BTech 451
Project in Information Technology

Mid-year Report

Yijing Wei

ID: 5284612

UPI: ywei110

June 2013

Abstract

This report summarizes the work that I have done for my Btech 451 project at Kiwiplan in both Semester One and Semeter Two.

There are nine chapters in this report. First I will give an introduction to the project, what the problem is and the goal of this project as well as the background of the company. Then the current Distributed Service Profiler will be introduced, what its current functions are and the design of its updated version.

My main research includes the research on Hibernate, which will be discussed in Chapter 4 Hibernate Research. Chapter 4 gives a detailed discussion on object relational mapping, Hibernate architecture, class objects, sessions, properties, persistent class, the advantages of Hibernate, what databases are supported by Hibernate, Hibernate Query Language (HQL) as well as Hibernate annotations.

Apart from the research on Hibernate, I also did some research into debugging strategies, including bottom-up debugging, traces and controlled execution. This is to help understand the features of distributed systems.

Chapter 6 is the research on monitoring tools for distributed systems. It gives an overview of distributed computing, object oriented distributed computing, the issues that need to addressed when monitoring distributed systems and the features that a good monitoring system should have.

Apart from the research on Hibernate, I also did some research into profiling tools, Hibernate Interceptor and the Java EmptyInterceptor interface.

I created a Hibernate testing application to deepen my understanding of Hibernate interceptor, which will be used to interceptor queries made to the database in the Distributed Service Profiler. The code of my testing application is available in the appendix at the end of this report. I also created a LoggerInterceptor and a LoggingManager for the Distributed Service Profiler.

Future work includes integrating the LoggerInterceptor with the Distributed Service Profiler, extracting argument bytes and the return bytes from queries, matching queries with their callers, creating and displaying query nodes and researching on profilers for distributed systems. Details will be provided in Chapter 9 Future Work.

Difficulties include time management problems and the lack of academic knowledge that I encountered in Semester One will also be included in this report, in Chapter 10.

Contents

Chapter 1
Introduction

The BTech (Information Technology) is a four-year Honours degree, with selection of courses mainly from Computer Science and Information Systems. BTech 451 is a whole year project, compulsory for BTech final year students. It carries 45 points, the weight of two courses. Students should guarantee 8-10 hours' work every week. This report describes the details of this project and the work that I have done so far for my BTech project with Kiwiplan. It only covers the progress made in Semester One. Progress made in Semester Two will be discussed in the final year report.

1.1 The Goal

My BTech project is carried out with Kiwiplan, which requires me to go to the company and work in their office every week. It is a database-based project implemented in Java. The aim of this project is to extend the current Distributed Service Profiler, which is used by the developers at Kiwiplan, to integrate with the database to intercept queries made to the database.

1.2 The Problem

Most services developed by Kiwiplan need to communicate with each other and with the database. The Distributed Service Profiler was produced several years ago by a BTech student to measure and display the information, such as the time taken for the method to process, the argument bytes and the return bytes of the method call, on the calls between various services, and it is used by the developers at Kiwiplan to analyse the performance of their code and to find bottlenecks. The current Distributed Service Profiler only displays the information of the calls and communication between various services. The information of the queries made to the database is not specified, even though it is included in the information on the service call. However, it is important to know the information of the queries, such as the time taken for a query to process and the bytes returned by it, because sometimes it is the queries that take the longest time and needs to be improved. Currently, the developers at Kiwiplan have to manually go through the logs to check all the queries, which is an exhausting process. Therefore, my role is to implement an extension for the Distributed Service Profiler to integrate with the database layer, and make it more efficient for performance analysis.

1.3The Company

Kiwiplan is a software company specializing in corrugating and packaging industry with over 30 years' history, more than 600 customer locations and over 160 employees. It provides industry-specific products, such as Material Management System (MMS), Supply Chain Simulator (SCS), Truck Scheduling System (TSS), and Machine Data Collection (MDC), to corrugated, folding carton, rigid and flexible packaging and other manufacturers in more than 36 countries world-wide. Kiwiplan also provides consulting services to ensure its customers have the skills to utilize Kiwiplan tools effectively.

Kiwiplan began developing software solutions to meet the needs of the corrugated industry in 1981 in Auckland, New Zealand, and it continues to be the leading software company in corrugating and packaging industry.
The branch that I have been working at is Kiwiplan NZ, located in East Tamaki, Auckland [1].

Chapter 2
Distributed Service Profiler

This chapter gives an overview of Distributed Service Profiler, its functions, its interface and how it works.

2.1 Introduction to Distributed Service Profiler

The Distributed Service Profiler is a profiler tool developed by a BTech student a few years ago. It has been modified and updated by BTech students as well as the developers at Kiwiplan ever since it was created. The Distributed Service Profiler is used by the developers at Kiwiplan to analyse the performance of their code and find bottlenecks.

Since Kiwiplan is a company which provides more than one service to its clients, and most of the services provided by Kiwiplan need to communicate with other services. Examples of the services are Material Services, Quality Management Service and Manufacturing Service. In order to measure the performance between various services, Kiwiplan needs a profiler tool that can be connected to various services and measure the communication between the services, and that is the birth of Distributed Service Profiler.

2.2 The Interface

Figure 2.1 on Page 8 is a snapshot of the interface of Distributed Service Profiler, which was taken using the ‘Screenshot’ function of the profiler. There are two panels. On the left side is the connection panel. It displays the services connected through the Distributed Service Profiler and the port numbers they are connected to. Different services are distinguished by different colours. A user can define his own port number and colour for a service with the help of the ‘Add Connection’ button. ‘Clear Results’ will clear the current results displayed by the result panel on the left and will leave the result panel blank.

The left panel is the result panel. Each method call is represented as a node in the colour of its corresponding service and a method call tree is built. The root of the call tree is always the Client, i.e. the user using the profiler. Each parent node is the caller of its child nodes. The leaf nodes are typically the queries made to the database. The leaf nodes are usually the ones that are of interest to the developers at Kiwiplan, since

they want to know which specific method calls or queries are taking longer to process than the others so that improvements can be made. However, since no information is given on the actual database queries, the developers have to go through the database log files to find out which query is taking the longest time to process, which is an exhausting process. Therefore, we want to find a way to integrate the Distributed Service Profiler with the database layer to make it more efficient for the developers to analyse their code.

Each node displays the information on the method call, such as the method name, the time taken for the method to complete, the argument and return bytes of the method and the number of hits (the number of times the method was called. The result panel displays the results of the communication between various services. Figure 2.2 below illustrates an example of such communication between various services. The client (Blue Node) calls Quality Management Service (Green Node), which wants to know all the material types from the Material Service (Purple Node), which will then query the database to retrieve all the material types.

The Distributed Service Profile also comes with scrolling and zooming features.

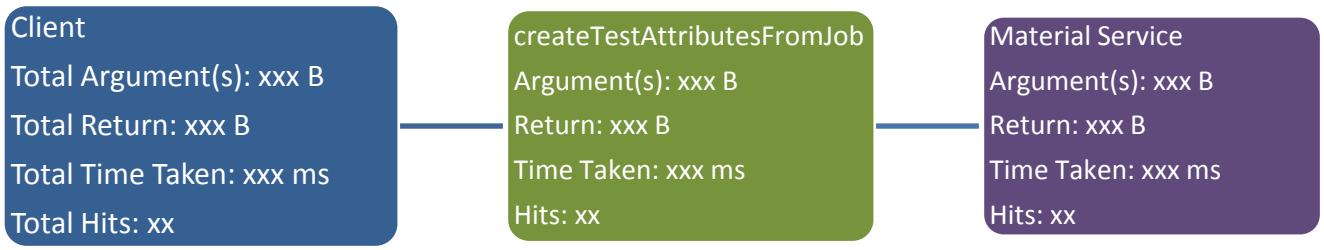


Figure 2.2 Illustration of the communication between services

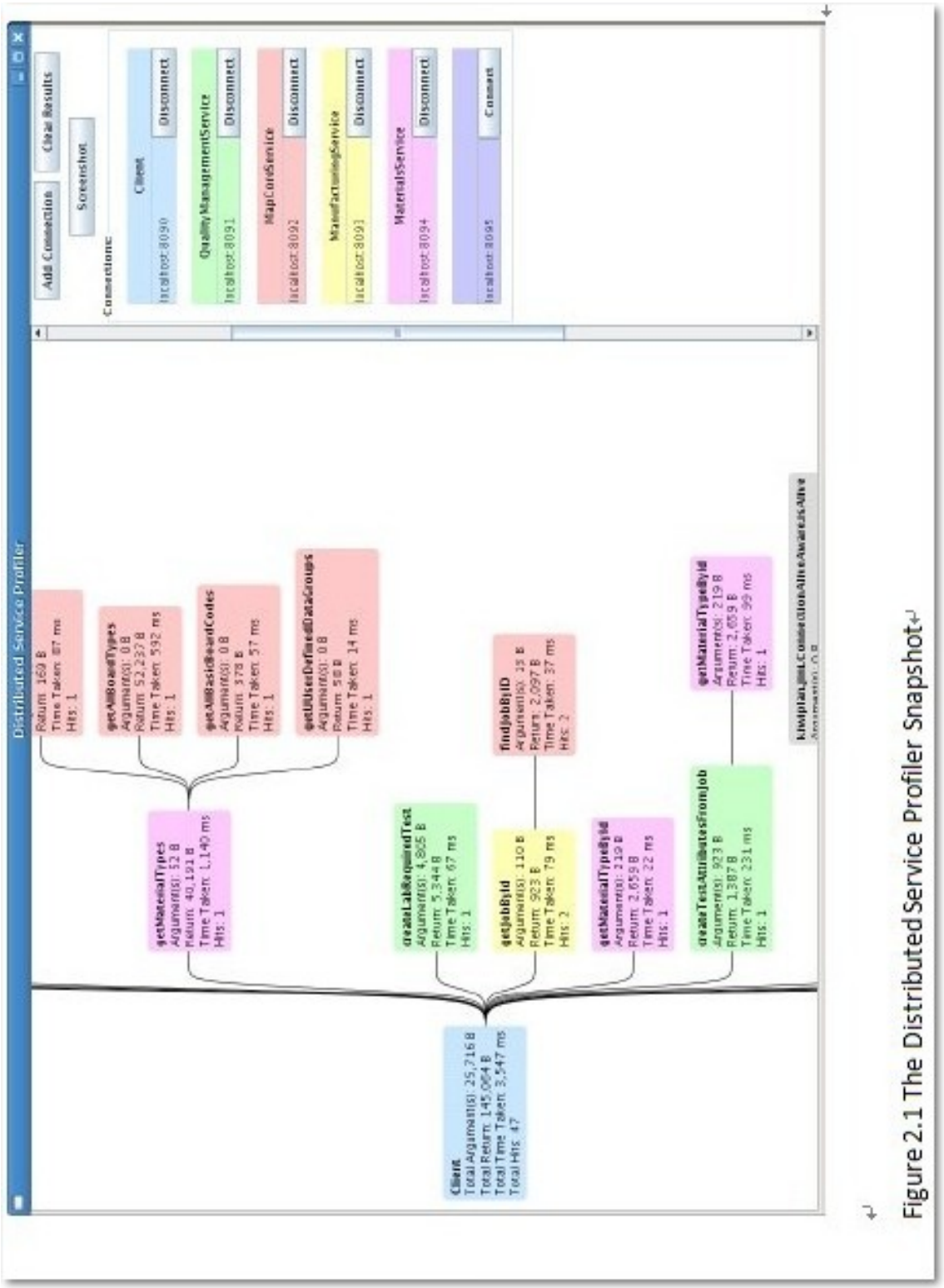


Figure 2.1 The Distributed Service Profiler Snapshot

2.3 The Design

I developed a mockup of the Distributed Service Profiler using Balsamiq, a mockup

tool used by many developers, along with my own design of how I want to add the database query node. The mock-up is shown in Figure 2.2 below.

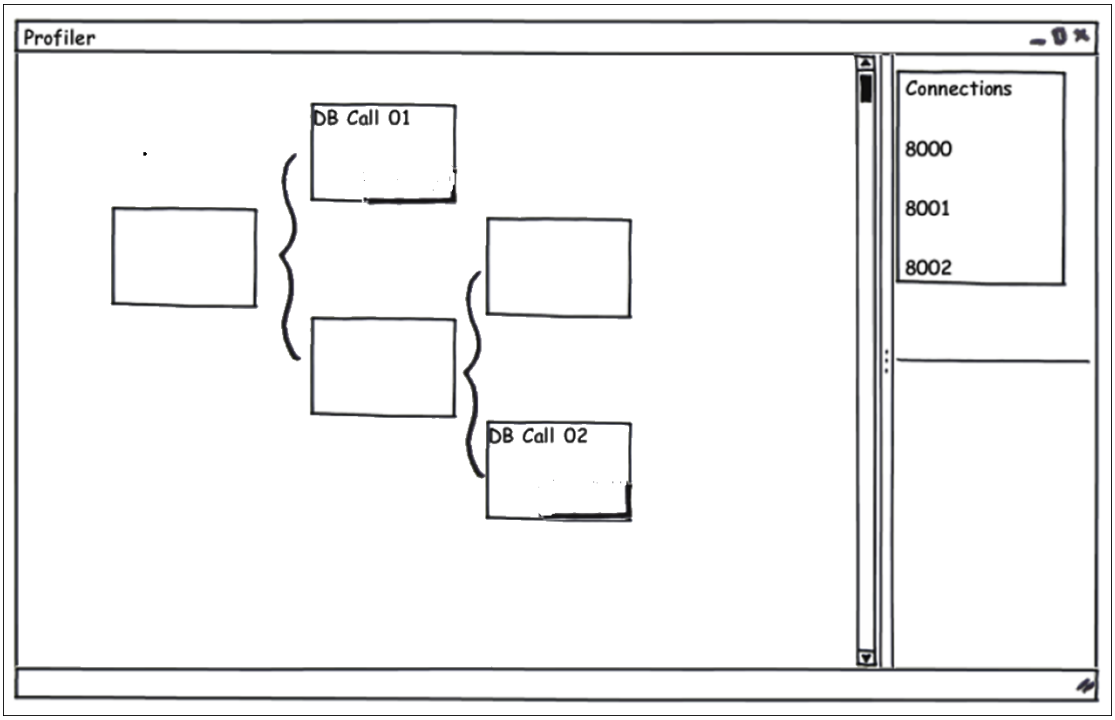


Figure 2.2 Mockup of the Distributed Service Profiler

The design of the addition of the query nodes will be consistent with the current Distributed Service Profiler. The connection panel on the right remains unchanged for now. Major changes will happen to the result panel on the left. Additional features may be added once the basic steps have been completed.

On the result panel on the left in Figure 2.2 above, the empty nodes are the normal method calls that already exist in the current Distributed Service Profiler. The nodes with ‘DB Call 01’ and ‘DB Call 02’ are the query nodes that will be added to the current Distributed Service Profiler. The query nodes will display the same information as the method call nodes, the argument size that is passed to the query, the return bytes returned by a query, time taken for the query to process and the number of hits. If we have a service method queries the database for information, then there will be a query node linked to it and displays corresponding information.

Chapter 3
Related Work

This chapter summarizes the research papers that I have studied and how they are related to my project. This chapter provides an overview of each paper. Bits and pieces of each paper will also be mentioned in my research chapters.

General problems associated with monitoring are outlined in [7]. This paper presents the architecture of a general purpose and an extensible distributed monitoring system, which focused on monitoring at the inter-process communication level. Three approaches to the display of process interactions are described including textual traces, animated graphical traces, and a combination of aspects of the textual and graphical approaches. This paper explains in details the difference between monitoring a distributed system and a sequential system, and what a good monitoring system should have, which provides me with a deeper understanding of the things to consider when developing a monitoring tool for distributed systems. Details will be discussed in Section 6.4 and 6.6.

Since my project focuses on monitoring distributed systems, I specifically studied distributed systems. [8] provides a good background of distributed systems. This paper introduces the history of distributed systems, several models and usages of distributed systems, the benefits that distributed systems have brought us, etc.. It is interesting that this paper uses human brains and natural phenomenon to illustrate distributed systems, which makes the concepts easy to understand. More details on distributed systems will be discussed in Section 6.1.

The issues involved in debugging a distributed computing system are discussed in [9]. This paper describes the major differences between debugging a distributed system and debugging a sequential system. Since all my previous assignments at university only involve sequential systems but the distributed service profiler is used in distributed systems, it is worth studying the difference between distributed systems and sequential systems. A methodology for distributed debugging is also presented in [9]. Although the distributed service profiler is used to monitor, not debug, the services, it is interesting to look at the differences between distributed and sequential systems from the perspective of debugging. Debugging a distributed computing system is in essence similar to the debugging of a sequential system. However, since a distributed system is a combination of several sequential systems, it is more difficult to debug a distributed system. This paper presents several debugging techniques and tools for distributed systems.

[10] proposes a framework to handle large distributed dynamic systems as a graph of interacting components. This paper provides a mathematical approach to monitoring

distributed systems and illustrates distributed systems as graphs. A distributed system is considered as a combination of elementary components, e.g. sequential systems. This combined system is asynchronous. Each component has its own clock and exchanges messages with its neighbours. This paper looks at monitoring distributed systems from the perspective of algorithms. It defines systems and distributed systems in terms of algorithms. The algebra in this paper is a little hard to understand due to my lack of mathematical background, however, the graphs which illustrate distributed systems provide me with a new mean to look at distributed systems.

[11] presents a method for monitoring distributed systems by applying concepts from the field of relational databases. This paper considers the Internet as a distributed system, which is different from my project. However, it does provide an original idea of monitoring distributed systems. The combination of unifield views of the data provided from the various systems and processing to extract the needed information is also straightforward. Relational query optimization techniques are used to reduce network traffic and efficiently produce the results. This approach views the output of several of the services as relational tables. The operations needed to combine the data from the services are operations used frequently for querying relational databases. Flexible agents are used for the realization of this approach. They only perform a specific type of tasks, namely the relational operation. The fact that relational databases can be applied to help monitoring distributed systems is what I think makes this paper worth reading.

Fundamental problems associated with monitoring distributed systems are also mentioned in [12]. One problem is that delays in transferring information makes it difficult to obtain a global and consistent view of all components in a distributed systems. More details of the problems of monitoring distributed systems will be discussed in Section 6.4. To overcome these problems, a monitoring system must provide a set of general functions for generating, processing, disseminating and presenting monitoring information. [12] presents a monitoring model in terms of a set of monitoring functions. The main functions include generation of monitored information, processing of monitored information to validate, dissemination of required information to the users and presentation of monitored information to users via flexible graphical facilities. This paper overall describes the problems associated with monitoring of distributed systems as well as the functions that a monitoring system should have, which has great value to me, since it helps me get a deeper understanding of the things that I should consider when monitoring distributed systems.

Chapter 4
Hibernate Research

This chapter describes the research that I have done for Hibernate. Hibernate is part of my main research, since it will be used to integrate the Distributed Service Profiler with the database.

4.1 Object Relational Mapping

JDBC stands for Java Database Connectivity and provides a set of Java API for accessing the relational databases from Java program. These Java APIs enables Java programs to execute SQL statements and interact with any SQL compliant databases.

JDBC provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification. The pros and cons of JDBC are listed below in Table 4.1.

Pros of JDBC	Cons of JDBC
Clean and simple SQL processing	Complex if it is used in large projects
Good performance with large data	Large programming overhead
Good for small applications	No encapsulation
Simple syntax, easy to learn	Hard to implement MVC model
	Query is DBMS specific

Table 4.1 Pros and cons of JDBC

When we work with an object-oriented system, there's a mismatch between the object model and the relational database. RDBMSs represent data in a tabular format whereas object-oriented languages, such as Java or C#, represent it as an interconnected graph of objects. The first problem is that, what if we need to modify the design of our database after having developed few pages or our application? Secondly, loading and storing objects in a relational database exposes us to the following mismatch problems, listed in Table 4.2.

Mismatch	Description
Granularity	Sometimes the object model has more classes than the number of corresponding tables in the database.
Inheritance	RDBMs do not define anything similar to inheritance, which is a natural paradigm in object-oriented programming languages.
Identity	A RDBM defines exactly one notion of sameness, i.e. the primary key. However, Java defines both object identity (e.g.: a==b) and

	object equality (e.g.: a.equals(b)).
Associations	Object-oriented languages represent associations using object references whereas RDBMs represent an association as a foreign key column.
Navigation	The ways objects are accessed in Java and in a RDBMS are fundamentally different.

Table 4.2 The mismatch problems exposed when objects in a relational database are loaded or stored

The Object Relational Mapping (ORM) handles all the mismatch problems mentioned in Table 4.2. ORMB is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C# etc. Compared to plain JDBC, an ORM system has following advantages:

1. ORM allows business code access rather than database tables.
2. ORM hides details of SQL queries from object oriented logic.
3. There is no need to deal with the database implementation.
4. Entities are based on business models rather than database structures.
5. ORM provides transaction management and automatic key generation.
6. Development of application is fast.

An ORM solution consists of the following entities:

1. An API to perform basic create, select, update and delete (CRUD) database operations on objects of persistent classes.
2. A language to specify queries that refer to classes and properties of classes.
3. A configurable facility for specifying mapping metadata.
4. A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.

There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database. The most popular frameworks are listed below:

- Enterprise JavaBeans Entity Beans
- Java Data Objects
- Castor
- TopLink
- Spring DAO
- Hibernate

4.2 Hibernate Introduction

Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an objected-oriented domain model to a traditional relational database [2].

The primary function provided by Hibernate is mapping Java data types to SQL data types, i.e. Java classes to database tables, which is accomplished through the configuration of mapping XML files. With the help of XML files, Hibernate can generate skeletal source code for the persistence class. It relieves the developers from 90% of data persistence related programming.

The position of Hibernate is between the traditional Java objects and database management system as shown in Figure 4.1 below.

Figure 4.1 The position of Hibernate

Hibernate is distributed under the GNU Lesser General Public License, free to download from the Internet. [3]

4.3 Hibernate Architecture

Hibernate isolates the Java application and the database so that you do not have to know the underlying APIs. Hibernate uses the database and the configuration file to provide persistence objects and services to the application. Figure 4.2 and Figure 4.3 below are a high level view of the architecture of Hibernate and a detailed view of the architecture of Hibernate respectively.

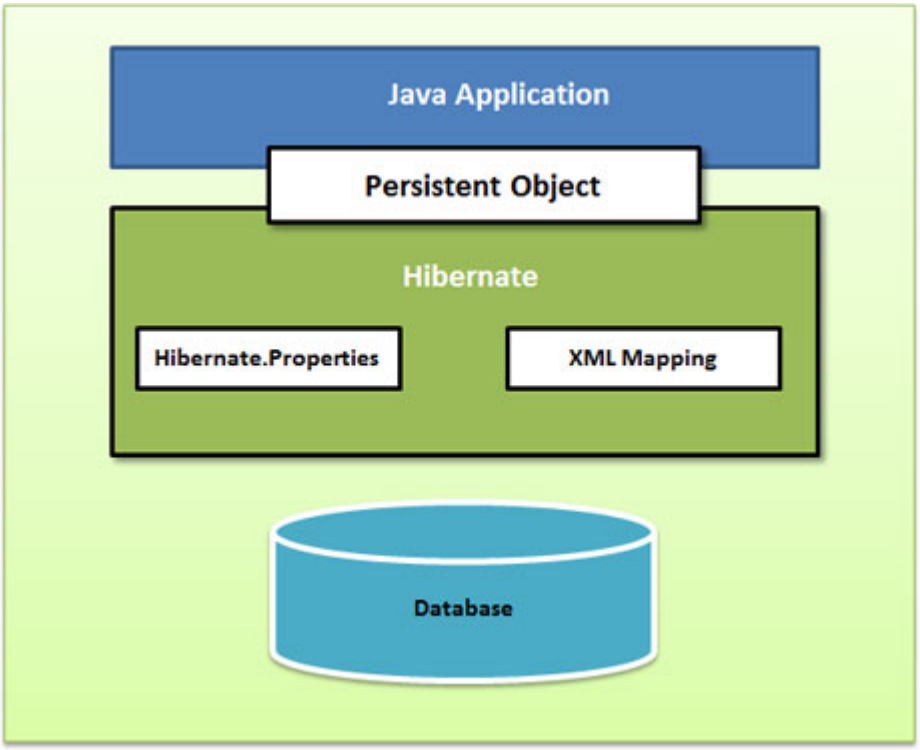


Figure 4.2 High level architecture of Hibernate

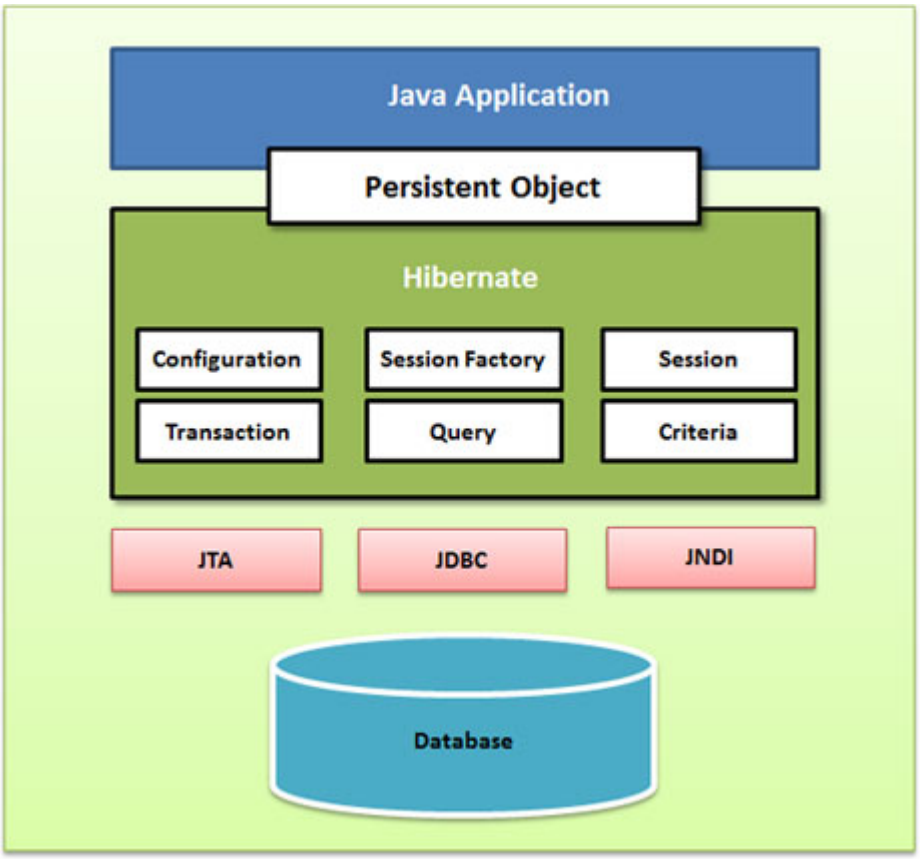


Figure 4.3 Detailed view of the architecture of Hibernate

In Figure 4.3, JTA (Java Transaction API), JDBC and JNDI (Java Naming and

Directory Interface) are the Java APIs that Hibernate uses. JDBC allows almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to integrate with J2EE application servers.

Configuration, session factory, session, transaction, query and criteria in Figure 4.3 are the Java objects used in Hibernate Application Architecture, and they will be explained in section 4.3 below.

4.4 Hibernate Class Objects

1. Configuration Object: The configuration object is the first object that you should create in a Hibernate application. It is created during the initialization of the application and is created only once. The configuration object is a configuration file for Hibernate, and it provides two key components: the database connection and the class mapping setup. Configuration files are usually either a standard Java properties file called hibernate.properties or an XML file named hibernate.cfg.xml.

2. SessionFactory Object: The SessionFactory object is a thread safe object and is used by all threads of the application. The SessionFactory object is a heavyweight object so usually it is created during the start up of an application and is kept for later use. Only one SessionFactory object is needed per database.

3. Session Object: Hibernate Session object will be discussed in details in section 4.4.

4. Transaction Object: A transaction is a unit of work with the database, and this functionality is supported by most RDBMS. The Transaction object is an optional object in Hibernate applications.

5. Query Object: Query objects use SQL or HQL (Hibernate Query Langueage) string to create, manipulate and retrieve objects from a database. A Query object is used to bind query parameters, limit the number of results returned by the query and execute query.

6. Criteria Object: Criteria objects are used to create and execute object oriented criteria queries to retrieve data from a database.

4.5 Hibernate Sessions

A Session object is used to establish a physical connection with the database. Unlike a SessionFactory object, a Session object is lightweighted and is instantiated each time an iteraction is needed with the database. The Session object should not be kept open for a long time since it is not thread safe. A Session object should only be created and destroyed as needed.

The main function of a Session object is to support database operations, such as create, select, update and delete for instances of mapped classes. At any point in time, instances may exist in one of the following states:

1. Transient: An instance is transient if it has not been associated with a Session, and has no representation in the database.

2. Persistent: A transient instance can be made into a persistent instance by associating it with a Session. A persistent instance will have a representation in the database.

3. Detached: A persistent instance becomes detached once we close the Hibernate Session.

4.6 Hibernate Properties

To configure Hibernate to work with a database, the following is a list of important Hibernate properties that need to be specified:

Property Name	**Description**
hibernate.dialect	Ensures that Hibernate generates appropriate SQL string for the database.
hibernate.connection.driver_class	Specifies the JDBC driver class.
hibernate.connection.url	Specifies the JDBC URL to the database.
hibernate.connection.username	Specifies the username of the database.
hibernate.connection.password	Specifies the password of the database.
hibernate.connection.pool_size	Limits the number of connections that can be waiting in the Hibernate database connection pool.
hibernate.connection.autocommit	Allows autocommit mode for the JDBC connection.

Table 4.1 Hibernate Properties #1

For a database with an application server and JNDI, the following properties need to be configured:

Property Name	Description
hibernate.connection.datasource	Specifies the JNDI name defined in the application server context you are using for the application.
hibernate.jndi.class	Specifies the InitialContext class for JNDI.
hibernate.jndi.<JNDIpropertyname>	Passes the JNDI properties to the JNDI InitialContext.
hibernate.jndi.url	Specifies the URL for JNDI.
hibernate.connection.username	Specifies the username of the database.
hibernate.connection.password	Specifies the password of the database.

Table 4.2 Hibernate Properties #2

4.7 Hibernate Persistent Class

Hibernate persistent classes are the Java object classes whose instances will be stored in the database tables. Hibernate works best if these classes follow some simple rules, known as the Plan Old Java Object (POJO) programming model. The POJO name is used to emphasize that a given object is an ordinary Java object, not a special object, and in particular not an Enterprise JavaBean. The main rules of POJO programming model are listed below, however, no rules are hard requirements.

1. All persistent Java classes must have a default constructor.
2. All persistent Java classes should contain an ID in order to allow easy identification of your objects within Hibernate and the database. ID maps to the primary key column of a database table
3. All persistent attributes in a persistent Java class should be declared private and have setXXX() mutators and getXXX() accessors defined.

4.8 The Advantages of Hibernate

All the main advantages of Hibernate listed below have reassured that I should use Hibernate to integrate the Distributed Service Profiler with the database.

1. Hibernate maps Java classes to database tables using XML files without any explicit code.
2. Hibernate provides simple APIs for inserting and accessing Java objects directly to and from the database.
3. Only XML file needs to be modified to reflect any changes in the database.
4. Hibernate allows us to work with our familiar Java object types instead of

unfamiliar SQL types.

5. No application server is required for Hibernate.
6. Hibernate is able to handle complex relationships between objects in the database.
7. Hibernate implements smart fetching strategies to minimize database access and thus more efficient.
8. Hibernate provides simple querying of data.

4. 9 Databases supported by Hibernate

Databases supported by Hibernate are also included in my research to ensure that the database used at Kiwiplan is supported by Hibernate. Hibernate supports most of the major RDBMS. Some common RDBMS supported by Hibernate are listed below:

- HSQL Database Engine
- DB2/NT
- MySQL – This is the RDBMS used by Kiwiplan
- Oracle
- Microsoft SQL Server Database
- Sybase SQL Server
- FrontBase
- PostgreSQL
- Informix Dynamic Server

4.10 Hibernate Query Language (HQL)

Similar to SQL, Hibernate Query Langue (HQL) is an object-oriented query language. SQL carries out database operations on tables and columns, whereas HQL works with persistent objects and their properties. Hibernate translates HQL queries into conventional SQL queries, which in turns perform operation on the database.

SQL queries can be used directly with Hibernate using Native SQL, however, it may cause database portability hassles. Therefore, it is recommended that we use HQL statements to avoid that problem and to take advantage of the SQL generation and caching strategies of Hibernate.

Like SQL, in HQL keywords such as SELECT, FROM and WHERE are not case sensitive, but properties like table names and column names are case sensitive.

The reason why I researched into HQL queries is to see how Hibernate carries out

database operations, what differences does it have from SQL queries.

4.11 Hibernate Annotations

Hibernate annotations is the newest way to define mappings without a use of XML file. Annotations can be used in addition to or as a replacement of XML mapping metadata. Hibernate annotations is the powerful way to provide the metadata for the object and relational table mapping. All the metadata is clubbed into the POJO java file along with the code, which helps the user to understand the table structure and POJO simultaneously during the development.

In order to make the application portable to other EJB 3 compliant ORM applications, annotations must be used to represent the mapping information. XML-based mappings provide greater flexibility. In my implementation, I will use XML-based mappings instead of annotations.

With Hibernate Annotation, all the metadata is clubbed into the POJO Java file along with the code, which helps the user to understand the table structure and POJO simultaneously during the development. An example EMPLOYEE class with annotations is given below to illustrate different annotations.

Employee.java

```
import javax.persistence.*;

@Entity
@Table(name = "EMPLOYEE")

public class Employee {
    @Id @GeneratedValue
    @Column(name = "id")
    private int id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "salary")
    private int salary;
```

```
    public Employee() {}
    public int getId() {
        return id;
    }

    public void setId( int id ) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary( int salary ) {
        this.salary = salary;
    }
}
```

Hibernate detects that the @Id annotation is on a field and assumes that it should access properties on an object directly through fields at runtime. Placing @Id annotation on the getId() method would enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy.

@Entity annotation is used to mark this Employee class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

@Table annotation allows the programmer to specify the details of the table that will

be used to persist the entity in the database. The @Table annotation provides four attributes to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table.

Each entity bean will have a primary key, which we have annotated on the class with the @Id annotation. The primary key can be a single field or a combination of multiple fields depending on the table structure. By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but we can override this by applying the @GeneratedValue annotation which takes two parameters strategy and generator, but I will only use the default key generation strategy in my project. Letting Hibernate determine which generator type to use makes t code portable between different databases.

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. @Column annotation can be used with the following most commonly used attributes as listed in Table 3.5:

Name of the attribute	Function
name	name attribute permits the name of the column to be explicitly specified
length	length attribute permits the size of the column used to map a value particularly for a String value
nullable	nullable attribute permits the column to be marked NOT NULL when the schema is generated
unique	unique attribute permits the column to be marked as containing only unique values

Table 4.5 Attributes that can be used with @Column

Chapter 5
Debugging Strategies

In this chapter we are going to discuss the debugging strategies including bottom-up debugging and traces debugging.

5.1 Bottom-Up Debugging

In order to test and debug a single process, we do not really need distributed debugging tools. The standard sequential program debugging tools is usually adequate. This is true even if a few auxiliary test processes will probably run on a single computer eliminating the problem of multiple processors and communication delays, which will be discussed in section 6.2. The auxiliary processes should be simple to make it reasonable to assume that they operate correctly.

Typically, a single process is tested and debugged as follows. The programmer starts up the auxiliary process. Then the main process with an interactive symbolic debugger compiled in is run. The programmer, through the interactive debugger, controls the execution of the main process. The programmer can set break points and examine the process variables in order to discover any bugs. After processes are individually tested, they can be combined and tested using the distributed monitoring tools. After having debugged processes individually, most of the bugs that remain will be the ones resulting from the interaction between the processes. The distributed monitoring tools are intended for locating these bugs. However, the distributed monitoring tools do not replace the sequential program tools. Both sequential and distributed tools should be used in union for debugging computing systems.

In a distributed program, the road to a bug free system is more hazardous than in a sequential program. Thus, the programmer must proceed cautiously and slowly. The programmer must be conservative. One way to be conservative is to use bottom-up debugging.

In bottom-up debugging, each program module is tested separately in special test surroundings. Later, the modules are put together and tested as a whole. In top-down debugging, the entire program is always tested as a whole in nearly the final form. The test program may have extra output statements and some sections may be replaced by dummy sections.

Each of these techniques has its own set of advantages and disadvantages. When debugging a sequential program, a top-down approach can eliminate the need for

artificial test surroundings and thus save programming effort. However, in a distributed system the effort required to find a bug can be too much to create the artificial test surroundings. In a distributed computing system, it is important to test and debug the interfaces among the processes as early as possible. Due to the usual complexity of distributed systems, finding the process interface bugs will be difficult unless the processes being checked are working properly.

5.2 Traces

The distributed debugging tools are helpful in finding out the bugs which result from the interaction of the various application processes. In many cases there bugs only occur when the processes exchange data or synchronize their activity. Since the observed results are hard to reproduce, tracing will play an important role in distributed system debugging.

A process trace records the history of the process. An entry is made on the trace for each important event that occurs in the life of the process. For example, the fact that a variable is changing values can be recorded in an entry. The branch taken in an if statement will also be recorded. If each process in the distributed computing system keeps a trace, then the programmer has a written record describing what occurred in the system. This record can be examined by the programmer after a bug is observed in order to discover what caused the bug.

Tracing process execution has a serious drawback – keeping the trace is time-consuming and requires large amounts of storage. In a distributed computing system, the problem is compounded by the fact that there are typically many processes. Examining and analyzing such voluminous amounts of traces can be a problem itself. However, traces are the only way to discover subtle synchronization bugs in distributed computing system in many cases. In a distributed computing system traces can be very expensive to generate and use. One possible way to solve this problem is to significantly reduce the amount of data that is recorded in the process traces by only making entries for the truly significant or important process events. It is difficult to precisely define which events are the important events, since it depends on what the process is doing. However, the important events should include all events which directly or indirectly can affect other processes. All variable state changes and the outcome of all if statements are not necessarily important events. The trace should be high level ignoring most low level details.

5.3 Controlled Execution

Traces may be very useful, but in some special cases we may want to run the system in a controlled fashion, observing the intermediate states. This is analogous to running a sequential program one instruction at a time under the control of an interactive debugger, or analogous to running the program until a breakpoint is encountered. If we want to use these ideas in a distributed system, we must be careful, since there are multiple processors. Out of the various candidates, which is the next instruction to be executed? If we set a breakpoint in a process, do we just want to stop that process, or do we want to stop the entire system?

In order to properly examine the state of the system, all processes should be stopped. This can be achieved by broadcast a stopping messages to all processes. The stopping message can be triggered either by the programmer or by a process which encounters a breakpoint.

Once all processes are paused, the programmer, through the master debugger, should be able to examine the state of the processes. This is straightforward if we assume that the debugging module at each computer contains an interactive debugger. If the programmer wishes to know the contents of a variable of process A, a requested is sent to the computer running process A. At that computer, the interactive debugger is used to look into the storage area of process A, and the requested value is forwarded to the master debugger. Process traces should also be kept for the programmers to consult them. After examining the state and the traces, the programmer can set other breakpoints and resume operation of the system by broadcasting a continue message to all processes.

If the programmer wants to execute one instruction only and then go back to examining the state and traces, the programmer must decide which one of the processes should be the one to execute its next instruction. For the programmer to make this decision, the master debugger must be able to display a list of all the active and ready processes on the master terminal. An active process is one that was running when the system execution was paused. A ready process is one which is ready to run but was not actually running. Only ready and active processes can execute their next instruction. When the programmer selects a process, the master debugger instructs the computer which runs it to execute its next instruction only. After that, the programmer can go back to examination mode.

In many cases, executing a single instruction may not be very productive. A better approach would be to run a process until its next significant event. Significant events are the events that we are tracing. From the point of view of the interaction among the processes, nothing much occurs in a process between significant events. Therefore, to find bugs which result from the interaction of processes, it makes sense to run a process until such an event occurs. Thus, when a programmer selects a process to run, the programmer can be given the option of executing a single instruction or the option of running the process until its next significant event.

Chapter 6
Research on Monitoring Tools for Distributed Systems

Since my project is monitoring a distributed system, it is important for me to understand the features of monitoring distributed systems. To do that, I have read some research and papers done by others and summarized them. In this chapter, the issues of monitoring distributed systems and different approaches to monitoring are discussed.

6.1 Distributed Computing

A distributed system is a system in which a collection of processes interacting with each other to accomplish a common goal. Distributed computing systems tend to be large. They have a large number of processes and processors. Distributed computing can refer to the system hardware, the system software, or both. The hardware includes the processors, the communication system, the processor interfaces to the communication system or nodes. The software includes the collection of programs running on the processors. These programs operate in an integrated fashion in order to achieve the common system goal. Since my project is related to distributed software, not hardware, distributed system only refers to the software. To study distributed computing, first we need to distinguish the difference between parallel and distributed computing. In parallel computing, all processors have access to a shared memory space to interact with each other (Figure 6.1 (c)). In distributed computing, each processor has its own memory space. Interaction is done by exchanging messages among processors (Figure 6.1 (b)).

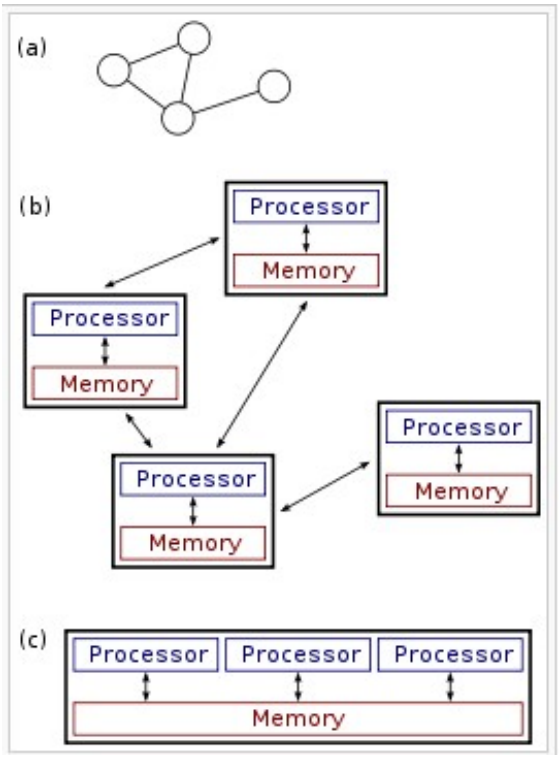


Figure 6.1 Parallel and Distributed Systems

Although we have distinguished parallel and distributed computing, these two terms overlap. A system can be both parallel and distributed.

6.2 Object Oriented Distributed Computing

In distributed object computing, objects can be loaded across a variety of platforms and in different processes and can communicate transparently with each other by issuing method requests as if they were located on a single machine. Figure 6.2.1 is an illustration of object oriented distributed computing.

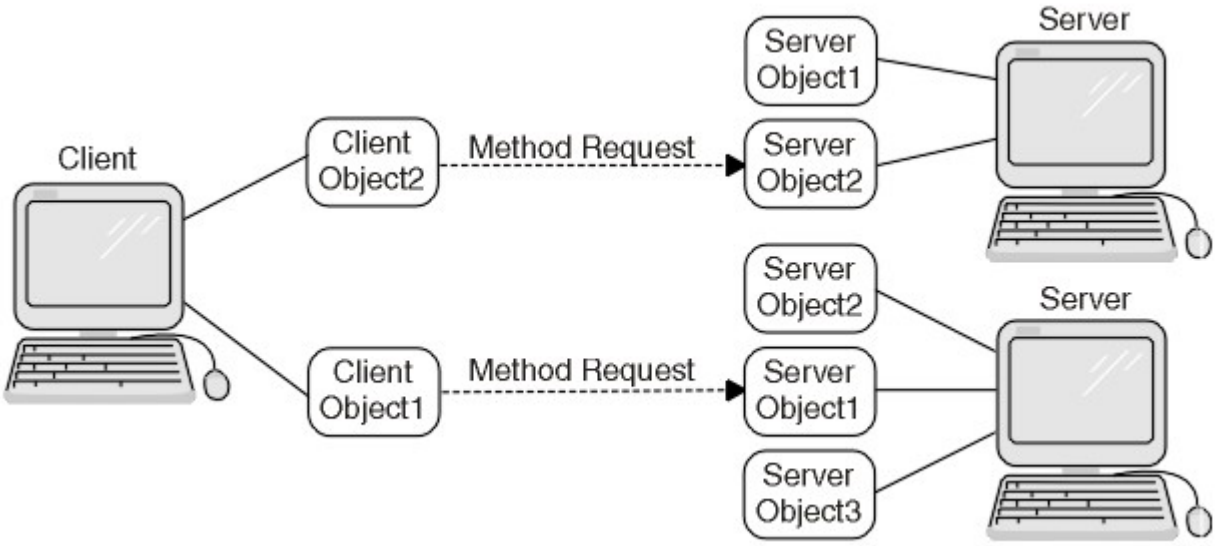


Figure 6.2.1 Object Oriented Distributed Computing

In Figure 6.2.1 client objects invoke methods on implementation objects exported by servers, implementation objects are typically located on different machines from the clients. Clients are unaware that the methods are being performed remotely, much like conventional clients that make Remote Procedure Calls (RPCs).

The Distributed Computing Environment (DCE) includes features that support the use of distributed objects. For example, DCE provides an interface definition language (IDL) that provides a way to group related operations together in a logical fashion, much the way a class definition does. An interface definition can be mapped to a class definition in which class member functions represent the operations defined in the interface, which is illustrated in Figure 6.2.2.

```
interface account
{
    void credit([in] float amount);
    void debit([in] float amount);
}
```

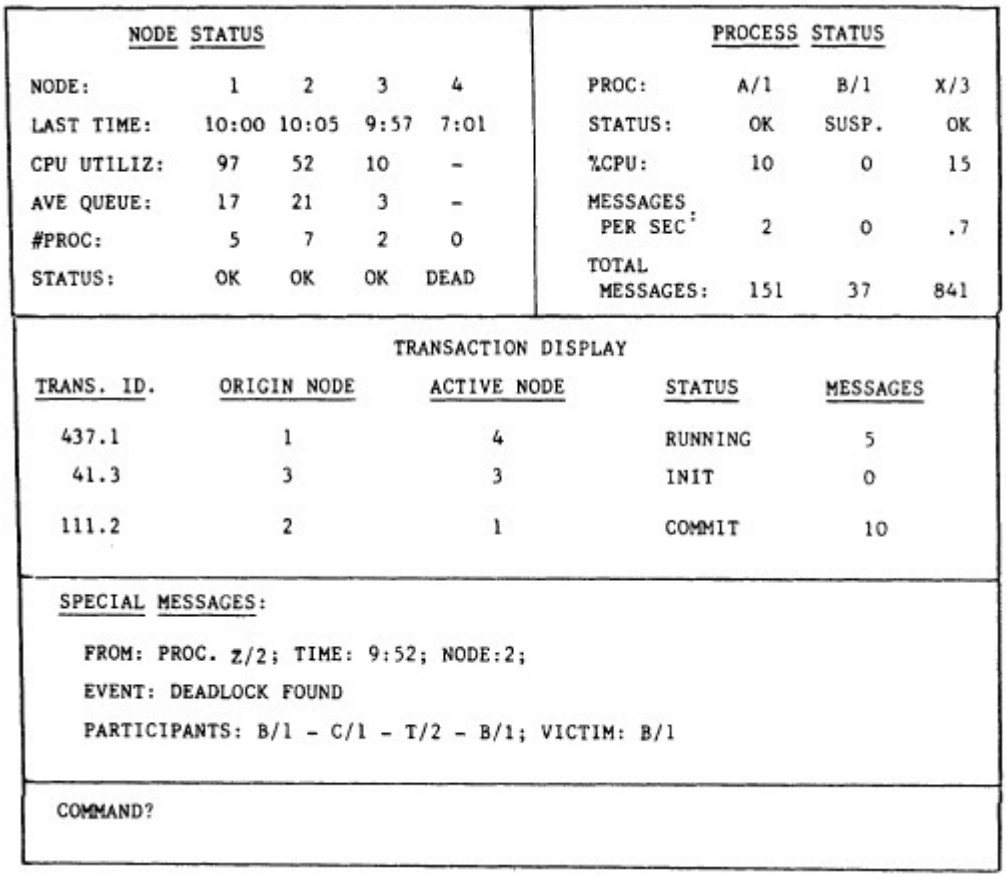
```
class account
{
public:
    void credit(float amount);
    void debit(float amount);
}
```

Figure 6.2.2 Simple mapping of a Distributed Computing Environment interface to a C++ class

6.3 Monitoring

After testing and debugging a distributed system for some time, we have reached a stage where the system seems to be running properly, at which point we probably cannot afford to examine every trace that is produced by the system. Still, we would like to keep an eye on the system to make sure that things are running smoothly.

The main idea is to provide the programmer on his or her master terminal with a real-time summary of the system execution. This overall picture should give the programmer information such as what operations are being executed, where the operations are being executed, the status of the processes and nodes, the message traffic and so on. This information will be helpful in observing the system performance and in discovering bottlenecks or other abnormal behaviour. Figure 6.3 illustrates one such type of display.



NODE STATUS

NODE:	1	2	3	4
LAST TIME:	10:00	10:05	9:57	7:01
CPU UTILIZ:	97	52	10	-
AVE QUEUE:	17	21	3	-
#PROC:	5	7	2	0
STATUS:	OK	OK	OK	DEAD

PROCESS STATUS

PROC:	A/1	B/1	X/3
STATUS:	OK	SUSP.	OK
%CPU:	10	0	15
MESSAGES PER SEC:	2	0	.7
TOTAL MESSAGES:	151	37	841

TRANSACTION DISPLAY

TRANS. ID.	ORIGIN NODE	ACTIVE NODE	STATUS	MESSAGES
437.1	1	4	RUNNING	5
41.3	3	3	INIT	0
111.2	2	1	COMMIT	10

SPECIAL MESSAGES:

FROM: PROC. Z/2; TIME: 9:52; NODE:2;
EVENT: DEADLOCK FOUND
PARTICIPANTS: B/1 - C/1 - T/2 - B/1; VICTIM: B/1

COMMAND?

Figure 6.3

In Figure 6.3, there is node status, process status, transaction (operation) display, special messages and command. The selection of information displayed is based on the functionality of the monitoring system. Other information such as traffic status may also be displayed.

6.4 The issues that need to be addressed

Although there is a set of techniques to reduce the number of bugs when the program is written, bugs will exist and there is always a need for debugging. The debugging of a distributed computing system is similar to the debugging of sequential computing system. We are familiar with sequential computing debugging – after a program is writer, various test cases are run on a single computer. When an unexpected behaviour is observed, the program is analysed to find out the cause of the problem. A set of tools are available for programmers to debug, such as interactive symbolic debuggers, interpreters, memory dumps, audit and trace packages, etc..

However, distributed systems are more difficult and time-consuming to debug, test and evaluate than sequential systems [7, 8, 11]. As we know, the larger a system is, the harder it is to debug. Largeness is not the only reason why distributed systems are difficult to debug. Typically, distributed computing systems are currently debugged as follows: each process in the system is run through a standard interactive debugger. To avoid confusion, the output of each debugger is displayed on a separate terminal. This requires not only a lot of running between terminals. The moving could be eliminated by multiplexing all the terminals into a single one. A number of reasons and the main issues that need to be addressed by monitoring tools for distributed systems are listed below.

1. A distributed system has many control foci. Sequential monitoring techniques cannot be used directly on distributed systems. Sequential monitoring techniques need to be extended to accommodate distributed systems. The distributed computing system is made up of a collection of processes which communicate via messages. In many cases, bugs are caused by improper synchronization among processes. The results observed in the system are hard to re-create, because they do not only depend on the system input, but also depend on the relative timing of the processes.

2. It is difficult to determine the state of a distributed system due to the communication delay among nodes. The message of a node may arrive at its destination node at unpredictable times. The processes being monitored can be running on different physical processors. The process that executes the illegal operation can be stopped immediately, but other processes will continue running. In order not to lose any critical information regarding the bug, the other processes should be stopped as soon as possible, but this will take some time. By the time all processes are stopped, the critical information may be lost and it will become difficult to discover the cause of the problem.

3. Distributed systems are inherently nondeterministic. That means that two executions of the same distributed system may produce different orderings of events. Thus, it is difficult to reproduce errors.

4. Monitoring a distributed system affects its behaviour. Sequential systems will not be altered by monitoring tools, however, this does not apply to distributed systems. Stopping or slowing down one process in a distributed system may alter the behaviour of other processes and the entire system.

5. Interactions between distributed systems and the system manager are more complex than sequential systems. A system manager may need to physically start the monitoring processes from different terminals.

One common problem of both sequential and distributed system monitoring is that the monitoring system need to handle the large amount of data produced during the monitoring process.

6.5 Jade Monitoring System

[7] presents a monitoring system for distributed systems in the context of the Jade programming environment. This system addresses all the issues listed in section 5.4. Before we go into the details of Jade monitoring system, section 5.5.1 is a brief introduction of the Jade programming environment.

6.5.1 Jade programming environment

The Jade programming environment supports the development and monitoring of distributed systems. Its distributed monitoring system as well as its inter-process communication facility, window systems and interactive graphics editor enables the monitoring of processes running on multiple machines. The window system allows user to create and manipulate windows. A window act as a virtual terminal to a Unix host, which allows user from a single workstation to interact with processes running on remote machines.

6.5.2 Jipc Interprocess Communication Facility

The interprocess communication protocol used by Jade programming environment is Jipc. Jipc is implemented in Unix system. The processes in Jade system communicate with each other using Jipc message protocol. Jipc also provides the primitives for creating, destructing and searching processes. Its primitives include *send*, *receive*, *receive_any*, *forward* and *reply*.

- *Send* – transmit a message from its one process (sender) to another (receiver) and block the sender until it receives a *reply* primitive.
- *Receive* – When a process receives a message from a specified process, it calls *receive* primitive.

- *Receive_any* – When a process receives a message from any process, it calls *receive_any* primitive.
- *Reply* – After receiving and processing a message, the receiver can send a *reply* primitive to the sender. When the sender receives this *reply* primitive, it becomes unblocked.
- *Forward* – After receiving and processing a message, the receiver can ask another process to send a *reply* primitive to the sender using the *forward* primitive.

6.5.3 Monitoring System Architecture of Jipc System

The separation of detecting information and analyzing makes the Jade monitoring system extensible. When a user wants to extend the Jade monitoring system, he/she does not need to be concerned with detecting information. He/She simply needs to consider how the information is presented.

Figure 6.5.3 gives an example of the architecture of Jade monitoring system. There are six processes, which are represented by circles, running on two machines. The process called Channel, which is represented by square, is running on each machine, and is collecting the information from the processes running on them. Each console, represented by rectangle, receives information from one or more channels, and it presents the information to users.

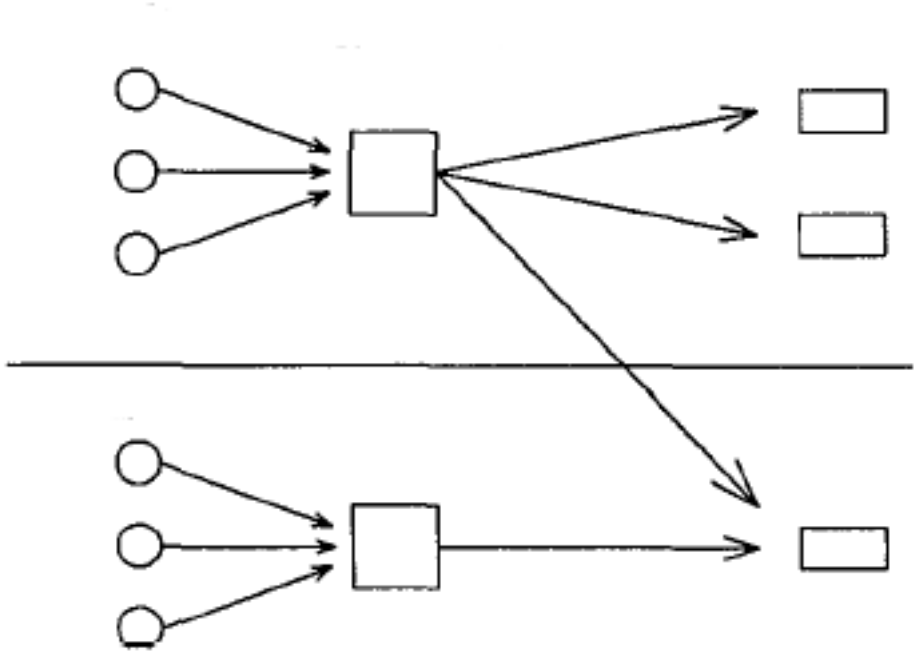


Figure 6.5.3 The architecture of Jade monitoring system

6.5.4 IPC Mechanism of Jipc System

IPC refers to Inter-Process Communication. A monitoring system needs to be able to determine how to collect the events generated by processes. To do this, an IPC mechanism for monitoring information must be defined. There are two approaches to the problem. One is to use the same IPC mechanism used by the processes. The other approach is to use a different IPC mechanism than the one used by the processes. Jipc has adopted the first approach, that is, to use the same IPC mechanism used by the processes. The advantage of this approach is that it makes the monitoring system more portable, since it relies on only one message-passing mechanism. However, this approach requires the monitoring system to be able to distinguish between the messages that pass information among processes and the messages that pass the monitoring system.

A Jipc message containing the information about the event is sent to the local channel whenever an event is detected in a process. Only the messages that pass the monitoring information are sent to the local channel. This is how the Jipc monitoring system distinguishes between messages that pass monitoring information and messages that pass information among processes.

6.5.5 Solving the Problem of Nondeterminism

One of the difficult problems of monitoring distributed systems is the inherent nondeterminism feature of distributed system. Independent events can occur in random order, so a correct execution of an application corresponds to a partial order of the communication events. Jipc monitoring system can be used to control the order of events. It can automatically recreate an execution path from a recorded trace. A controller allows users to determine the order in which pending events occur. When a Jipc primitive (described in Section 6.5.2) is invoked by a process that is being monitored, a message is used to notify the local channel. The channel will distribute this message to all consoles and allow the process to continue. The monitored process is blocked until it receives a reply from the channel. A controller is able to delay this message to the monitored process in order to prevent the monitoring process from continuing.

There will a set of pending events that are being delayed by the controller at any given time. The user can ask the controller to release a set of pending events or to continue receiving event messages from other monitored processes. The user can ask the controller to release the pending events that will result in the controller distributing the event to the consoles and allowing the monitored process to continue at any time. In this way, the system operates normally, but no event can be completed until the user allows it. The user is able to produce any sequence of events that could be generated by the processes, which allows the user to observe how a system behaves.

Controller can also be used to produce an event sequence based on logical or simulated time. A logical clock can be related to each process. Messages sent by a process is time stamped with the value of the logical clock at the time the message is sent. The controller can select the event with the smallest time stamp from the set of pending events as the one to execute next. This feature can be used during the development process to simulate the operations and the interaction of the processes that are being monitored.

An important requirement for monitoring distributed systems is system state re-creation. The ability to re-create the execution of a distributed system allows errors that only manifest themselves on selected execution paths to be isolated and debugged. Jipc does so using a console and a controller. The console keeps a record of all events that are executed in an application system and the commands used by the user that can affect the execution of the system. When a system has finished its execution, the controller can use the record kept by the console to re-create the original execution of the system by taking the events from the record and compare them with the events in the system. Thereby the controller is able to know what event must be executed next to ensure that this execution matches the original execution. The controller waits for this event to occur or waits until it knows that this event cannot occur. Re-creation of events continues until a process does something different from what the record indicates or until the user enters a command which would change the execution or until the user stops re-creation.

6.6 Features That a Good Monitoring System Should Have

After researching into different distributed monitoring systems, their strength and weakness and the problems that they address, here is a summary of what a good distributed monitoring system should have.

1. Collecting and distributing monitoring information should use the same inter-process communication protocol as the application processes. This simplifies the design, debugging and maintenance of the monitoring system. When an application is ported to a target network of machines, support for the monitoring system is available without having to port a second inter-process communication protocol.

2. The detection and collection of monitoring data should be separate from its analysis and display. This separation supports the development of an integrated set of tools that share a common implementation and that can work together effectively. In this way, writers of new tools do not have to be familiar with the low-level details of how monitoring information is gathered, which allows a variety of monitoring tools to be implemented efficiently.

3. A variety of different monitoring views and interpretations is needed. No single tool can display all the information the user requires. The set of tools required must have at least two dimensions. First, it is necessary to be able to move from low-level inter-process communication views through higher levels of abstraction. Second, the use of complementary views, such as textual display and graphical display, is very effective.

4. Textual and graphical displays offer an orthogonal view of the execution of a system. Textual display offers a complete record of the execution of a system. Graphical display offers a clear picture of the state of the system. Textual display is more useful when the user is tracking down the cause of an error. Graphical display is more insight into the overall operation of a system.

5. A good distributed monitoring system should have the ability to control nondeterminism and to re-create specific execution paths. This allows better test coverage since execution paths can be explicitly tested. Erroneous executions can be reproduced easily. The ability to control nondeterminism also supports system prototyping. The use of a combination of real and simulated tie to automatically determine the order of independent events enables the execution of an application system to be coordinated with simulations of unavailable components.

6. Animated and graphical display provides better form of dynamic documentation. Animated and graphical display is used most often to demonstrate a system to those who are unfamiliar with the system's structure and operation.

7. A good monitoring system should be able to exploit semantic information about the application system being monitored. When the user is monitoring the system at low level, i.e. inter-process communication level, the monitoring system must be able to interpret and display information in a way that reflects the structure and dynamic behavior of the system. The monitoring system must be able to recognize the pattern of interactions at low level.

Chapter 7
Other Research

This chapter describes the research that I have done so far related to the Distributed Service Profiler. All the research work gives me a better understanding of how the Distributed Service Profiler works and how I can extend it.

7.1 Profiling Tools

Program profiling or software profiling is a form of dynamic program analysis that measures, for example, the space of memory or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls [5]. Profiling helps developers analyse their code and help them find bottlenecks, i.e. the part of code that consumers the most memory, or the part of code that takes the longest time to process.

JProfiler is developed by ej-technologies GmbH, and is targeted at Java EE and Java SE applications [6]. JProfiler is a commercially licensed Java profiling tool that works both as a stand-alone application and as a plug-in for Eclipse software development environment. The difference between JProfiler and the Distributed Service Profiler by Kiwiplan is that JProfiler only analyses performance within programs whereas the Distributed Service Profiler analyses performance between various services.

7.2 Hibernate Interceptor

An interceptor is used to intercept or hook different kinds of database operations. It allows the application to inspect and manipulate the properties of a persistent object before it is saved, updated, deleted or loaded. Hibernate Interceptor is a Java interface which can be configured with an application, which will be the Distributed Service Profiler in this case, and a database to intercept queries made to the database by that application. It will be invoked every time a select, update, insert or delete is made to the database. We can either implement Interceptor directly or extend the EmptyInterceptor interface.

There are two kinds of interceptors, Session-scoped and SessionFactory-scoped. A Session-scoped interceptor is when a session is opened using one of the overloaded SessionFactory.openSession() methods.

Session session = sf.openSession(new MyInterceptor());

A SessionFactory-scoped interceptor is registered with the Configuration object. The supplied interceptor will be applied to all sessions opened from the SessionFactory. As mentioned in section 3.3, SessionFactory objects are thread safe, which makes the interceptors thread safe.
new Configuration().setInterceptor(new MyInterceptor());

7.3 EmptyInterceptor Interface

EmptyInterceptor interface is part of the Java Hibernate package. It is an interceptor that does nothing [4]. EmptyInterceptor provides necessary methods that can be used to inspect and manipulate the properties of persistent objects. I looked into EmptyInterceptor, because I have decided to implement the methods in EmptyInterceptor to intercept queries made to the database in my project. Not every method in EmptyInterceptor will be implemented. The methods that I am going to implement are listed in Table 7.1 below with their descriptions. The return types and parameter types are omitted for simplicity.

Method Name	Description
afterTransactionBegin()	Called when a Hibernate transaction begins.
afterTransactionCompletion()	Called after a transaction is committed or rolled back.
onDelete()	Called before an object is deleted.
onFlushDirty()	Called when an object is detected to be dirty during a flush.
onLoad()	Called just before an object is initialized.
onSave()	Called before an object is saved.
postFlush()	Called after a flush.
preFlush()	Called before a flush.

Table 7.1 Methods in EmptyInterceptor

Since this report is not the final report, other methods from the EmptyInterceptor interface but not in Table 7.1 above may also be implemented in the project.

A stopwatch (e.g. System.currentTimeMillis()) can be put in the afterTransactionBegin() and afterTransactionCompletion() methods to measure the time taken for each query to complete. The total time of all queries will just be a sum of the time taken for each query to process. To measure how many database objects have been created, updated or deleted, we can put a counter in onSave(), onFlushDirty() and onDelete() methods.

Chapter 8
Programming Work

This chapter describes the work of the coding part that I have done so far.

8.1 Hibernate Testing Application

To ensure that I understand how Hibernate works and what kind of configurations needs to be done, I made a small Hibernate testing application. To make Hibernate interceptor work, we need at least the following files:

- An interceptor, which extends the EmptyInterceptor, called MyInterceptor.java in my testing application.
- A table in the database. It will be the Employee table that I created for testing in the database. Employee table has a few attributes such as ID, first name, last name and salary. It is initially empty and will be modified by my testing application.
- A Java Employee class called Employee.java. Employee.java is basically just a set of mutators and accessors.
- A mapping file which maps the Java Employee class to the Employee table in the database. This file is called Employee.hbm.xml in my testing application.
- A Hibernate configuration file called hibernate.cfg.xml. This configuration file specifies the details about the database such as which URL to use, the password and the username of the database, and list the mapping files needed for the Hibernate Interceptor.

This Hibernate testing application will be a template for the LoggerInterceptor that I am going to implement for the Distributed Service Profiler. The code of this testing application is included in the appendix for future use.

8.2 LoggerInterceptor

The LoggerInterceptor is the Hibernate interceptor created specifically for the

Distributed Service Profiler. It is very similar to the Hibernate testing application that has been discussed in section 8.1 and it implements the methods as in the MyInterceptor.java in the Hibernate testing application, onSave(), onLoad(), onDelete(), afterTransactionBegin(), afterTransactionCompletion(), onFlushDirty(), preFlush() and postFlush(). The parameters of these methods are worth studying, since they are related to the database operations, and we want to see exactly what these operations do, and how they interact with the database. The parameters provide information on the entity that is currently being processed. Let us illustrate these parameters using an example. Given an EMPLOYEE table below:

EMPLOYEE			
Id	First_name	Last_name	Salary
001	John	Williamson	1000
002	Lisa	Key	3500

After an update operation on the first row, the modified table is shown below:

EMPLOYEE			
Id	First_name	Last_name	Salary
001	John	Williamson	5000
002	Lisa	Key	3500

When the LoggerInterceptor intercepts this update query, onFlushDirty() is invoked. The method signature is onFlushDirty(Object entity, Serializable id, Object[] currentState, Object[] previousState, String[] propertyNames, Type[] types). The table below shows the content of the parameters (Table 8.1).

Parameter name	Meaning	Value
Object entity	Refers to the object currently being updated	Employee
Serializable id	Refers to the id of this row	001
Object[] currentState	Refers to the new values of this row (the values after update)	{001, John, Williamson, 5000}
Object[] previousState	Refers to the old values of this row (the values before update)	{001, John, Williamson, 1000}
String[] propertyNames	Refers to the attribute names (column names)	{id, first_name, last_name, salary}
Type[] types	Refers to the domains of the attributes	{integer, string, string, integer}

Table 8.1 The parameters of onFlushDirty()

Since the parameters of the methods implemented in LoggerInterceptor reflect the information of the entities being passed into the database, we can extract useful

information from those parameters.

8.3 LoggingManager

Apart from the LoggerInterceptor that we have discussed above in Section 8.2, a LoggingManager is created. It is used to initialise the LoggerInterceptor and is passed as one of the parameters into the LoggerInterceptor to pass on the service name and logging status. Logging Manager communicates with the service profiler by extending MBeans.

sendEntryNotification() and sendExitNotification() are implemented in LoggingManager in order to send notifications to the profiler where the query information could be used and displayed in the call tree. sendEntryNotification() and sendExitNotification() creates an AttributeChangeNotification and emits the notification to the profiler. This notification contains the method name, the time stamp and a message tag. The message tag indicated the start or the end of the query. For the LoggingManager to be able to communicate with the profiler, the profiler must be able to find the LoggingManager MBean to register to a listener. I used ManagementFactory.getPlatformMBeanServer() to register the LoggingManager to the MBeanServer.

8.4 Logging

Logging is the process of writing log messages during the execution of a program to a central place. This logging allows programmers to report errors and warning messages as well as info messages, e.g. runtime statistics, so that the messages can be retrieved and analyzed after the program is executed. To explore how the Hibernate interceptor intercepts queries made to a database in details, logging is used. All the important steps of Hibernate transactions, such as starting a transaction, committing a transaction, retrieving data from the database, etc., are recorded in a text file.

Java contains the Java Logging API, which allows programmers to configure which message types are written. Individual classes can use this logger to write messages to the configured log files. The java.util.logging package provides the logging capabilities via the Logger class. The log manager is responsible for creating and managing the logger and the maintenance of the configuration. java.util.logging.LogManager is the class that reads the logging configuration, creates and maintains the logger instances. We can use this class to set our own applications specific configuration. Give a property file ‘mylogging.properties’, below is the code for logger manager:

```
LogManager.getLogManager().readConfiguration(newFileInputStream("mylogging.properties"));
```

If we do not specify any configuration, the property file is read from JRE Home lib/logging.properties file.

Multiple handlers can be added to a logger and whenever we log a message, every handler will process it accordingly. There are two default handlers provided by Java Logging API:

1. ConsoleHandler: ConsoleHandler writes all the logging messages to console
2. FileHandler: FileHandler writes all the logging messages to file in the XML format.

Since the logging messages written by ConsoleHandler is not stored in a file, and thus is not available for future use, I have decided to use FileHandler. Formatters are used to format the log messages. There are two available formatters in Java Logging API:

1. SimpleFormatter: SimpleFormatter generates text messages with basic information. ConsoleHandler uses this formatter class to print log messages to console.
2. XMLFormatter: This formatter generates XML message for the log. FileHandler uses XMLFormatter as a default formatter.

We can create own custom Formatter class by extending java.util.logging.Formatter class and attach it to any of the handlers. The XML format output produced by XMLFormatter is difficult to read, therefore I have decided to create my own formatter. It is a simple formatter which formats the output produced by FileHandler.

MyFormatter.java

```
import java.util.Date;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;

public class MyFormatter extends Formatter {

	@Override
	public String format(LogRecord record) {
		return "Thread ID " +
record.getThreadID()+"::"+record.getSourceClassName()+"::"
			+record.getSourceMethodName()+"::"
			+new Date(record.getMillis())+"::\n "
			+record.getMessage()+"\n";
	}
}
```

8.5 Stack Trace

We need to find a way to match the database queries (callees) with the service method calls (callers). There are several ways to do it, and I have chosen stack trace, since it is the simplest to implement, and it does not involve the PersistenceWrapper.

The stack trace from the interceptor goes back to the point where the service makes the database call. Thread.currentThread().getStackTrace() can be used to get an array of StackTraceElements. Each element in this array presents a single stack frame, and is a method invocation containing the declaring class, method name, file name and line number. getClassName() and getMethodName() can be called on each element. Thread.currentThread().getStackTrace() is called in the startQuery() method in LoggingInterceptor, which is called each time a database query is intercepted.

Service method calls are stored as tree nodes. Database calls would originate from leaf nodes, i.e. their callers. We can check the method names from list of leaf nodes with methods from stack trace. Once the calls are matched, a new node will be created containing the database call and will be linked to the service method node (caller)

Stack trace is used to match the database calls with the original service method calls. This is essential in order to be able to integrate the information from the interceptor with the profiler to build up a call tree.

Chapter 9
Future Work

This chapter describes the future work that needs to be done to complete the distributed service profiler, and what extensions can be done to complement the basic functions of the profiler.

Since I did not complete all the implementations of the distributed service profiler, future programming work needs to be done. Hopefully all the details and code provided in this report will help future programmers who work on distributed service profiler.

I have created a LoggerInterceptor and a LoggingManager, but currently they are just standalone applications. Therefore, integrating the LoggingInterceptor and LoggingManager with the distributed service profiler is part of the future work. One thing to note is that since the distributed service profiler needs to communicate with other services, the LoggingInterceptor and LoggingManager should be configured to be able to intercept the queries from all the services which the distributed service profiler communicates with.

Matching the database queries (callees) with the service method calls (callers) can be done using the stack trace. However, we still need to find a way to create the database query nodes and display them, which relates to the GUI part of the distributed service profiler.

Testing also needs to be done to debug the implementation and examine the usability of the distributed service profiler. After testing, there might be some additional programming work in order to fix the bugs and improve the performance.

Additional features of the distributed service profiler can also be done if time allows. These additional features include:

- Display the largest argument bytes for each method call
- Order nodes by time both ascending and descending
- A search function, which allows users to search nodes in the call tree by method name.

Chapter 10
Difficulties and What I Have Gained

This chapter describes the difficulties and concerns that rise in this project, and the actions that I am going to take to overcome them. What I have gained from this project is also included in this chapter.

10.1 Time Management

Although I go to Kiwiplan each week regularly, I still lack the time management skills of balancing my other course work and this project. This year has been a busy semester, since apart from this project I also taking some other courses, which require me to work on other projects, and working part time at university. In Semester Two, 16 to 20 hours' work each week needs to be guaranteed for this project, which involves 8 to 10 hours at the company and the other 8 to 10 hours doing academic work. In order to improve my time management skills, I can start with making a plan of what needs to be done each week and follow the plan strictly to develop a good habit of completing everything on time and even earlier than the deadline.

10.2 Academic Knowledge

The biggest difficulty that I come across is the lack of academic knowledge. When I was examining the source code of the Distributed Service Profiler, there were a lot of things I did not understand and I had to look into the Java API, which has decreased my efficiency significantly. Hibernate, MBeans, notifications, etc. were all new concepts to me, so I had to start from zero and a lot of time was spent on doing research. The lack of academic knowledge can be complemented by doing more research and reading more academic papers related to profilers on distributed systems. The experience of doing this project has increased my knowledge on various frameworks, such as Hibernate, MBeans, etc.

10.3 Distributed Systems

Since the distributed service profiler is not just a standalone application and it communicates with other services, I need to consider how my implementation can

integrate with the database layer as well as intercept queries made by all the other services, which increases the difficulty level of this project. At first, I did not realize how distributed service profiler works with other services, so the implementation I did in the beginning was not able to intercept the queries successfully. Knowledge on distributed systems and monitoring distributed systems is also gained during this project.

10.4 Development Process

One of the many differences between this project and the course assignments that I have done in the past is that this project requires a full development process, whereas for the course assignment, I only need to follow the instructions and complete the code. For this project, since the problem was already identified, I started with requirement specifications and designing. After that was the actual implementation, and finally, testing needs to be done.

10.5 Documentation and commenting

The importance of documentation and commenting is not so obvious when I am doing the course assignments as university, since there is a specific instruction specifying what you should do for the assignment and the lecturers and markers knows what you are doing. However, for this project, I noticed that documentation and commenting is extremely important, especially when you want other people to understand your code and possibly work on it. The distributed service profiler is a large application which communicates with other services, so the code is a bit complicated. However, with the help of documentation and commenting written by the programmers, I found it easier to understand the codes. I could not imagine how much time it would take me to go through the code of distributed service profiler without the documentation and commenting.

Code Appendix

This appendix contains the code for my Hibernate testing application described in section 7.1.

Employee.java

```
import javax.persistence.Entity;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
```

```
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }
}
```

Hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
    <property name="hibernate.dialect">
        org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
        com.mysql.jdbc.Driver
    </property>

    <!-- Assume test is the database name -->
    <property name="hibernate.connection.url">
        jdbc:mysql://localhost:3306/tss_422
    </property>
    <property name="hibernate.connection.username">
       root
    </property>
    <property name="hibernate.connection.password">

    </property>

    <!-- List of XML mapping files -->
    <mapping resource="Employee.hbm.xml"/>
     <event type="load">

                <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
            </event>
</session-factory>
</hibernate-configuration>
```

MyInterceptor.java

```
import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

import java.io.ByteArrayOutputStream;

public class MyInterceptor extends EmptyInterceptor {

     private static final long serialVersionUID = -150609044112042689 9L;

    public static int updates;
    public static int creates;
    public static int deletes;
    public long time;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
        deletes++;
    }

    // This method is called when Employee object gets updated.
    public boolean onFlushDirty(Object entity,
                         Serializable id,
                         Object[] currentState,
                         Object[] previousState,
                         String[] propertyNames,
                         Type[] types) {
        if ( entity instanceof Employee ) {
            //System.out.println("Update Operation");
            updates++;
            return true;
        }
        return false;
    }
```

```
    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        // do nothing
        return true;
    }

    // This method is called when Employee object gets created.
    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        if ( entity instanceof Employee ) {
            //System.out.println("Create Operation");
            creates++;
            return true;
        }
        return false;
    }

    //called before commit into database
    public void preFlush(Iterator iterator) {
        //System.out.println("preFlush");
    }

    //called after committed into database
    public void postFlush(Iterator iterator) {
        //System.out.println("postFlush");
    }

    public void afterTransactionBegin(Transaction tx){
        time = System.currentTimeMillis();
    }

    public void afterTransactionCompletion(Transaction tx){
        time = System.currentTimeMillis()-time;
    }
}
```

ManageEmployee.java

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.stat.SessionStatistics;
import org.hibernate.stat.Statistics;

//import com.sun.tools.xjc.dtdx.DXAttribute;
//import com.sun.tools.xjc.dtdx.DXAttribute.Type;

public class ManageEmployee {
    private static SessionFactory factory;

    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();

        /* Add few employee records in database */
        Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
        Integer empID3 = ME.addEmployee("John", "Paul", 10000);

        /* List down all the employees */
        ME.listEmployees();

        /* Update employee's records */
        ME.updateEmployee(empID1, 5000);
        /* Delete an employee from the database */
       ME.deleteEmployee(empID2);

        /* List down new list of the employees */
        ME.listEmployees();
  }
```

```
    /* Method to CREATE an employee in the database */
    public Integer addEmployee(String fname, String lname, int salary){
         MyInterceptor mi= new MyInterceptor();
         Session session = factory.openSession( mi);
         //Statistics stats = factory.getStatistics();
         //SessionStatistics ss = session.getStatistics();
        Transaction tx = null;
        Integer employeeID = null;
        try{
            tx = session.beginTransaction();

            Employee employee = new Employee(fname, lname, salary);
            employeeID = (Integer) session.save(employee);
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {

            session.close();
        }
        //System.out.println(
        //                Thread.currentThread().getStackTrace()[1].getMethodName()+"
called by "+Thread.currentThread().getStackTrace()[2].getMethodName());
        System.out.println("addEmployee() time taken:"+mi.time/1000.0+" seconds");
        System.out.println("addEmployee() number of hits: "+mi.creates);
        /*String[] s = stats.getEntityNames();
        for(int i=0;i<s.length;i++){
            System.out.println(s[i]);
        }*/
        return employeeID;
    }

    /* Method to READ all the employees */
    public void listEmployees( ){
        Session session = factory.openSession( new MyInterceptor() );
        Transaction tx = null;
        try{
            tx = session.beginTransaction();
            String s = session.createQuery("FROM Employee").getQueryString();
            List employees = session.createQuery("FROM Employee").list();
            System.out.println("");
            System.out.println("*-------------------------------------*");
```

```
            for (Iterator iterator =
                                    employees.iterator(); iterator.hasNext();){
                Employee employee = (Employee) iterator.next();
                System.out.print("First Name: " + employee.getFirstName());
                System.out.print("    Last Name: " + employee.getLastName());
                System.out.println("    Salary: " + employee.getSalary());
            }
            System.out.println("*-------------------------------------*");
            System.out.println("");
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {
            session.close();
        }
    }

    /* Method to UPDATE salary for an employee */
    public void updateEmployee(Integer EmployeeID, int salary ){
         MyInterceptor mi= new MyInterceptor();
         Session session = factory.openSession( mi);
        Transaction tx = null;
        try{
            tx = session.beginTransaction();
            Employee employee =
                            (Employee)session.get(Employee.class, EmployeeID);
            employee.setSalary( salary );
           session.update(employee);
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {
            session.close();
        }
        System.out.println("updateEmployee()       time       taken:"+mi.time/1000.0+"
seconds");
        System.out.println("updateEmployee() number of hits: "+mi.updates);
    }

    /* Method to DELETE an employee from the records */
    public void deleteEmployee(Integer EmployeeID){
         MyInterceptor mi= new MyInterceptor();
```

```
         Session session = factory.openSession( mi);
        Transaction tx = null;
        try{
            tx = session.beginTransaction();
            Employee employee =
                        (Employee)session.get(Employee.class, EmployeeID);
            session.delete(employee);
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {
            session.close();
        }
        System.out.println("deleteEmployee()  time  taken:"+mi.time/1000.0+"
seconds");
        System.out.println("deleteEmployee() number of hits: "+mi.deletes);
    }


}
```

Acknowledgements

Dr. S Manoharan – Senior Lecturer at University of Auckland
– Coordinator for the BTech programme in Information Technology

Dr. Xin Feng Ye – Senior Lecturer at University of Auckland
– Academic Supervisor of my BTech project

Ana Stilinovic – Senior Developer at Kiwiplan
– Industry Supervisor of my BTech project

Tim Walker – Development Manager at Kiwiplan

I would like to thank Dr. S Manoharan for giving me this opportunity of working with Kiwiplan, Dr. Xin Feng Ye for supervising me throughout the project, Ana Stilinovic for being my industry supervisor at Kiwiplan and also Tim Walker for supporting me in this project. All the above people have given me valuable advice on the project as well as on the current trend of the IT industry. The experience that they shared with me is something that I can never learn from a lecture and I appreciate everyone's help.

Bibliography

[1] “Kiwiplan – Homepage”. http://www.kiwiplan.com/, June 2013.

[2] “Hibernate (Java)”. http://en.wikipedia.org/wiki/Hibernate_(Java), June 2013.

[3] “GNU Lesser General Public License”. http://www.gnu.org/licenses/lgpl.html, June 2013.

[4] “EmptyInterceptor (Hibernate JavaDocs)”. http://docs.jboss.org/hibernate/orm/3.6/javadocs/org/hibernate/EmptyInterceptor.html, June 2013.

[5] “Profiling (computer programming)”. http://en.wikipedia.org/wiki/Profiling_(computer_programming). June 2013.

[6] “JProfiler”. http://en.wikipedia.org/wiki/JProfiler. June 2013.

[7] Jeffrey Joyce, Greg Lomow, Konard Slind, and Brain Unger, “Monitoring Distributed Systems,” in ACM Transactions on Computer Systems (TOCS) Volume 5 Issue 2. New York, USA: 1987. pp. 121-150.
Available: http://dl.acm.org.ezproxy.auckland.ac.nz/citation.cfm?doid=13677.22723

[8] Leonard Kleinrock, “Distributed Systems,” in Communications of the ACM – Special issue: computing in the frontiers of science and engineering, Volumn 28 Issue 11. New York: November 1985. C.2.4, pp.1200-1213.
Available: http://dl.acm.org.ezproxy.auckland.ac.nz/citation.cfm?doid=4547.4552

[9] Hector Garcia-Molina, Frank Germano and Walter H. Kohler, “Debugging a distributed computing system,” in IEEE Transactions on Software Engineering, Vol.SE-10(2). March 1984. pp. 210-219.
Available:
http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5010224&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D5010224

[10] Eric Fabre and Vincent Pigourier, “Monitoring Distributed Systems with Distributed Algorithms,” in Decision and Control, Proceedings of the 41st IEEE Conference on, Volumn 1. December 2002. pp. 411-416.
Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1184529&tag=1

[11] Leander Conradie and Maria-Athina Mountzia, “A Relational Model for Distributed Systems Monitoring using Flexible Agents” in ACM Transactions on Computer Systems (TOCS), Volume 5 Issue 2. May 1987. pp. 121-150.
Available: http://dl.acm.org/citation.cfm?id=22723

[12] Masoud Mansouri-Samani and Morris Sloman, “Monitoring Distributed Systems,” in Network and distributed systems management. pp. 303-347.
Available:
http://dl.acm.org/citation.cfm?id=184430.184469&coll=DL&dl=GUIDE&CFID=256904645&CFTOKEN=28878111