. It also provides the Jini BLAST Service to let the Task Manager submit a BLAST search remotely. A BLAST Server provides the following functionalities:

- Register the Jini BLAST Service proxy object to all available lookup services.
- Publish the service properties to the system.
- Receive Tasks or Subtasks from the Task Manager.
- Prepare for the running of BLAST, for instance by removing vector contamination.
- Invoke the BLAST search program to perform the task.
- Return results to the Task Manager.
- Notify the Task Manager for any error or exception.
- Report server status to the administrator.
- Allow the administrator to enable and disable the service without losing data.

3.2.4 Lookup Service

To make things simple, our current version of the system uses the general Lookup service coming with the Jini Starter Kit [10]. It provides a repository for proxies of all Jini services used in this system.

3.2.5 Administrator

The administrator is a software component that allows the user to interact with this Jini based system. It provides a GUI for viewing the runtime system architecture. It is optional for the BLAST search. That means without any administrator instance running. the system can still perform a BLAST search. It has the following functionalities:

- **System monitor:** the user can view the system runtime architecture via a GUI module.
- **BLAST search history and performance analysis:** the user can query backed up results and execute parameters for each request.
- **Changing the node's status:** user can use administrator GUI to change the Jini service status.

System monitor

The system monitor provides a GUI to allow the user to view a picture of the runtime system architecture. The user can browse the currently available BLAST Servers, Task Managers and clients. It displays the following information and refreshes according to an adjustable time interval.

- Current status: either enabled or disabled.
- Current running tasks for a BLAST Server and a Task Manager.
- Current pending tasks for a Task Manager.

BLAST search history and performance analysis

The user can use the GUI to query the backup results to review the system's running history and to view performance statistics, such as the number of requests performed by the system and by each BLAST Server. These values can help us to find optimal work balance distribution. For each request, the system reports the following parameters:

- The request sequence number and length.
- The database partition size: bytes, length and number of sequences.
- The execution time and waiting time.

Changing the node's status

The user can block a client or disable a service at an appropriate time. These types of status changes should not result in lost data. Any unfinished tasks are handled by currently available services.

3.2.6 Results and Error Space

An instance of JavaSpaces from Jini starter kit serves as a cross-network shared memory to store results and error message objects. The Task Managers write results into this space, and clients read results from this space. All nodes can write error messages into this space, and any Task Manger can retrieve an error message to perform exception handling.

3.3 System Architecture

3.3.1 System Components and Functionalities

The system is a distributed computing system based on the Java network computing infrastructure Jini and JavaSpaces [8, 9]. From the system specification, Section 3.2, it is not difficult to list the components and their functionalities.

- Jini BLAST Client: a Jini client to find Jini Task Manager Services, receive user input and display the result. It is also a Jini service to provide client administration functions (Section 3.3.2).
- Lookup Services: a Jini service repository.
- Jini Task Manger: a Jini server that provides the Jini Task Manager Service to receive requests from clients; it is the main workflow management component to coordinate the work of system components; it is also a Jini client to find Jini Blast Services.
- Jini BLAST Server: a Jini server that provides the Jini BLAST Service; communicates with the native BLAST search engine to produce a BLAST Result; provides administrative information and allows the administrator to change the status. Each Jini BLAST Server can run on diverse platforms with different implementations of the BLAST search. From a hardware point of view, a Jini BLAST Server can be a standalone server or a Linux cluster.
- Administrator: a Jini client to monitor the system runtime architecture and to control the system. Not mandatory for BLAST searches.
- Task Manager Database: storage to back up data and log event information.
- Results Space and Error Space: cross-node shared memory to store results and error message objects.

For software functionality specification please refer to Section 3.2.

3.3.2 Administrator and Jini Adminservices

In order to provide a runtime system overview, each node in this system includes one Jini service. The purpose of this service is to provide status and activity information for each node. The Administrator is a Jini client that can find and download these services to create a graphic picture of the runtime system [3] (See Fig 6) and to communicate with Jini service providers. On some nodes, such as the BLAST Server and Task Manager, we use the one Jini service to provide both administration and system services. For instance, the Jini BLAST Service also provides the administration service function for the BLAST Server.

3.3.3 Communication between the Components

Since our system is a Jini-based dynamic distributed computing system, a large amount of data, messages and code is exchanged among system components. The interaction among the components in this system is changed dynamically at runtime. Generally, all communications are classified into two categories: Lookup services communication and non-Lookup services communication.

Lookup service communication

Communication between Lookup services and other nodes/components includes the Jini lookup services discovery, registration of service proxies, service searching and download. In our system, there are several Jini clients and Jini services. Before clients and services can begin to communicate with each other to complete tasks, they must talk to the Lookup Services first. In general, the following types of communication occur between lookup services and each individual node (Fig 7):

- BLAST Client: finding and downloading Jini Task Manager Services and Result Space from Lookup Service; registering BLAST Client administration service.
- Task Manager: finding and downloading Jini BLAST Services, Result and Error Space; registering Jini Task Manager Service.
- BLAST Server: registering Jini BLAST Service.

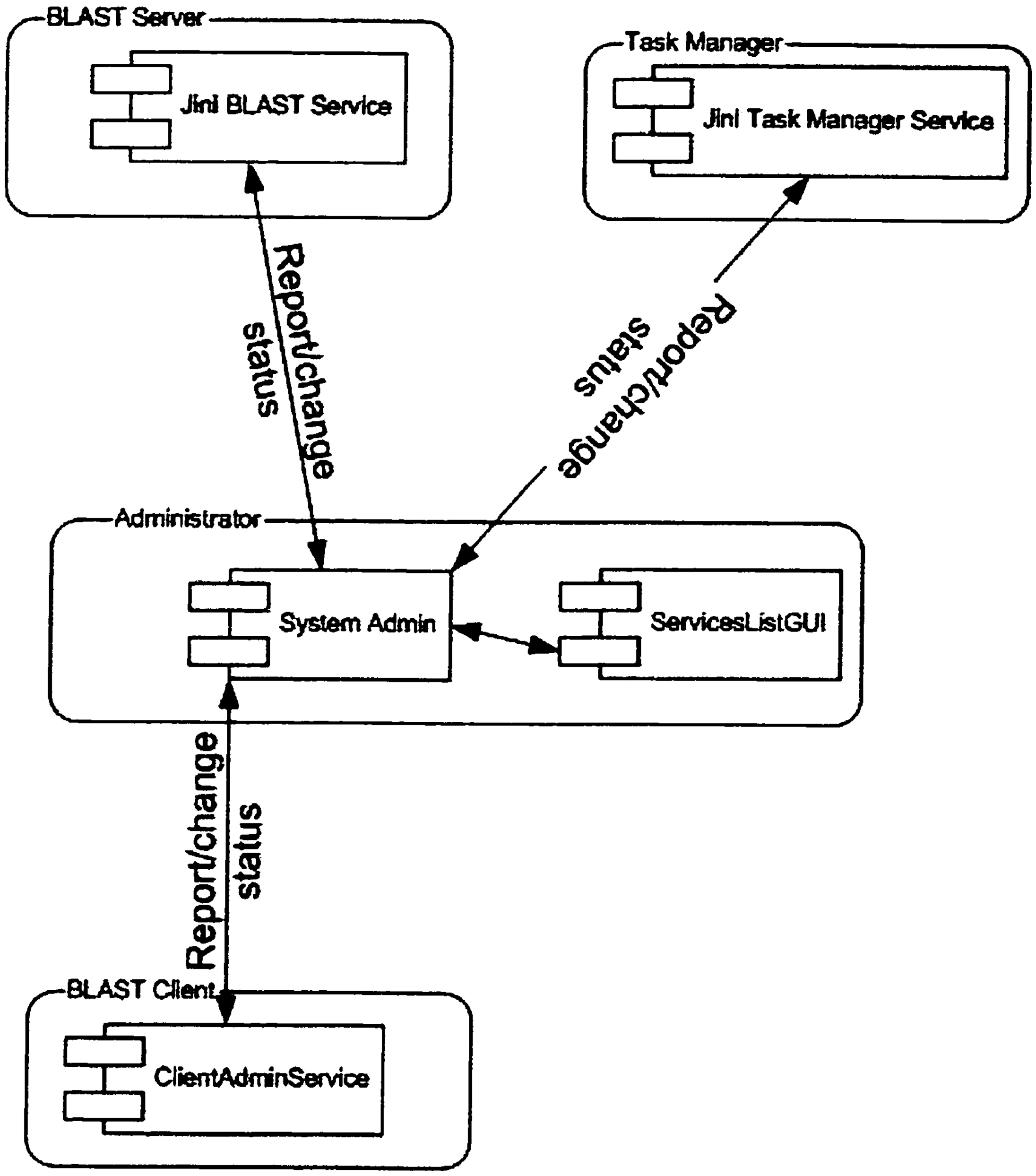

Figure 6: Architecture of Administrator and Admin Services.

- Administrator: finding and downloading Jini Administration Services for each node.
- Result and Error Space: registering space.

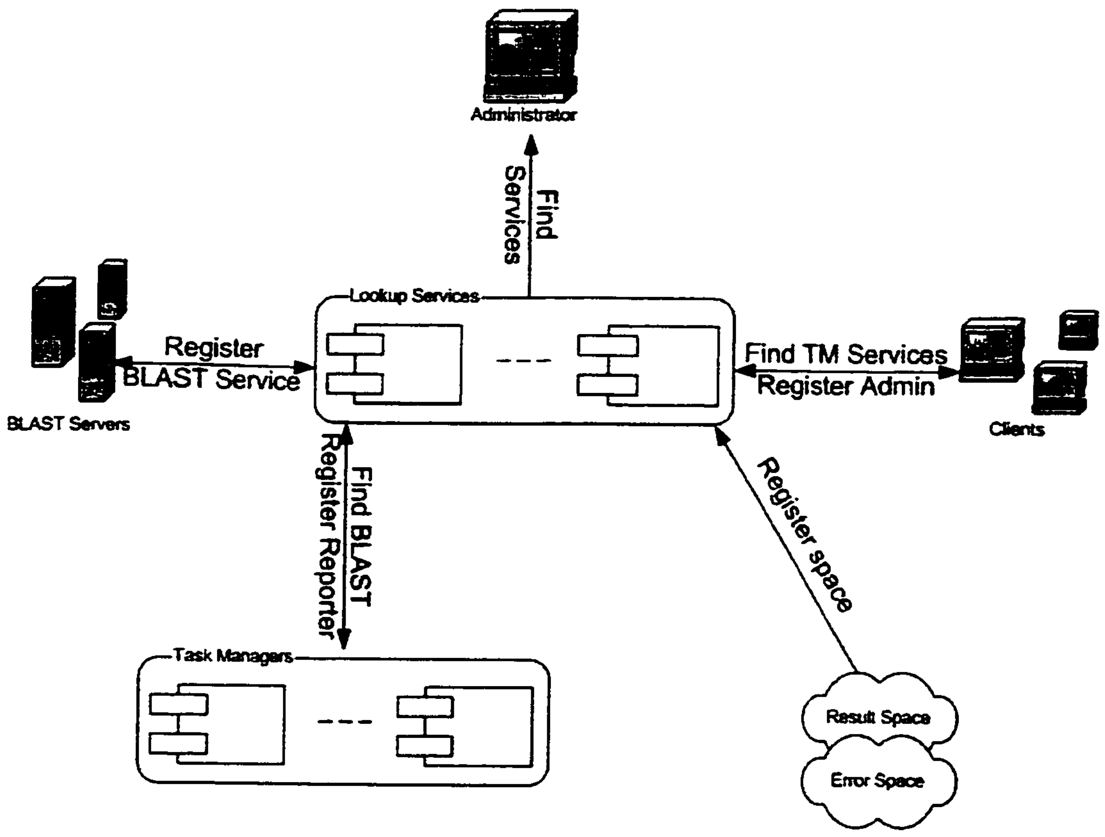

Figure 7: System Architecture and Interaction with Jini Lookup Service.

Non-Lookup service communication

This type of communication consists of the exchange of data or objects between nodes in order to perform BLAST searches, task dispatching and other system coordination. Generally the following data/object exchanges (Fig 8) occur:

- BLAST Client sends requests to Task Manager.

- Task Manager sends tasks to BLAST Server.
- BLAST Server sends back results to Task Manager.
- Task Manager writes requests. tasks and results to Task Manager Database.
- Task Manager writes results into Results Space.
- Task Manager notifies client for events.
- BLAST Client retrieves old results from Task Manager Database.
- BLAST Client reads the results from Result Space.
- BLAST Server notifies Task Manager that certain events have happened.
- Administrator retrieves service administration information from each node.

3.4 Module Design

In this section. we discuss the module design for each component as shown in Fig 9. The components and their modules are:

- BLAST Client:
  - User Input GUI.
  - Result GUI.
  - Service Browser.
  - Jini BLAST Client.
  - Jini Client Admin Service.
- Task Manager:
  - Jini Task Manager.
  - Task Dispatcher.

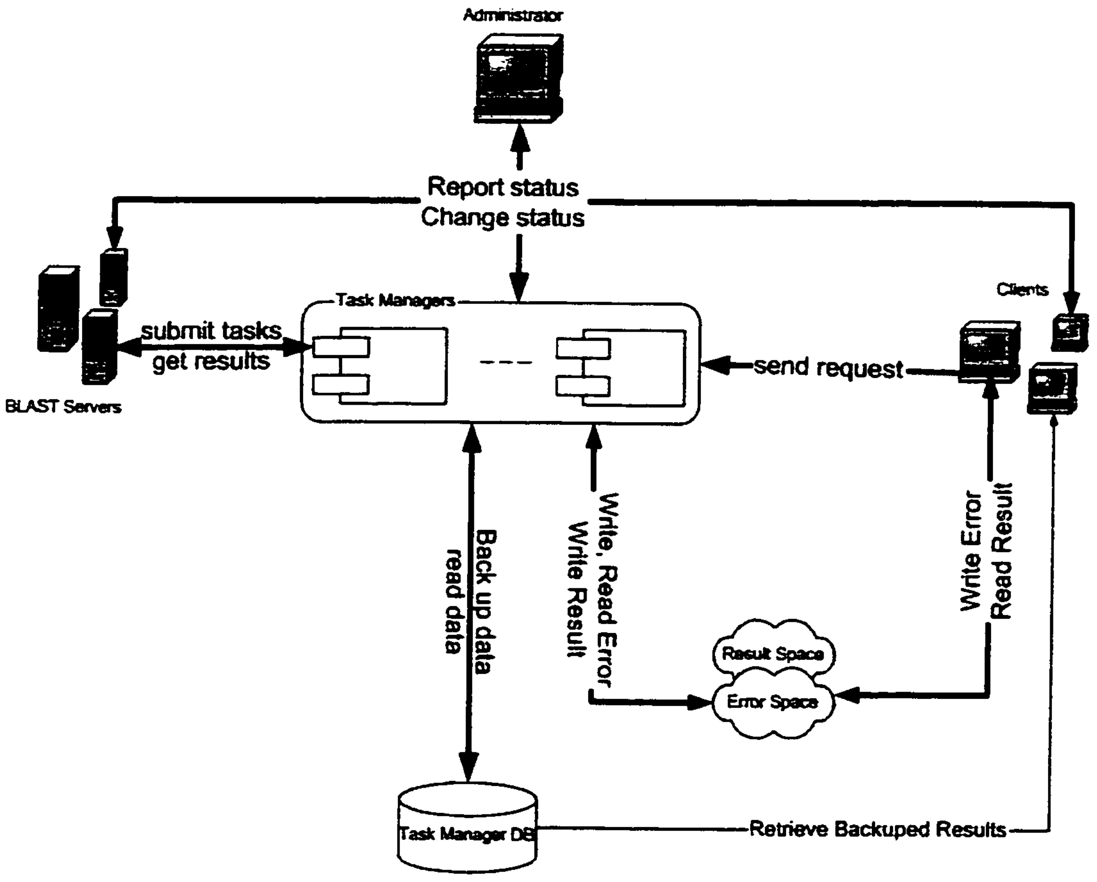

Figure 8: System Communication without Lookup Services.

  - Results Assembler.
  - Database Access.
  - Failure Manager.
  - Jini Task Manager Service.
- BLAST Server:
  - Jini BLAST Service.
  - BLAST Search Engine.
  - Jini BLAST Server.
- Administrator:
  - Jini Administrator.
  - System Browser.
  - Activity History Browser.

3.4.1 BLAST Client

In order to perform its functionality, a BLAST Client has the following components (panel A in Fig 9):

- **User Input GUI:** get user input.
- **Result GUI:** display BLAST results.
- **Service Browser:** display a list of available Jini BLAST Services and their properties.
- **Jini BLAST Client:** This is the main client program. It finds lookup services, downloads Jini Task Manager Services and sends requests to Jini Task Manager Services. It is also an object holder for the BLAST Server.

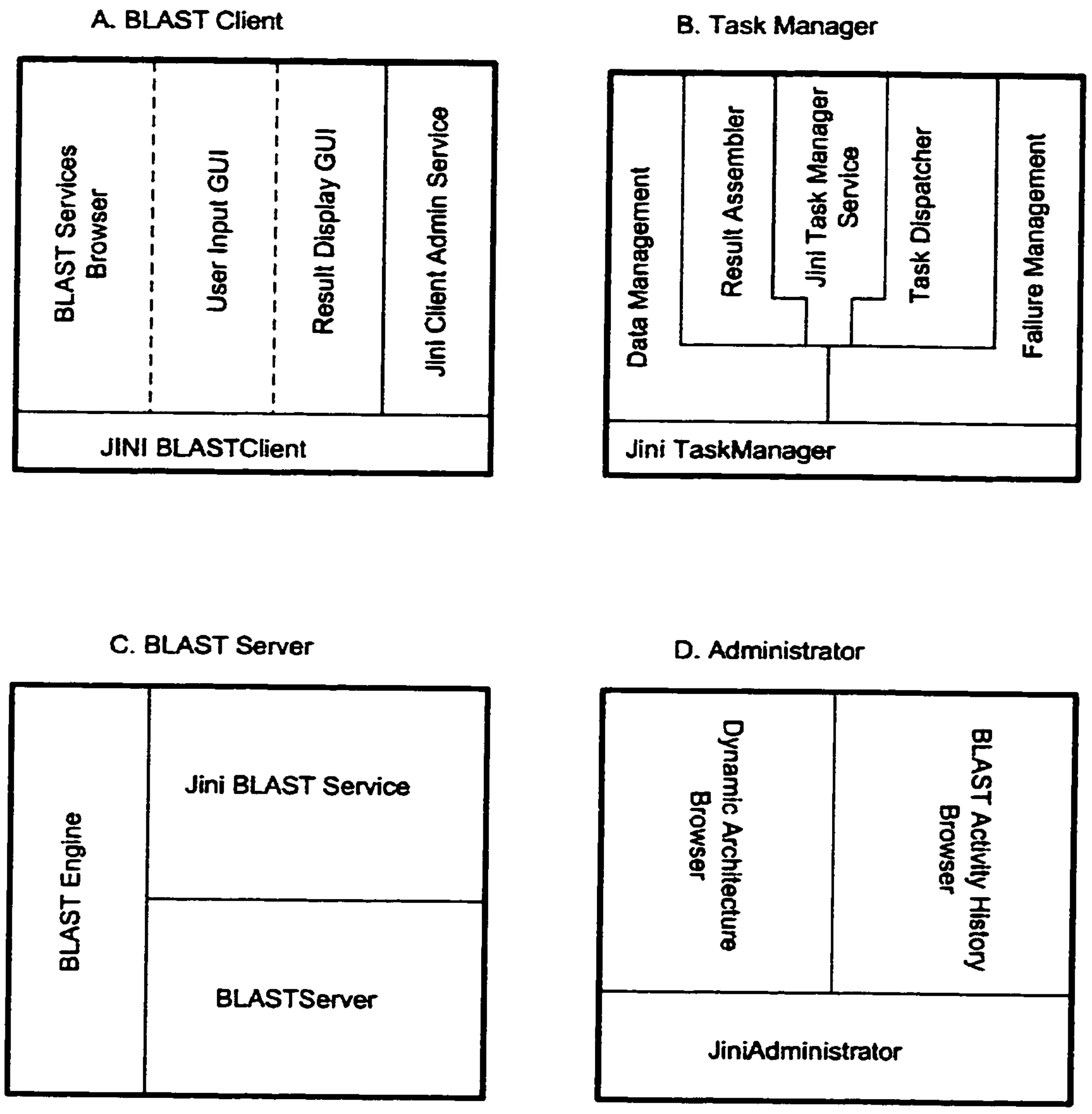

Figure 9: Module Design.

- **Jini Client Admin Service:** sends client information to the administrators and receives commands from the administrator

3.4.2 Task Manager

This is the main workflow management application of the system. In general, the Task Manager contains the following components (panel B in Fig 9):

- **Jini Task Manager:** This is the main program and object holder. It finds and downloads the Jini BLAST Services via the lookup services; it sends tasks to the Jini BLAST Service; gets results from the Jini BLAST Service; writes results to the Result Space and notifies the client that the results are ready.
- **Task Dispatcher:** generate tasks and subtasks; maintain a task queue and dispatch subtasks to the Jini BLAST Service.
- **Results Assembler:** prepare and merge the BLAST results.
- **Database Access:** provide basic database access and data manipulation functions.
- **Failure Manager:** continuously retrieves error message objects from the Error Space and executes appropriate error handling functions.
- **Jini Task Manager Service:** a Jini service is downloaded to the client and receives requests from the client. It also sends admin information to the Administrator.

3.4.3 BLAST Server

The BLAST program runs on this node. The Jini BLAST Service enables other nodes in the system to see and communicate with this program. It contains the following modules (panel C in Fig 9):

- **Jini BLAST Service:** a proxy registered in the Lookup Services; publishes Jini BLAST Service properties; receives BLAST Tasks; notifies Jini Task Manager regarding results.
- **BLAST Search Engine:** a Java Native Interface (JNI) based program that calls a legacy program to execute the BLAST search algorithm and write results back to the Jini BLAST Server results list.
- **Jini BLAST Server**: This is the main program and object holder for the server. It registers the Jini BLAST Service in the lookup services.

3.4.4 Administrator

This is the system administration tool software. The user can use it to view the system runtime architecture. That means it can display how many services and clients are currently running in the system. The user can view detailed properties for each service/client and the runtime status for each service. It contains the following components (panel D in Fig 9):

- **Jini Administrator:** This is the main program and object holder. It is a Jini client that can find and download all Jini BLAST Services. Jini Task Manager Services and Jini Client Administration Services; send commands back to the Jini services; coordinate the communication between components inside the administrator; process events and messages from other nodes in the system.
- **System Browser:** a GUI that displays the system runtime architecture and the status of each node.
- **Activity History Browser:** a GUI that retrieves system activities from the database based on the query criteria.

3.4.5 Task Manager Database

The Task Manager Database can be any relational or object oriented database management system. In the current version, we use a MySQL relational database.

3.4.6 Results and Error Space

The Results and Error Space is an instance of the JavaSpaces implementation from the Jini starter kit. No specific implementation is required. It provides a cross-node shared memory to store results and error objects.

3.5 XML Data Format

In our system. XML is the standard format for exchanging data between nodes and components. In this section, we list XML examples of requests, tasks and results. Each data instance in the whole system can be identified using a sequenced ID number and unique node name. The ID number is unique on the node that generated the data instance.

3.5.1 Request

As shown in Fig 10. the XML request data includes a query sequence, detailed BLAST search parameters and the identity of the client that originated the request.

3.5.2 Task

The XML task data includes detailed information about the BLAST search, information about the request and task type information that enable the task to be reassembled later on. See Fig 11.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request id="123456" client = "clientname">
    <Algorithm>blastn</Algorithm>
    <Request Sequence>String of sequence in FASTA format</Request Sequence>
    <Database>database 1</Database>
        .
        .
        .
    <Database>database n</Database>
    <Alignment Option>pairwise</Alignment Option>
</Request>
```

Figure 10: XML Example for BLAST Request.

```
<?xml version="1.0" encoding="UTF-8"?>
<Task id="123456" task_manager = "TaskManager" type="Task">
    <Request id="123456" client = "clientname">
        <Algorithm>blastn</Algorithm>
        <Request Sequence>String of sequence in FASTA format</Request Sequence>
        <Database>database 1</Database>
            .
            .
            .
        <Database>database n</Database>
        <Alignment Option>pairwise</Alignment Option>
    </Request>
</Task>
```

Figure 11: XML Example for BLAST Task.

```
<?xml version="1.0" encoding="UTF-8"?>
<Task id="123456" task_manager = "TaskManager" type="SubTask">
    <Request id="123456" client = "clientname">
        <Algorithm>blastn</Algorithm>
        <Request Sequence>String of sequence in FASTA format</Request Sequence>
        <Database>database</Database>
        <Alignment Option>pairwise</Alignment Option>
    </Request>
    <parent task>parent task id</parent task>
</Task>
```

Figure 12: XML Example for BLAST Subtask.

3.5.3 Subtask

A subtask is almost the same as a task. The difference is that a subtask only contains one database and contains a parent task ID (Fig 12).

3.5.4 Result

XML result data not only includes the BLAST results data but also contains all of the task data for the task and the request. In the result assembling stage, if two different results (with different ids and server-name combination) come from the same task, they are merged into one result (Fig 13) [24].

3.6 Class Diagrams

This section presents the Java packages and classes used in this system. Our system contains the following packages:

- **blastbasic:** contains all utility classes and superclasses for other packages.
- **blastclient:** contains all classes to construct a Jini BLAST Client.

```
<?xml version="1.0" encoding="UTF-8"?>
<Result id="123456" BLAST_Server = "BLAST server name">
  <Task id="123456" task_manager = "TaskManager" type="SubTask">
    <Request id="123456" client = "clientname">
      <Algorithm>blastn</Algorithm>
      <Request_Sequence>String of sequence in FASTA format</Request_Sequence>
      <Database>database</Database>
      <Alignment_Option>pairwise </Alignment_Option>
    </Request>
    <parent_task>parent task id</parent_task>
  </Task>
  <results_data on_complement_strand="true" confidence="100">
    <algorithm> blast2x version="2.0.11" </algorithm>
    <database>nr</database>
    <result_type>blast_grp</result_type>
    <match_desc>&gt;gi|7513096|pir||JE0225 JH8 protein ? human </match_desc>
    <query_region start="23981" end="24850">
      <db_id id="AC069411F1" version="AC069411F1.10" db_code="gb"/>
    </query_region>
    <match_region start="2" end="289">
      <bio_sequence element_id="SP78QZH69TJ169Q">
        <db_id id="JE0225" db_code="pir"/>
      </bio_sequence>
    </match_region>
    <match_align>
Query:24850 SVM*DLRKQKDTLWKFYAESDEQKLMKNRKTLPNVKNKDLSQVLRDQICQCCSEHMPLNG 24671
            S +  +R  K  L +F+A SD  K ++ R+TL   K + L +VL +      SE +P++G
Sbjct: 2    STLYSIRAHKAQLLRFFASSDSNKALEQRRTLHTPKLEHLDRVLYEWFLGKRSEGVPVSG 61
    </match_align>
    <result_property prop_type="blast_score">166</result_property>
    <result_property prop_type="blast_pe_scr">2e-38</result_property>
    <result_property prop_type="blast_ident">33</result_property>
    <result_property prop_type="blast_sim">52</result_property>
    <result_property prop_type="blast_gaps">0</result_property>
  </results_data>
</Result>
```

Figure 13: XML Example for BLAST Result.

- **blastserver:** contains all classes to construct a Jini BLAST Server. ıtaskmanager: contains all classes for workflow management and constructing a Task Manager. It contains several classes and two sub-packages:
  - **database:** provides database accessing and data manipulating functionalities.
  - **javaspace:** provides JavaSpaces instance searching functionality.
- **administrator:** contains all classes to construct an Administrator.

We also need the Jini Starter Kit [10] to run instances for Jini Lookup Services and JavaSpaces. In the following sections, we present more details about the packages and their classes related with workflow management. For the GUI related classes, we only provide some programming interfaces.

3.6.1 Blastbasic Package

The BLAST Basic package contains the basic utility classes and superclasses for other packages. As shown in Fig 17, it contains the following classes:

- `JiniNode`: this class is a superclass for the main classes of each node in our system. It provides basic Jini client and server functionalities of finding all available lookup services; to create and register Jini services for each node; to listen for events that happened on the service. `JiniNode` contains several abstract functions that allow the subclass to implement different services searching, matching and downloading functions.
- `BasicData`: the superclass for `BLASTRequest`, `BLASTTask` and `BLASTResult`. The abstract function *parseData()* guarantees that subclasses can fetch the necessary object property information from XML data via their own implementation. See Fig 14.

```
package blastbasic;

import java.io.Serializable;

abstract public class BasicData implements Serializable
{
 //the contents of data in XML format
 protected String strXMLData = "";
 //the number asigned at the node
 protected long lngSequenceNumber;
 //where the data created
 protected String strNodeName="";

 public BasicData(){}

 public void setData(String argXMLData){strXMLData = argXMLData;}
 public String getData(){return strXMLData;}

 public String getNodeName(){return strNodeName;}
 public long getSequenceNum(){return lngSequenceNumber;}
 public String getID(){return strNodeName+lngSequenceNumber;}

 public boolean equals(Object argData){
  BasicData tmpData = (BasicData)argData;
  return strNodeName.equals(tmpData.strNodeName)
  &&(lngSequenceNumber==tmpData.lngSequenceNumber);
 }

 abstract public void parseData();
 abstract public void createXMLData();
}
```

Figure 14: Class `BasicData`.

```
package blastbasic;
import net.jini.core.entry.Entry;

abstract public class Infor implements Entry{
 public static final String SERVICETYPE_BLASTSERVER = "BLAST Server";
 public static final String SERVICETYPE_TASKMANAGER = "Task Manager";
 public static final String SERVICETYPE_CLIENTADMIN = "Client Admin";

 public String strServiceType = "";
 public String strName     = "";
 //physical address for the service node
 public String strLocation  = "";
 public String strIPAddress = "";

 public Infor(){}

 abstract public void initialize(String argIniFile);
}
```

Figure 15: Class `Infor` Lists All Service Properties.

- `Infor`: an implementation of the `Entry` interface, working as Jini service properties. The immediate subclasses are `ServerInfor`, `TaskManagerInfor` and `ClientInfor`. As shown in Fig 15, it publishes service properties to the whole system.

- `BasicService`: an interface for all Jini services used in our system. Also provides an administration API for changing the status. Since the BLAST search needs to read sequences from the database, and since the Task Manager must exchange data with the backup database, our Jini service architecture uses an RMI based thin proxy. As shown in Fig 16, the `BasicService` interface extends from `Remote`. See Section 2.2.3.

```
package blastbasic;
import net.jini.core.event.*;
import net.jini.core.lookup.ServiceID;
import net.jini.lookup.ServiceIDListener;
import net.jini.core.entry.Entry;
import java.io.Serializable;
import java.rmi.*;

public interface BasicService extends Remote,
  Serializable, ServiceIDListener{

  public int getStatus() throws RemoteException;
  public int getServiceType()throws RemoteException;
  public void changeStatus(int argStatus)throws RemoteException;
  public EventRegistration addRemoteListener(RemoteEventListener listener)
      throws java.util.TooManyListenersException, RemoteException;
  public void fireNotify(RemoteEvent event)throws RemoteException;
  public Entry[] getInfor()throws RemoteException;

  public ServiceID getServiceID()throws RemoteException;
}
```

Figure 16: `BasicService` Interface

- `BasicServiceImp`: the implementation of `BasicService`. the superclass for all service implementation. It notifies all clients who have downloaded its proxy.

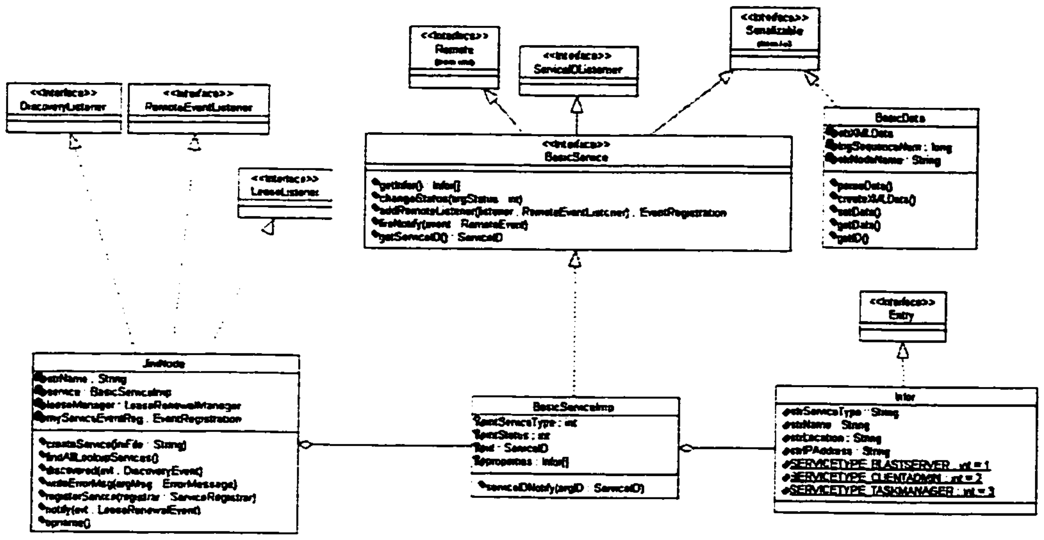

Figure 17: Class Diagram for BLAST Basic Package

3.6.2 Blastserver Package

The Blastserver package contains the classes for constructing the server node. See Fig 20.

- `BLASTEngine`: the class extends from `Thread` to invoke the BLAST search program written in C++ code using the Java Native Interface (JNI) [23]. See Fig 18.

- `ServerInfor`: the subclass of `blastbasic.Infor` (Fig 15) contains detailed server information. It is published in the Lookup Services. Clients use it to find the desired server. As shown in Fig 19. it lists the server's computing resources.

```
package blastserver;

import java.sql.*;
import taskmanager.BLASTSubTask;
import taskmanager.BLASTResult;
import blastbasic.BasicData;

public class BLASTEngine extends Thread{
  private JiniBLASTServiceImp jiniservice;
  private BLASTSubTask task = null;
  public BLASTEngine(JiniBLASTServiceImp
     argService, ThreadGroup argGroup){
    super(argGroup, "");
    jiniservice = argService;
    System.loadLibrary("blastlb");
  }
  public void setTask(BLASTSubTask argTask){task = argTask; }

  public void run(){
    result = executeBLAST(task);
    jiniservice.addResult(result);
  }

  public native BLASTResult executeBLAST(BLASTSubTask argTask);
}
```

Figure 18: BLASTEngine with JNI Function Call

```
package blastserver;

import blastbasic.Infor;

public class ServerInfor extends Infor{

 public int intNumOfCPU;
 public int intTotalRam;
 public String Database;
 public String strAlgorithm;
 public int intTotalCPUSpeed;

 public ServerInfor(){
  super();
 }

 public void initialize(String argIniFile)
 {
  //following code read in properties from ini file.
 }
}
```

Figure 19: `ServerInfor` Contains BLAST Service Properties.

- JiniBLASTService: the BLAST service proxy interface. It is used to create a service template to search the proxy implementation.

- JiniBLASTServiceImp: the implementation of JiniBLASTService. The maximum BLASTEngine number limits the number of concurrent BLAST search threads running on each server.

- BLASTServer: the subclass of blastbasic.JiniNode to start program; find lookup Services and register its own Jini BLAST service.

- ServerEvent: subclass of RemoteEvent. serves as an event-message for notifying JiniTaskManager.

3.6.3 Blastclient Package

The BLASTClient package provides a user interface to get user input and display results. See Fig 21, Fig 22, Fig 23.

- ResultForm: GUI to display BLASTResults.

- InputForm: GUI to get user input.

- ServiceListForm: the graphic user interface to list JiniTaskManagerServices and JiniBLASTServices. The user can browse and select the appropriate one for each request.

- JiniBLASTClient: the subclass of basic.JiniNode to start the client; find and download the JiniTaskManagerServices; register the client admin service.

- JiniClientAdminService: Jini service provides communication between the BLAST Client and the Administrator.

- ClientInfor: the subclass of blastbasic.Infor to provide necessary BLAST Client information to the Administrator.

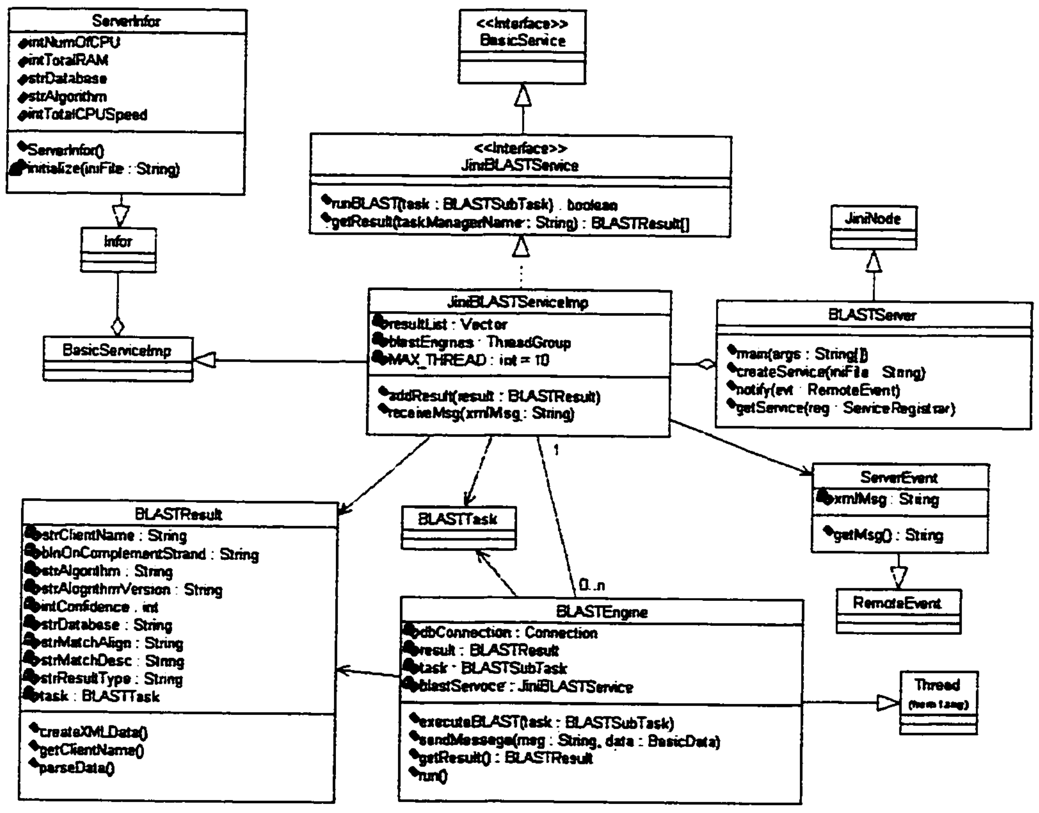

Figure 20: Class Diagram for the BLAST Server Package

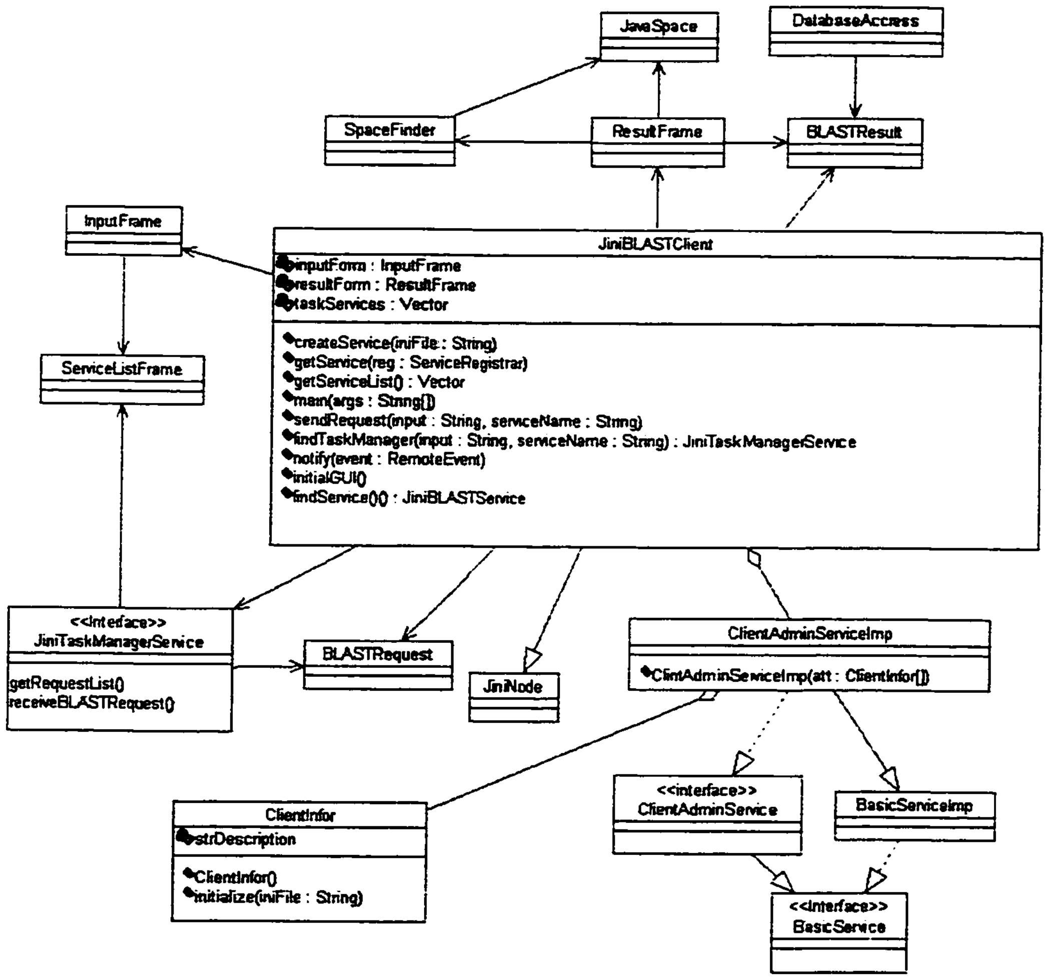

Figure 21: Class Diagram for the BLAST Client Package

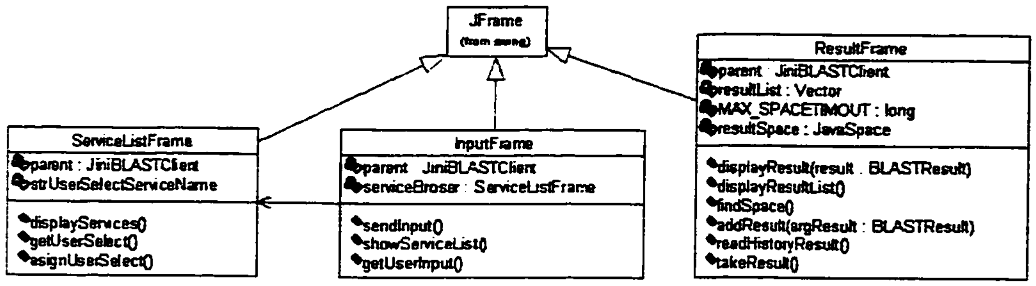

Figure 22: Class Diagram for BLAST Client GUI

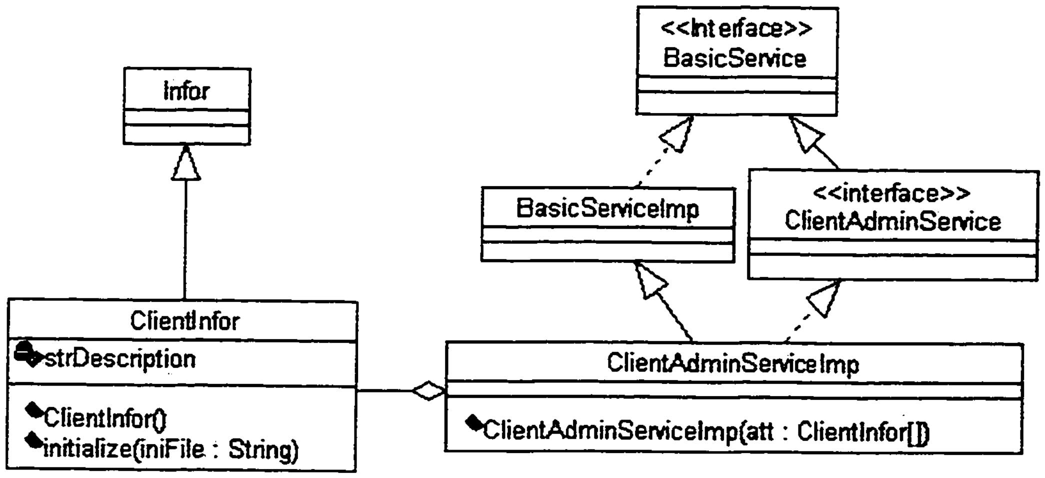

Figure 23: Class Diagram for the BLAST Client Admin Service

```
package taskmanager;

import blastbasic.*;
import blastserver.ServerInfor;
import java.util.Vector;

public class TaskManagerInfor extends Infor{

 public Vector serverPropertyList = new Vector();

 public TaskManagerInfor(){}

 public void initialize(String argIniFile){
  /*todo: implement this blastbasic.Infor abstract method*/
 }

 public void addServerInfor(ServerInfor argInfor){
  serverPropertyList.add(argInfor);
 }
}
```

Figure 24: Class `TaskMangerInfor` Contains BLAST Services Information

3.6.4 Taskmanager Package

The Taskmanager package contains all of the classes for workflow management. See Fig 25, Fig 26 and Fig 27. It contains the following classes and sub packages:

- `TaskManagerInfor`: the subclass of `blastbasic.Infor` (Fig 15). It works as an `Entry` object for publishing `JiniTaskManagerService` properties. As shown in Fig 24, it also contains all available `JiniBLASTServices` information for each Task Manager. In this way, the client can receive a transparent BLAST service list without knowing the existence of the Task Manager.

- `JiniTaskManager`: the subclass of `JiniNode`. Main program of Task Manager. Creates and registers an instance of `JiniTaskManagerService` and downloads `JiniBLASTService`.

- `JiniTaskManagerService`: Jini service interface for Task Manager Service.
- `JiniTaskManagerServiceImp`: the class implements `JiniTaskManagerService`. It extends from `BasicServiceImp`.
- `FailureManager`: extended from `Thread` to continually check errors from the Error Space and take action to handle the failure of BLAST Client. BLAST Server and other Task Managers.
- `BLASTRequest`: contains request information created on the client and passed to `JiniTaskManagerService`. It contains the fields in XML data (Section 3.5.1).
- `BasicTask`: is the superclass of `BLASTTask` and `BLASTSubTasks`.
- `BLASTTask`: contains task information and may contain subtasks.
- `BLASTSubTask`: generated from `BLASTTask` to contain a single job. In our case, each `BLASTSubTask` only contains a request against one database partition. It is created on the Task Manager and is passed to the BLAST Server.
- `BLASTResult`: contains result data. It is created on the server and is returned to the Task Manager. Since it is written into the Result Space. it also implements the `Entry` interface.
- `TaskDispatcher`: extended from `Thread`. Creates tasks and subtasks, dispatches subtask to the `JiniBLASTService`.
- `ResultAssembler`: prepares and merges results created from the same task.
- Package `Javaspaces`: contains one class: `SpaceFinder`, to find an instance of `JavaSpaces` using a space name.
- Package `database`: sub-package contains one class `DatabaseAccess`. to provide basic database access and data manipulation functions.
- `BaseEventMessage`: the class to hold an event message.

- **LogMessage**: the subclass of **BaseEventMessage** to hold the log messages for all events.

- **ErrorMessage**: objects to hold error information. They may be created on any node. and are written into error space. **FailureManager** reads this object and executes the error handling code.

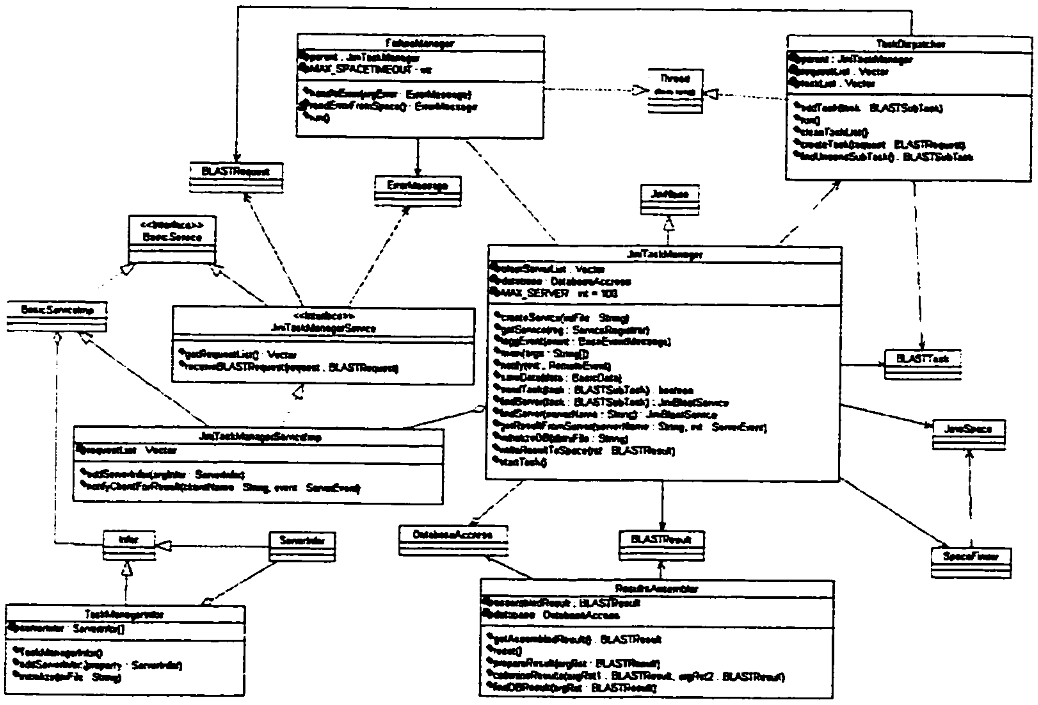

Figure 25: Class Diagram for the Task Manager Package

3.6.5 Administrator Package

The Administrator package provides an administrative interface to monitor the system runtime architecture and activity history. It also enables the user to block nodes in

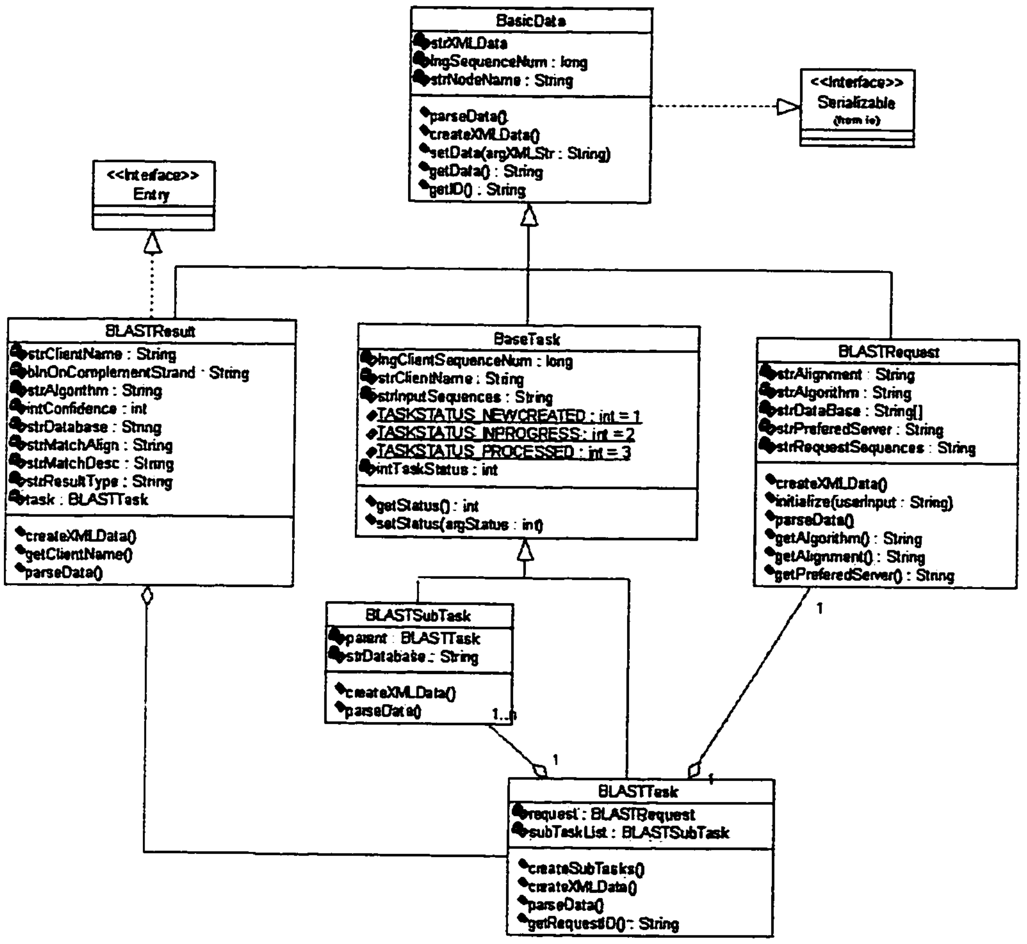

Figure 26: Class Diagram for the Data Classes

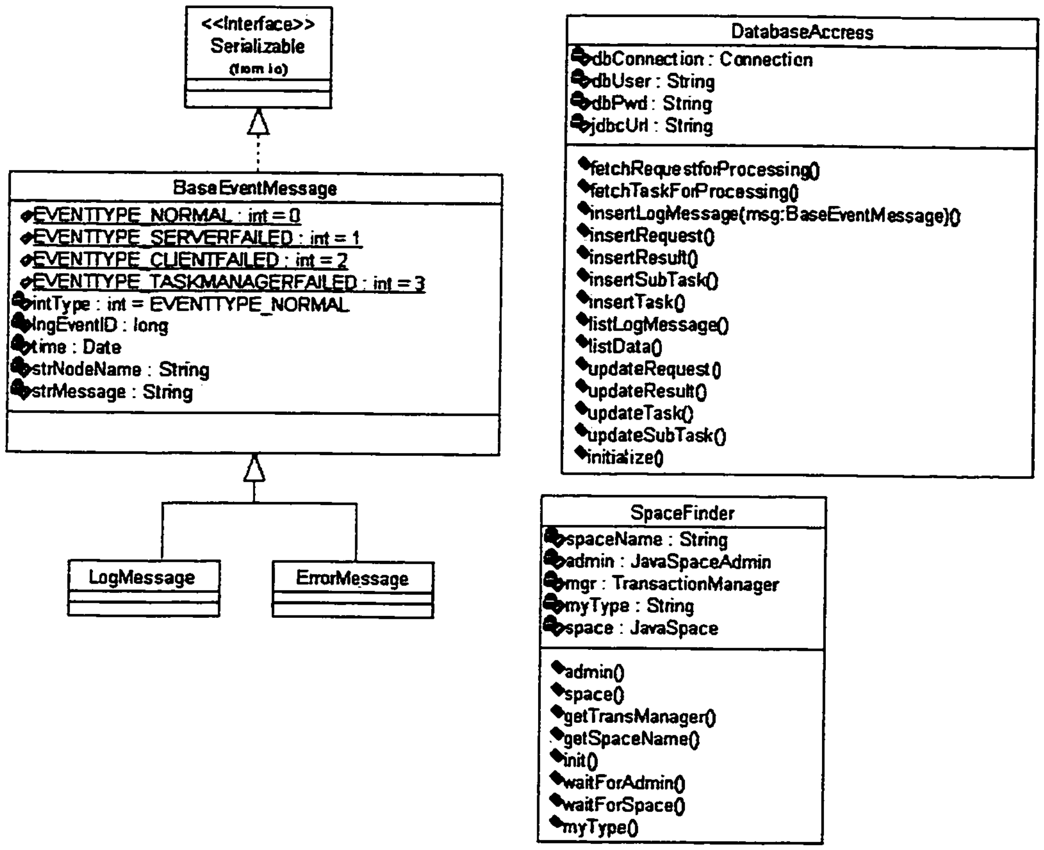

Figure 27: Class Diagram for the Message classes and Task Manager Sub-packages

the system. The current design is simple. However, in the future, the administrator's job can be extended to manually modify tasks and work balance. It contains the following classes (Fig 29):

- `JiniSystemAdmin`: the subclass of `JiniNode`. It finds all Jini services in the system.
- `ServiceNode`: an object to represent a system node in order to create a runtime architecture GUI picture.
- `JiniServiceListFrame`: GUI to display system runtime architecture.
- `ServiceActivityFrame`: GUI to let the user review blast activities and query old activities in the database.

3.7 How the System Works

In this section, we present some sequence diagrams to explain how the system works. We describe the following system activities:

- Register and Download Services: describe how the Jini service node registers the service proxy and how the Jini client downloads the service proxy.
- Client send Request: describe how BLAST Client sends a request to TaskManager.
- Creating and Dispatching Task: describe detail information about Task Manager creating and dispatching tasks.
- Write Results to ResultSpace: describe how results are sent back to TaskManager and finally written into ResultSpace.
- Read Results from ResultSpace: describe how client reads results from ResultSpaces.

```
package administrator;

import blastbasic.*;
import net.jini.core.lookup.ServiceID;
import net.jini.core.entry.Entry;

public class ServiceNode{

  private BasicService service;
  private ServiceID id;
  private Entry[] serviceProperties = null;

  public ServiceNode(BasicService argService,
  ServiceID argID, Entry[] entries){
    service = argService;
    id = argID;
    serviceProperties = entries;
  }

  public boolean equals(Object argNode){

    ServiceNode tmpNode = (ServiceNode)argNode;
    return tmpNode.id.equals(id);
  }

  public void updateProperties(Entry[] argEntries){

    serviceProperties = argEntries;
  }
}
```

Figure 28: Class `ServiceNode` Creates a Representative for each Node.

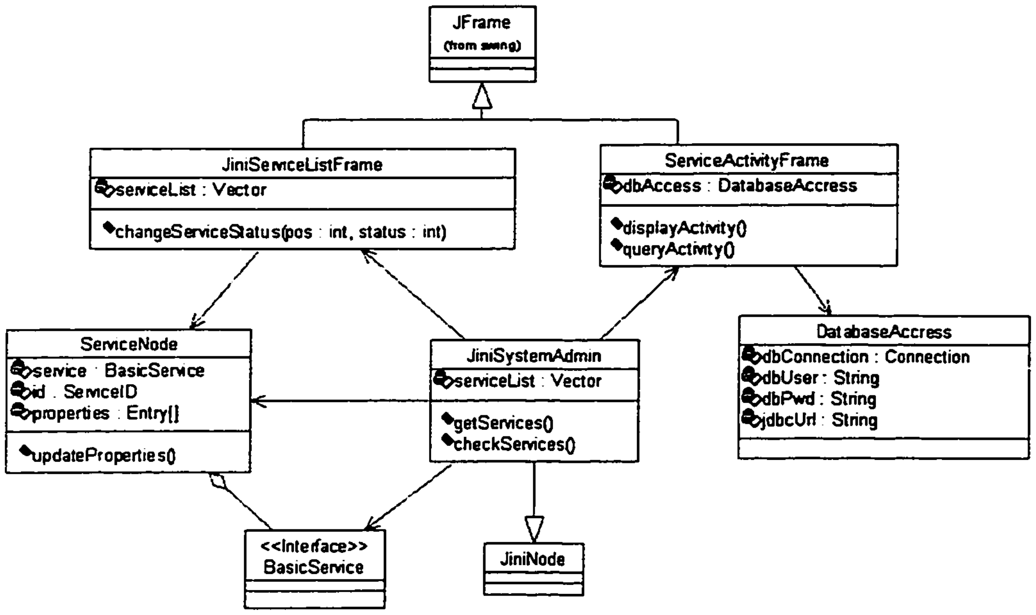

Figure 29: Class Diagram for the Administrator Package

- Failure Handling: describe how TaskManager handles a network exception.

3.7.1 Register and Download Services

In our system, each node has a main program extended from `blastbasic.JiniNode`. First, as shown in Fig 30, each node, except the Administrator, creates its own Jini service. Then, it finds all Lookup Services by creating a `LookupDiscover` instance. The Jini node is added as a `DiscoverListener` for the `LookupDiscover` instance. An instance of `DiscoveryEvent` is passed back to the node. The node's main program can get a registrar from the `DiscoveryEvent` object and then register its services by calling its *register()* function. An instance of `ServiceRegistration` is returned. Each node can get its service ID from `ServiceRegistration` and assign the service ID to its own service. It is used to renew the node's service lease or used for re-registration so the same service can keep the same ID.

When the node receives the `ServiceRegistrar` instance, the Jini node can use it to find and download all matched Jini services (Fig 30).

The sequence diagram presents some general ideas about how service registration and match is done. As shown in Fig 31, Fig 32, Fig 33 and Fig 34, we use the Task Manager as an example to show the inheritance relationships and function calls for Jini service registration and downloading. `JiniTaskManger` implements the abstract method *createService()* to create an instance of `JiniTaskManagerServiceImp`. The superclass `JiniNode` implements the general methods *findAllLookupServices()* and *discovered()* to allow all nodes to find all lookup services and register their services. The *registerSerivce()* in the superclass helps each node register its own Jini Service. `JiniTaskManager` also implements the abstract method *getService()* to download proxies of `JiniBLASTServiceImp`. The `JiniBLASTServer` uses the same algorithm to register a proxy of `JiniBLASTServiceImp`. `JiniBLASTClient` and `JiniSystemAdmin` also find and download necessary proxies using this same method.

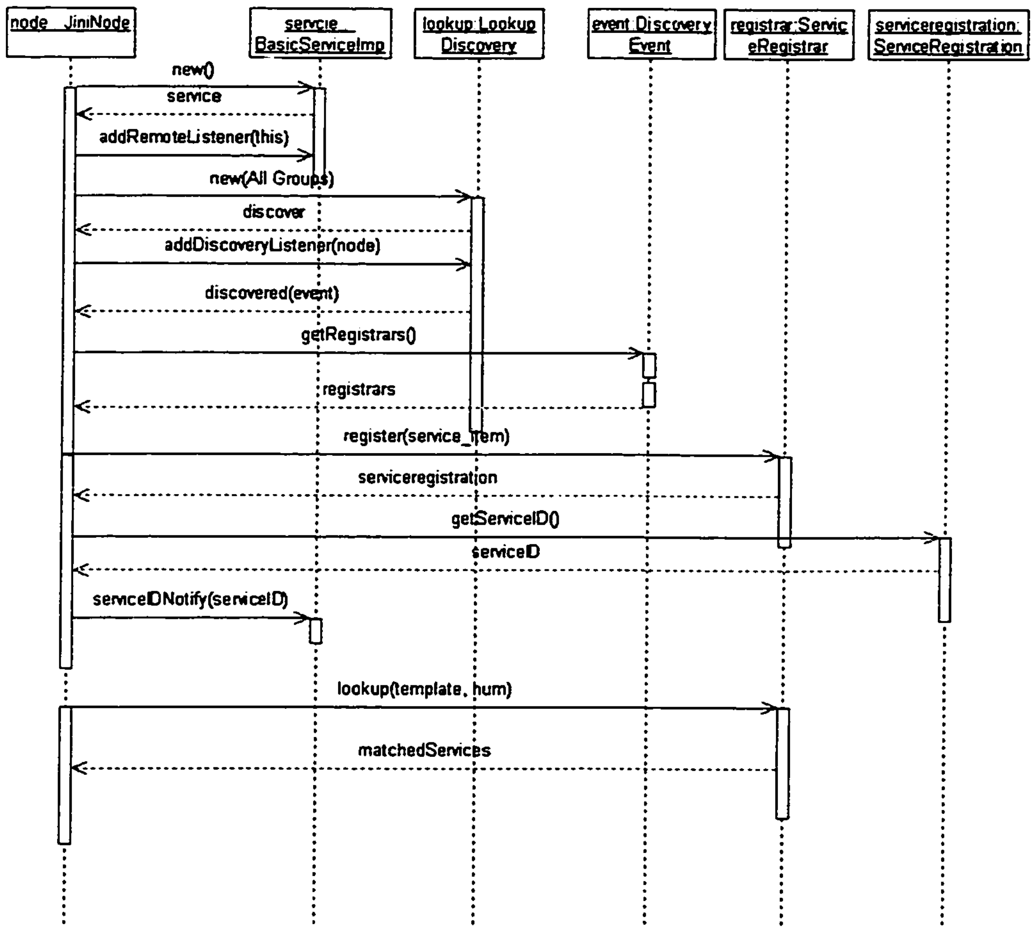

Figure 30: Sequence Diagram for Jini Services Registration and Downloading.

```
abstract public class JiniNode implements DiscoveryListener,
LeaseListener, RemoteEventListener{
 protected String strName; //node name
 protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
 protected BasicServiceImp service = null;
 protected EventRegistration myServiceEventReg = null;

 public JiniNode(){ }

 public JiniNode(BasicServiceImp argService) {
  service = argService;
  findAllLookupServices();
 }

 //must read an ini file to set service properties
 abstract public void createService(String argIniFile);
 //download service
 abstract public void getService(ServiceRegistrar argRegistrar);

 //find all lookup services by multicast
 public void findAllLookupServices(){
  System.setSecurityManager(new java.rmi.RMISecurityManager());
  LookupDiscovery discover = null;
  try{
       discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
  }catch(Exception e){
       e.printStackTrace();
  System.exit(1);
  }
  discover.addDiscoveryListener(this);
 }
 .
 .
 .
}
```

Figure 31: Class `JiniNode` Provides Basic Jini Service and Client APIs.

```
abstract public class JiniNode implements DiscoveryListener,
LeaseListener, RemoteEventListener{
.
.
.
 public void discovered(DiscoveryEvent evt){
  //get all registrar list
  ServiceRegistrar[] registrars = evt.getRegistrars();
  try{
   for (int n = 0; n < registrars.length; n++){
  ServiceRegistrar registrar = registrars[n];

      registerService(registrar);


      getService(registrar);
   }
  }catch(java.rmi.RemoteException exp){
   //error handling
  }
 }

 public void registerService(ServiceRegistrar argRegistrar) throws java.rmi.RemoteException{
  //
  ServiceItem item = new ServiceItem(null, service, service.getInfor());
  // Export a service
  ServiceRegistration reg = argRegistrar.register(item, Lease.FOREVER);
  service.serviceIDNotify(reg.getServiceID());

  leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
 }
}
```

Figure 32: The *discovered()* and *registerService()* Functions in Class `JiniNode`

```
package taskmanager;

public class JiniTaskManager extends JiniNode{
 private Vector blastServerList = new Vector();
  .
.
.
 public void createService(String argIniFile){
  /*todo: implement this blastbasic.JiniNode abstract method*/
  TaskManagerInfor[] properties = new TaskManagerInfor[1];
  TaskManagerInfor taskManagerProperties = new TaskManagerInfor();
  taskManagerProperties.initialize(argIniFile);
  taskManagerProperties.strServiceType = Infor.SERVICETYPE_TASKMANAGER;
  properties[0] = taskManagerProperties;
  JiniTaskManagerServiceImp taskMangerService =null;
  try{
   taskMangerService = new JiniTaskManagerServiceImp(properties);
   myServiceEventReg = taskMangerService.addRemoteListener(this);
   super.service=taskMangerService;
  }catch(java.util.TooManyListenersException exp){
    //do nothing
  }catch(RemoteException exp){
   //do someting
  }  //set service
 }
 .
.
.
}
```

Figure 33: Class `JiniTaskManger` Implements the Function *createService()* to Create `JiniTaskManagerService` Instance.

```
package taskmanager;

public class JiniTaskManager extends JiniNode{

 private Vector blastServerList = new Vector();
 .
.
.
 public void getService(ServiceRegistrar argRegistrar){
  /*todo: implement this blastbasic.JiniNode abstract method*/
  Class[] classes = new Class[] {JiniBLASTService.class};
  ServiceTemplate template = new ServiceTemplate(null, classes, null);
  TaskManagerInfor tmpInfor = null;
  ServiceMatches matchedServices = null;
  //get task manager service properties
  try{
    matchedServices = argRegistrar.lookup(template, 100);
    tmpInfor =(TaskManagerInfor)service.getInfor()[0];
  }catch(RemoteException exp){
    //do something
  }
  for(int i = 0; i < matchedServices.items.length; i++){
    JiniBLASTService tmpService = (JiniBLASTService)matchedServices.items[i].service;
    if(tmpService != null){
     tmpInfor.addServerInfor((ServerInfor)matchedServices.items[i].attributeSets[0]);
     blastServerList.addElement(tmpService);
    }
  }
}
.
.
.
}
```

Figure 34: Class `JiniTaskManger` Implements the *getService()* Function to Download All Available `JiniBLASTServices`.

3.7.2 Client Send Request

The user issues a command on `InputFrame` to pass all data and parameters to `JiniBLASTClient`. The latter creates a new `BLASTRequest` instance and initialize it with user input. Then the client finds a `JiniTaskManagerService` matching the user input or using a service name. The client then calls the *receiveBLASTRequest()* function and passes the request to it. The `JiniTaskManagerService` adds the request into a pending `BLASTRequest` list. The client also adds itself as a `RemoteListener` to that `JiniTaskManagerService` in order to listen to events occurring in the `JiniTaskManager`. See Fig 35, Fig 36.

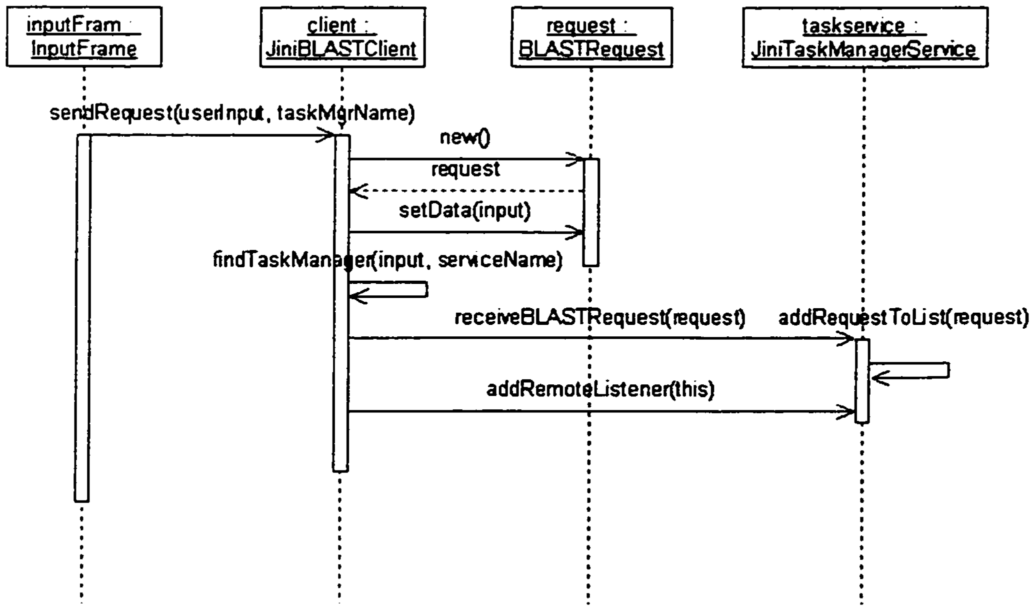
inputFram : InputFrame
client : JiniBLASTClient
request : BLASTRequest
taskservice : JiniTaskManagerService
sendRequest(userInput, taskMgrName)
new()
request
setData(input)
findTaskManager(input, serviceName)
receiveBLASTRequest(request)
addRequestToList(request)
addRemoteListener(this)

Figure 35: Sequence Diagram for Sending a Request to the `JiniTaskManager`

```
package blastclient;

public class JiniBLASTClient extends JiniNode {
  .
.
.
  //argUserInput in XML format
  public void sendRequest(String argUserInput, String argTaskServiceName){
      BLASTRequest newRequest = new BLASTRequest();
      newRequest.setData(argUserInput);
//find task manager service proxy by name or by user input
      JiniTaskManagerService tmpService = findService(argUserInput, argTaskServiceName);

//add this as remotelistener to service
try{
       tmpService.addRemoteListener(this);
       tmpService.receiveBLASTRequest(newRequest);
      }catch(java.util.TooManyListenersException exp){
       //add error handling
      }catch(RemoteException exp){
       //create error mesaage and write it to error space
      }
  }
  .
.
.
}
```

Figure 36: `JiniBLASTClient` Sends Requests to the `JiniTaskMangerServiceImp`.

3.7.3 Creating and Dispatching Task

The JiniTaskManager gets the BLASTRequest list from JiniTaskManagerServiceImp and uses it to create a new TaskDispatcher. Since the TaskDispatcher and the JiniTaskManagerServiceImp share the same BLASTRequest list, every time a new request is added to the list in JiniTaskManagerServiceImp, the TaskDispatcher can read it. Then JiniTaskManager starts a TaskDispatcher thread. If the request list is not empty, the TaskDispatcher removes a request from the list and creates a new BLASTTask using that request. At the same time, all BLASTSubTasks are created and added into the job queue. Then TaskDispatcher continues to take a BLASTSubTask and pass it back to the JiniTaskManger by calling *sendTask()* until the job queue is empty. JiniTaskManager finds a JiniBLASTService based on the BLASTSubTask data, and passes that BLASTSubTask to the JiniBLASTService by calling *runBLAST()*.

In JiniBLASTServiceImp, the constant integer MAX_BLASTPROCESSES limits the total number of BLAST search processes running concurrently on that server. The instance of ThreadGroup manages all BLAST threads running on the same server (Fig 40). If there are enough resources, or we have not exceeded the search engine threads limits, then the JiniBLASTService starts a new BLASTEngine thread to execute a BLAST search. If everything is fine, a boolean variable is returned to the TaskDispatcher. The TaskDispatcher updates the task status and reorganizes the task list to remove dispatched tasks from the job queue. If the task is not dispatched successfully, it stays in the job queue and waits to be sent out again. See Fig 37, Fig 38 and Fig 39.

3.7.4 Write Results to ResultSpace

As shown in Fig 18, after the BLASTEngine finishes executing the BLAST search, it creates a new BLASTResult and adds it to the result list in the JiniBLASTService-Imp instance. The latter creates a ServerEvent and notifies the JiniTaskManager,

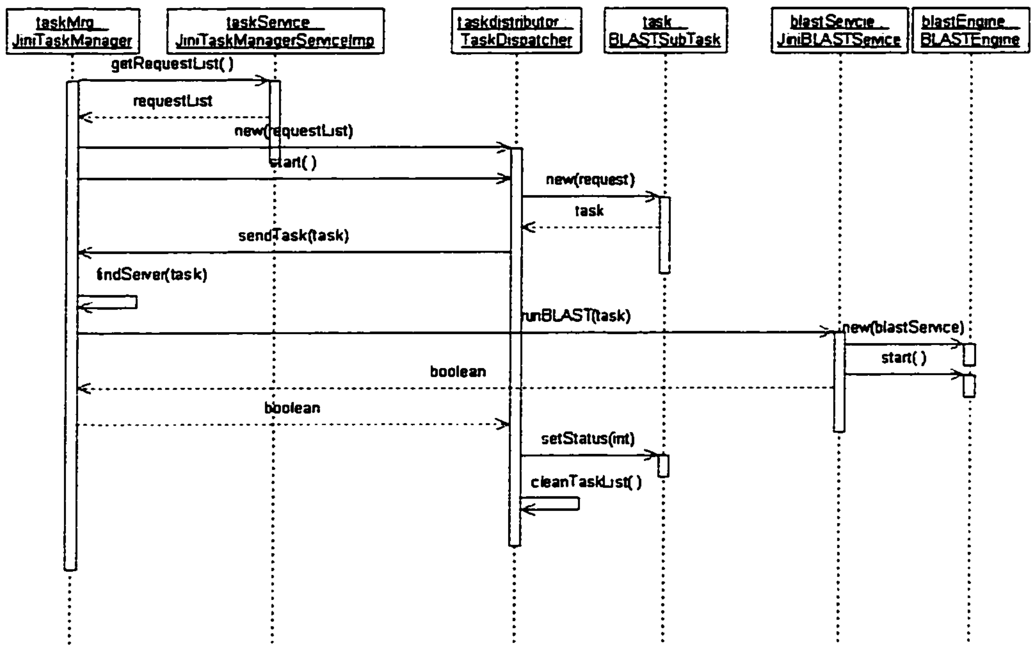

Figure 37: Sequence Diagram for Creating and Dispatching Tasks

```
public class JiniTaskManager extends JiniNode{
  private Vector blastServerList = new Vector();
  private TaskDispatcher taskDistributor = null;
  private FailureManager failureHandler = null;
                    .
                    .
                    .
  public boolean sendTask(BLASTSubTask argTask){
    JiniBLASTService tmpBLAST = findServer(argTask);
    boolean sendSuccess = true;
    try{
      if(tmpBLAST.runBLAST(argTask)){
        tmpBLAST.addRemoteListener(this);
      }

    }catch(java.util.TooManyListenersException exp){
        //do nothing

    }catch(RemoteException exp){
    //do something
    }
    return sendSuccess;
  }
  private void startTask(){
    //cast the service in super class to JiniTaskManagerSericeImp
    Vector tmpRequestList = ((JiniTaskManagerServiceImp)super.service).getRequestList();
    taskDistributor = new TaskDispatcher(tmpRequestList, this);
    taskDistributor.start();
    failureHandler = new FailureManager(this);
    failureHandler.start();
  }
}
```

Figure 38: `JiniTaskManger` Sends Tasks to `JiniBLASTServiceImp` via Proxy.

```
public class TaskDispatcher extends Thread{
 private JiniTaskManager parent = null;
 private Vector requestList = null;
 private Vector taskList = new Vector();

 public TaskDispatcher(Vector argRequestList, JiniTaskManager argManager){
  requestList = argRequestList;
  parent = argManager;
 }

 public void run()  {

  while(true){
   if(!requestList.isEmpty()){

    BLASTRequest tmpRequest = (BLASTRequest)requestList.remove(0);
    parent.saveData(tmpRequest);
    createTask(tmpRequest); //create new task and add it to list
   }

   if(!taskList.isEmpty()){
    BLASTSubTask tmpTask = findUnsentTask();
    if(parent.sendTask(tmpTask)){//update task status
     tmpTask.setStatus(BaseTask.TASKSTATUS_INPROGRESS);
     //update Task or subTask status.
    }
    cleanTaskList();//remove sent out task
   }
   //sleep two seconds for each loop
   try{
    sleep(2*1000);
   }catch(InterruptedException exp){

   }
  }
 }
.
.
.
}
```

Figure 39: Class `TaskDispatcher` Begins a Thread to Create and Distribute Subtasks.

```
package blastserver;

public class JiniBLASTServiceImp extends BasicServiceImp
implements JiniBLASTService{
.
.
.
 //max number of BLAST engien can be executed concurrently
 private final int MAX_BALSTPROCESSES = 10;
 private ThreadGroup blastEngines = new ThreadGroup("BLAST Engines");

 public boolean runBLAST(BLASTSubTask argTask)throws RemoteException{
  /*todo: Implement this blastserver.JiniBLASTService method*/
  if (blastEngines.activeCount()<this.MAX_BALSTPROCESSES){
   //running BLAST search and create result then notify TASKManager
   BLASTEngine newBLAST = new BLASTEngine(this, blastEngines);
   newBLAST.setTask(argTask);
   newBLAST.start();
   return true;
  }
  else
   return false;
 }
}
```

Figure 40: Class `JiniBLASTServiceImp` Manages a `BLASTEnginge` Thread Group and Limits the Number of Processes Running on BLAST Server.

where the BLASTTask comes from. The JiniTaskManager receives all of its own results from the JiniBLASTService and creates a new ResultAssembler instance. The ResultAssembler instance checks each result and tries to merge it with the other results (if there are any) which were created from the same task. After preparing the results, the JiniTaskManger creates a new instance of SpaceFinder and uses it to get a proxy of the JavaSpaces instance. Then a new BLASTResult is written into the space. See Fig 41, Fig 42 and Fig 43.

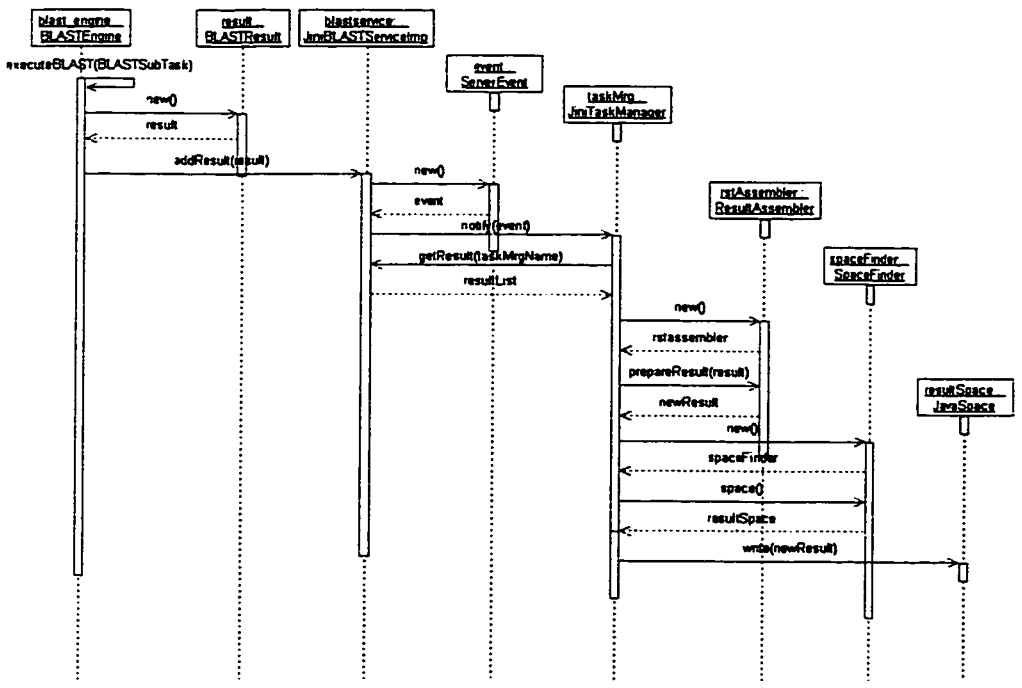

Figure 41: Sequence Diagram for Getting Result and Writing it to ResultSpace

3.7.5 Read Results from ResultSpace

After the JiniTaskManager writes a BLASTResult to the Result Space, it calls the *notifyClientForResult()* function of JiniTaskManagerServiceImp (see Fig 43). The

```
package blastserver;

public class JiniBLASTServiceImp
extends BasicServiceImp implements JiniBLASTService{
.
.
.
  public void addResult(BLASTResult argResult){
   resultList.add(argResult);
   prepareNotify(BLASTMSG_FINISH);
  }
 public BLASTResult[] getResult(String argTaskManagerName)
    throws RemoteException{
   //search result from the list and return
   BLASTResult[] tmpResult;
   //add code here to search results in the List for given TaskManagerName
   return tmpResult;
 }
 private void prepareNotify(String msg){
   /**
    * prepare XML message in following format

   <MSG>
     <EventTyep>Result Ready</Event>
     <EventID node="nodeame" sequenceNum="123456789"/>
     <Server>Server Name</Server>
     <TaskManager>Task Manager Name</TaskManager>
     <Clientt>Client Name</Client>
   </MSG>
   */
   //will be replaced by real message
   String xmlMsg = "Message In XML formate";
   ServerEvent event = new ServerEvent(this, BLASTEvent.SERVER_EVENT,
       System.currentTimeMillis(), null, xmlMsg);
   super.fireNotify(event);
 }
}
```

Figure 42: Class JiniBLASTServiceImp Adds the Result to the List and Notify JiniTaskManager.

```
package taskmanager;

public class JiniTaskManager extends JiniNode{
.
.
.
 public void notify(RemoteEvent argEvt){
  //make sure the event is from taskmanager service of the same node
  if( argEvt instanceof ServerEvent){
    //looking for BLAST Service name
    String strServerName = parseXMLMessage(((ServerEvent)argEvt).getMsg(), "Server");
    String strEventType = parseXMLMessage(((ServerEvent)argEvt).getMsg(), "EventTyep");
    if(strEventType.equals("Result Ready"))
     getResultFromServer(strServerName, (ServerEvent)argEvt);
  }
 }
 //use server name to find BLASTService and getResult from that service
 private void getResultFromServer(String argServerName, ServerEvent argEvt){
  JiniBLASTService tmpService = findServer(argServerName);
  BLASTResult tmpResults[] = null;
  try{
   tmpResults = tmpService.getResult(getNodeName());
  }catch(RemoteException exp){
   //add error handling
  }
  ResultsAssembler assembler = new ResultsAssembler(database);
  for (int i = 0; i < tmpResults.length; i++){
   assembler.reset(); //reset the assembled data to null;
   assembler.prepareResult(tmpResults[i]);
   BLASTResult newResult = assembler.getAssembledResult();
   writeResultToSpace(newResult);
    // cast service in super class to a JiniTaskManagerServiceImp
   ((JiniTaskManagerServiceImp)service).notifyClientForResult(newResult.getClientName(), argEvt);
  }
 }
 //this funcition will search a java space and write the result to space.
 private void writeResultToSpace(BLASTResult argResult){
  //add code here to write result to java space
  SpaceFinder spaceFinder = new SpaceFinder("BLASTResult");
  JavaSpace resultSpace = spaceFinder.space();
  try{
   resultSpace.write(argResult, null, Long.MAX_VALUE);
  }catch(RemoteException exp){
    //wait for a while and re-write again
  }catch(net.jini.core.transaction.TransactionException exp){
  }
 }
}
```

Figure 43: Class TaskManager Gets Results from the BLAST Service and Writes Results to the ResultSpace.

latter finds the correct `JiniBLASTClient` using the client's name and notifies it with a "result ready message" (Fig 44). The `JiniBLASTClient` interprets that event and calls the *takeResult()* function from the `ResultFrame` instance (Fig 45). The `ResultFrame` instance finds the ResultSpace via the `SpaceFinder` instance. It creates a `BLASTResult` template using the client's name. All matched results in the space are taken to the client and sent to the result GUI (Fig 46). For more information see sequence diagram Fig 47.

```
package taskmanager;

public class JiniTaskManagerServiceImp
  extends BasicServiceImp implements JiniTaskManagerService{
  .
.
.
 public void notifyClientForResult(String clientName, ServerEvent argEvt){
   Object[] listeners = listenerList.getListenerList();
   JiniBLASTClient tempClient = null;
   for (int i=0; i<listeners.length; i++){
     if(listeners[i] instanceof JiniBLASTClient){
       tempClient = (JiniBLASTClient)listeners[i];
       if(tempClient.getNodeName().equalsIgnoreCase(clientName))
         tempClient.notify(argEvt);
     }
   }
 }
}
```

Figure 44: Class `JiniTaskManagerServiceImp` Finds an Instance of the `JiniClient` and Notifies it that Result is Ready.

```
public class JiniBLASTClient extends JiniNode{
.
.
.
 //GUI objects
 private ResultFrame resultFrame = null;
 public void notify(RemoteEvent argEvent){
  if( argEvent instanceof ServerEvent){
   //looking for BLAST Service name
   String strServerName = parseXMLMessage(((ServerEvent)argEvent).getMsg(), "Server");
   String strEventType = parseXMLMessage(((ServerEvent)argEvent).getMsg(), "EventTyep");
   if(strEventType.equals("Result Ready"))
     resultFrame.takeResults();
   }
  }
}
```

Figure 45: `JiniBLASTClient` Get Notify Message and Call method *takeResults()*.

3.7.6 Failure Handling

Since this is a distributed computing system, network exceptions and other errors may occur on any node in the system. In general, the failure of the BLAST Client and the Administrator does not cause any major problems in our system. The functionality of Administrator is that of providing a system monitor. This is not mandatory for the BLAST search. The failure of a client, on the other hand, is that it may lose the result of its query. However, the user can issue a command to retrieve the result from ResultSpace or to retrieve results from the database later through another client terminal using the request ID.

We must pay attention to the failure of the Task Manager and the BLAST Server. As shown in Fig 48, in this system, the `JiniNode` contains the implementation of the function *writeErrorMsgToSpace()*. On each node, if there is a network exception or any other error, an `ErrorMessage` instance is created and written to the Error Space. The `ErrorMessage` object contains the error type and error location. For instance,

```
package blastclient;

public class ResultFrame extends JFrame{
  private JiniBLASTClient parent = null;
  private Vector resultList = new Vector();
  private JavaSpace resultSpace = null;
  private final long MAX_SPACETIMEOUT = 10*60*1000;
  public void findSpace(){

    SpaceFinder spaceFinder = new SpaceFinder("BLASTResult");
    resultSpace = spaceFinder.space();
  }
  //read and remove results form result Space
  public void takeResults(){

    BLASTResult resultTemplate = new BLASTResult();

    resultTemplate.strClientName = parent.getNodeName();
    boolean blnFoundResult = true;
    while(blnFoundResult){
     //if found return a result else return null
     try{
       BLASTResult result = (BLASTResult)resultSpace.takeIfExists(
 resultTemplate,null, MAX_SPACETIMEOUT);
       if(result!=null){
        resultList.add(result);
        Thread.sleep(1000); //give some time for other thread.
       }else
        blnFoundResult=false; //stop loop.
     }catch(RemoteException exp){
      //dosomething
     }catch(InterruptedException exp){
      //dosomething
     }catch(TransactionException exp){
      //dosomething
     }catch(UnusableEntryException exp){
      //dosomething
     }
    }
  }
}
```

Figure 46: `ResultFrame` Gets Results from ResultSpace for a Client.

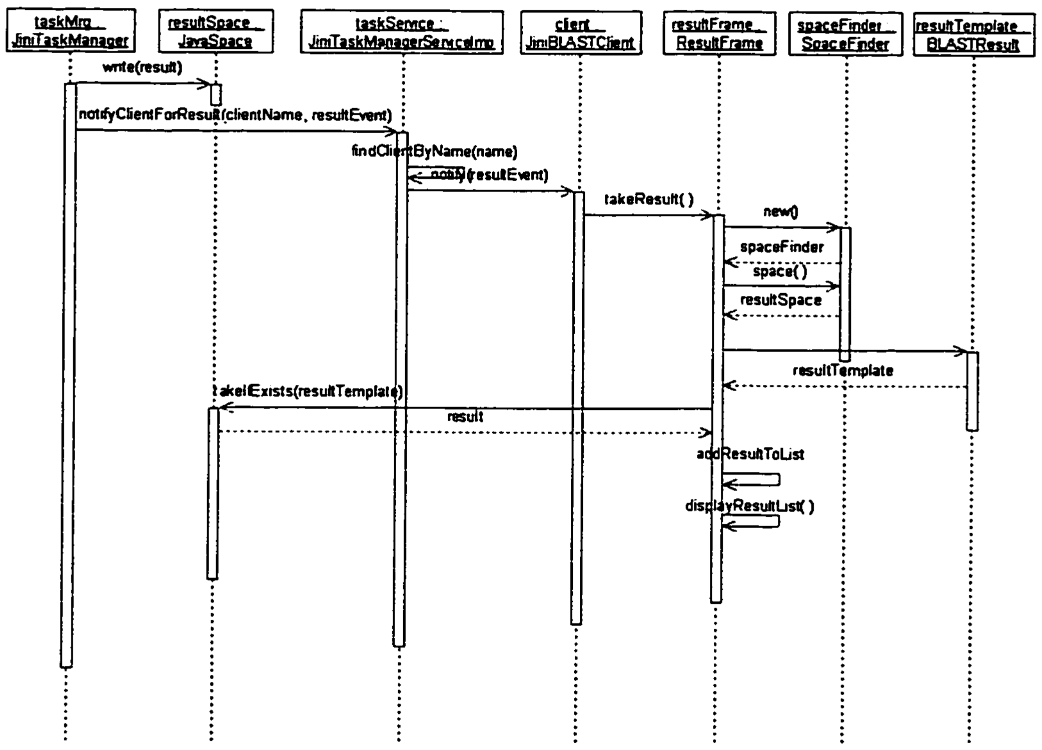
taskMgr : JiniTaskManager
resultSpace : JavaSpace
taskService : JiniTaskManagerServiceImp
client : JiniBLASTClient
resultFrame : ResultFrame
spaceFinder : SpaceFinder
resultTemplate : BLASTResult
write(result)
notifyClientForResult(clientName, resultEvent)
findClientByName(name)
notify(resultEvent)
takeResult( )
new()
spaceFinder
space( )
resultSpace
resultTemplate
takeIfExists(resultTemplate)
result
addResultToList
displayResultList( )

Figure 47: Sequence Diagram of a Client Reading a Result from ResultSpace

in the client, if there is a `RemoteException` raised during the sending of request, an `ErrorMessage` object with the failed Task Manager name is created and written into the ErrorSpace (see Fig 49). As shown in Fig 38 and Fig 50, a `FailureManager` thread in each TaskManager keeps reading `ErrorMessage` objects from that space and invokes an error handling procedure, such as sending all affected `BLASTTasks` again.

```
package blastbasic;

abstract public class JiniNode implements DiscoveryListener,
LeaseListener, RemoteEventListener{
.
.
.
 public void writeErrorMsgToSpace(ErrorMessage argMsg){
  //add code here to write ErrorMessage for Java Space
  SpaceFinder spaceFinder = new SpaceFinder("BLASTErrorMsg");
  JavaSpace errorSpace = spaceFinder.space();
  try{
   errorSpace.write(argMsg, null, Long.MAX_VALUE);
  }catch(RemoteException exp){
   //do something to handle error
  }catch(net.jini.core.transaction.TransactionException exp){
   //do something to handle error
  }
 }
}
```

Figure 48: *wroteErrorMsgToSpace()* in `JiniNode`.

```
package blastclient;

public class JiniBLASTClient extends JiniNode{
.
.
.
 //argUserInput in XML format
 public void sendRequest(String argUserInput, String argTaskServiceName)
 {
  BLASTRequest newRequest = new BLASTRequest();
  newRequest.setData(argUserInput);
  JiniTaskManagerService tmpService = findService(argUserInput, argTaskServiceName);
  //add this as remotelistener to service
  try{
   tmpService.addRemoteListener(this);
   tmpService.receiveBLASTRequest(newRequest);
  }catch(java.util.TooManyListenersException exp){
     //add error handling
  }catch(RemoteException exp){
     //create system ErrorMessage
   ErrorMessage newError = new ErrorMessage();
   newError.intType = ErrorMessage.EVENTTYPE_TASKMANAGERFAILED;
   newError.strNodeName = argTaskServiceName;
   newError.strMessageDesc = "some description";
   writeErrorMsgToSpace(newError);
  }
 }
}
```

Figure 49: `JiniBLASTClient` Creates an Instance of `ErrorMessage` and Writes it to the Error Space When an Exception Occurs.

```
package taskmanager;

public class FailureManager extends Thread{
  private JiniTaskManager parent = null;
  private JavaSpace errorMsgSpace = null; //JavaSpace we used for results deposit
  private final long MAX_SPACETIMEOUT = 10*60*1000;
  public void run(){
     while(true){
      //read error from errorspace and do something
      handleError(readErrorFromSpace());
      try{
        sleep(2*1000);
      }catch(InterruptedException exp){

}
     }
  }
  private ErrorMessage readErrorFromSpace(){
    //take all one errorMessage from space
    //create errorMessage template for matching
   ErrorMessage errorTemplate = new ErrorMessage();
   ErrorMessage error = null;
   try{
    error = (ErrorMessage)errorMsgSpace.takeIfExists(errorTemplate, null, MAX_SPACETIMEOUT);
   }catch(RemoteException exp){
    //add code to display exp message
   }
   return error;
  }
  public void handleError(ErrorMessage argError){
   switch(argError.intType){
    case BaseEventMessage.EVENTTYPE_SERVERFAILED:
        //add code here to handle error
        parent.writeErrorMsgToSpace(argError);
        break;
    case BaseEventMessage.EVENTTYPE_TASKMANAGERFAILED:
        //add code here to handle error
        break;
   }
  }
}
```

Figure 50: `FailureManager` Thread Continually Checks for the Potential Failure of the System Node.

Chapter 4

Conclusion and Future Work

4.1 System Requirements and Project Status

In this section we will review our system requirements and what we have achieved.

4.1.1 System Requirement and Our Design

In this section we list our system requirements and verify if our design satisfies these requirements. Our system is a workflow management system with the following features:

- High throughput.
- Dynamic configuration.
- Robustness.
- Heterogeneous platforms.

In the current design, we use multiple BLAST Servers to get high performance. By applying Jini technology, our system can dynamically configure itself and self heal. The Jini leasing protocol, multiple Task Managers and cross network shared error space (an instance of JavaSpaces) improves the failure tolerance of our system.

Based on Java and JNI technologies, BLAST Server and the workflow management system can run on different OS platforms.

4.1.2 Project Status

Currently the system design and most of the detail design for workflow management related module are completed. Most of the workflow related Java packages and classes are implemented and compiled using JDK 1.3.1 in JBuilder 4.0 environment. In order for our system to be compiled without error, we provide API for classes whose implementation is not certain at the time of this writing.

In order to make a working system, we still need to do the following design and implementation work:

- Graphic user interface design: in our current design, both client and administrator user interfaces are to be determined.
- C++ BLAST algorithm implementation: our system will invoke the BLAST algorithm written in C++.
- Implementation for some uncertain classes: some classes and functions, in our current design, are undetermined. For instance, what type of error handling do we want to do when a Task Manager is crashed.
- Testing and debugging: before we can release our working system, we also need to test and debug our system.

4.2 Conclusion

From this project, we found that the Jini and JavaSpaces technology is a good choice for implementing a distributed computing system in the domain of biological research.

From a workflow point of view, the Jini and JavaSpaces technology can split the software development into multiple groups. For instance, in our system one group can concentrate on BLAST search server component development, another can develop workflow management related software, and another group can focus on client GUI development. As Schaefer has shown in his paper [22], these three groups can work independently, with little coordination.

From a distributed computing point of view, the Jini and JavaSpaces technology provides an easy way to dynamically distribute a task and maintain a reasonable workload balance; to exchange data and code over network wires; to decentralize the failure handling; to mix servers with varying implementations and platforms in a single system. In fact, the capability of exchanging objects over the network makes Jini an ideal infrastructure for grid computing on a global scale [25, 26].

4.3 Future Work

This project is an attempt at testing and demonstrating the possibilities and advantages of using Jini to implement a high throughput BLAST cluster server. Obviously, the next step is implementing this system. There are many ways to improve on our system or make our system more general, both in biology computing and distributed computing domains.

4.3.1 System Implementation and Integration

The whole project is partitioned into several parts:

- The BLAST engine design and implementation;
- The workflow management system design and implementation;
- Client graphic user interface design and implementation;
- System integration and testing.

In order to execute the BLAST engine efficiently, we will implement it using C++. One of Dr Butlers graduate students, Jerry Jing just began to design and implement the BLAST engine using standard template libraries in C++. In this thesis, we discuss the design and partial implementation of the workflow management system. In the next step, we will implement a client GUI. After all the implementation work is completed, we can integrate the whole system together and provide a working system for testing and evaluation.

4.3.2 Load Balancing

This thesis presents a system architecture for high performance BLAST servers. Since our system contains multiple BLAST Servers and multiple Task Managers, the load balancing and job management are a critical issue for the final system performance. In the current design, although the clients can select their preferred Task Manager or BLAST Server, most requests will be sent randomly to any one of the Task Managers. The task manager dispatches the tasks by matching the database partition and BLAST algorithm on the severs. In future designs, we will add more heuristic algorithms to send requests to the Task Manager with the smallest job queues or the Task Manager with the shortest time to finish all its queued jobs. The Task Manager will use heuristic algorithms to send tasks to the BLAST Server with the fewest pending jobs. These algorithms need to dynamically detect the computing resources on the server, update server information on the Jini Lookup Service and notify the Jini client with service proxies downloaded.

4.3.3 Job Parallelism

In our current design, we only use data parallelism to split BLAST search tasks. This means that a subtask is based on database partitions. In order to improve performance, we may add a Job Parallelism algorithm in our system in such a way that each subtask will contain only one database partition and one query sequence. For instance, if we have 10 database partitions and the user input contains 10 sequences, the final number of subtasks will be 100. By this way, each server only does a small part of the whole task [13]. However, we must avoid creating too much overhead by splitting tasks into excessively small subtasks.

4.3.4 Job Priority

In our current design, all tasks will be added to the job queue and executed in a "first-in first-out" manner. In future versions, we may prefer that more important jobs are finished as fast as possible. Thus we can add job priority to the task definition, and allow the administrator to change the task's priority value. The client may set desired priority values if given permission.

4.3.5 Jini Administrator Service Bean

In our current design, we use Jini services to talk with the administrator node. Although it works, there is one problem: if we stop a service, we may end the lease of that service, disabling the administrator from seeing that service until it restarts. To resolve this problem, future design can add a small Jini service component called a Jini administrator service bean to each node. It will independently set up communication between a node and the administrator, and separate system Jini services from administration services. For example, on each BLAST server there are two Jini services, one is the Jini BLAST Service and another is the Jini Admin Service. The first one provides a BLAST search interface and the second one provides communication [3] for the administration.

4.3.6 Framework for High Performance Computing Workflow in Biology

Although our system is designed as a BLAST cluster, we found that it is easy to adapt to other high-performance computing needs by modifying the definition of requests, tasks and results, server implementations as well as client GUI. Thus, future work can involve exploring the possibilities of developing a Jini-based workflow framework for high performance computing in the field of biology.

Bibliography

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers and D.J. Lipman, Basic local alignment search tool. *J. of Mol. Biol*, **215**(October 1990) 403-410.

[2] J Newmarch, "A Programmer's Guide to Jini Technology", Apress, Berkeley 2000.

[3] M. Fahrmair, C. Salzmann and M. Schoenmakers, Carp@ - Managing Dynamic Jini Systems, in "Proc. OOPSLA Workshop on Object Oriented Reflection and Software Engineering", pp. 209-227, Springer, Denver, CO, 1999.

[4] N. Goodman, "Farming for the New Biology", TurboGenomics Inc. Tech.Rep. Press010101, TurboGenomics Inc. New Haven, CT, 2001.

[5] T.L. Madden, R.L. Tatusov and J. Zhang, "Applications of Network BLAST server" Meth. Enzymol., **266**:131-141, 1996.

[6] J. Zhang and T.L. Madden, "PowerBLAST: A new network BLAST application for interactive or automated sequence analysis and annotation." Genome Res. **7**:649-656, 1997.

[7] Object Management Group, "Workflow Facility RFP#2", OMG Document bom/98-06-07, Needham, MA, 1998.

[8] W.K. Edwards, "Core Jini (2nd Edition)", Prentice-Hall, 2000.

[9] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo and Ann Wollrath, "The Jini Specification", Addison-Wesley 1999.

[10] R Flenner, "Jini and JavaSpaces Application Development", Sams 2001.

[11] W. Gish, D.J. States, "Identification of Protein Coding Regions by Database Similarity Search", *Nature Genet*, **3**:266-272 1993.

[12] M. Migliardi, S. Schubiger and V. Sunderam, "A Distributed JAVASPACE Implementation for HARNESS", *Journal of Parallel and Distributed Computing*, **60**:1325-1340, doi:10.1006jpdc.2000.1656, 2000.

[13] R.D.Bjornson, A.H.Sherman, S.B.Weston, N.Willard and J.Wing. "TurboBLAST: A Parallel Implementation of BLAST based on the TurboHub Process Integration Architecture.". TurboGenomics Inc. Tech.Rep. TurboBLAST, TurboGenomics Inc. New Haven, CT, 2002.

[14] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller and D.J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", *Nucleic Acids Research*, **25**:3389-3402, 1997.

[15] Lloyd Allison, "Dynamic Programming Algorithm for Sequence Alignment." Online tutorial.
http://www.csse.monash.edu.au/~lloyd/tildeStrings/Notes/DPA.html.

[16] R.R. Shinghal, Natural Language Processing: syntax and semantic. In "Formal Concepts in Artificial Intelligence",Chapman & Hall Computing Series, London, 1992.

[17] Pittsburgh Supercomputing Center. "A Tutorial on Searching Sequence Databases and Sequence Scoring Methods", Online Tutorial, http://www.psc.edu/biomed/training/tutorials/sequence/db/index.html

[18] BlackStone Computing, "PowerCloud BLAST: Improve Turnaround Time of BLAST Analysis Through the Appropriate Segmentation of BLAST Datasets", BlackStone Computing Tech. Rep, powerCloudBLAST.pdf, 2002.

[19] J.T. Inman, H.R. Flores, G.D. May, J.W. Weller and C.J. Bell, "A High-Throughput Distributed DNA Sequence Analysis and Database System". *IBM Systems Journal*, **40**:464-486, 2001.

[20] T. Suzumura, S. Matsuoka and H. Nakada, "Design and Implementation of A Jini-based Computing Portal System", Summer United Workshops on Parallel, Distributed and Cooperative Processing, 2001.

[21] M. Migliardi, S. Schubiger and V. Sunderam, "A distributed JavaSpace implementation for HARNESS", *Journal of Parallel and Distributed Computing*, **60**, 1325-1340, 2000.

[22] L. Schaefer, G Graf and T. Lindquist, "Three-Segment Agent Architecture using JINI for Workflow", in Proc. 2001 Intl. Conf. on Artificial Intelligence, Las Vegas, NV, 2001.

[23] Sheng Liang, "Java Native Interface:Programmer's Guide and Specification", Addison-Wesley, 1999.

[24] B. King, Introduction to AGAVE Genomic Annotation XML, in "Abstract for Bioinformatics Open Source Conference (BOSC)", San Diego, CA, 2001.

[25] Benjamin Clifford, "JiniGrid: Specification and Implementation of a Task Farm Service for Jini", in Edinburgh Parallel Computer Centre SSP Report, EPCC-SS-2000-02, September 2000.

[26] Z. Juhasz, A. Andics and S. Pota, "JM: A Jini Framework for Global Computing", in Proc 2nd International Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems at IEEE International Symposium on Cluster Computing and the Grid (CCGrid'2002), 395-400, Berlin, Germany 2002.

[27] Liusong Yang, A design of a BLAST server with Jini technology, Concordia Theses, Research Papers, Montreal Canada April 2002

Appendix A

BLAST Databases

A.1 General Database

- **nr**

  All GenBank+EMBL+DDBJ+PDB sequences (but no EST, STS, GSS, or phase 0, 1 or 2 HTGS sequences). No longer "non-redundant".

- **est**

  Database of GenBank+EMBL+DDBJ sequences from EST Divisions.

- **est_human**

  Human subset of GenBank+EMBL+DDBJ sequences from EST Divisions.

- **est_mouse**

  Mouse subset of GenBank+EMBL+DDBJ sequences from EST Divisions.

- **est_others**

  Non-Mouse, non-Human sequences of GenBank+EMBL+DDBJ sequences from EST Divisions.

- **gss**

  Genome Survey Sequence, including single-pass genomic data, exon-trapped sequences, and Alu PCR sequences.

- **htgs**

  Unfinished High Throughput Genomic Sequences: phases 0, 1 and 2 (finished, phase 3 HTG sequences are in nr).

- **pat**

  Nucleotides from the Patent division of GenBank.

- **yeast**

  Yeast (Saccharomyces cerevisiae) genomic nucleotide sequences.

- **mito**

  Database of mitochondrial sequences.

- **vector**

  Vector subset of GenBank(R), NCBI, in ftp://ftp.ncbi.nih.gov/blast/db/.

- **E. coli**

  Escherichia coli genomic nucleotide sequences.

- **pdb**

  Sequences derived from the 3-dimensional structure from Brookhaven Protein Data Bank.

- **Drosophila genome**

  Drosophila genome provided by Celera and Berkeley Drosophila Genome Project (BDGP).

- **month**

  All new or revised GenBank+EMBL+DDBJ+PDB sequences released in the last 30 days.

- **alu**

  Select Alu repeats from REPBASE, suitable for masking Alu repeats from query sequences. It is available by anonymous FTP from ftp.ncbi.nih.gov (under the

/pub/jmc/alu directory). See "Alu alert" by Claverie and Makalowski, Nature vol. 371, page 752 (1994).

- **dbsts**

  Database of GenBank+EMBL+DDBJ sequences from STS Divisions .

- **chromosome**

  Searches Complete Genomes, Complete Chromosome, or contigs from the NCBI Reference Sequence project.

A.2 Human Genome Blast Databases

- **genome**

  human genomic contig sequences with NT_#### accessions.

- **mrna**

  human RefSeq mrna with NM_#### or XM_#### accessions

- **protein**

  human RefSeq proteins with NP_#### or XP_#### accessions

- **gscan mrna**

  predicted mRNA sequences generated by running GenomeScan program on human genomic contigs.

- **gscan protein**

  CDS translations from GenomeScan mRNA set.

Appendix B

BLAST Search Algorithms

- **blastn**

  Standard nucleotide BLAST.

- **blastp**

  Standard protein-protein BLAST.

- **blastx**

  Nucleotide query against protein db BLAST.

- **tblastn**

  Protein query against translated db.

- **tblastx**

  Nucleotide query against translated db.

- **PSI-BLAST**

  One of blastp algorithms, means position specific iterative BLAST a feature of BLAST 2.0 in which a position specific scoring matrix, PSSM is constructed (automatically) from a multiple alignment of the highest scoring hits in an initial BLAST search. For more information please visit http://www.ncbi.nlm.nih.gov/Education/BLASTinfo/psi1.html.

- **MEGABLAST**

  It uses greedy algorithm for nucleotide sequence alignment search. It is optimized for aligning sequences that differ slightly as a result of sequencing or other similar "errors". When larger word size is used, it is up to 10 times faster than more common sequence similarity programs.
  http://www.ncbi.nlm.nih.gov/blast/megablast.html.