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 so far for my BTech 451 project at Kiwiplan in Semester One.

There are 7 main sections 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 is the research on Hibernate, which will be discussed in Chapter 3 Hibernate Research. Chapter 3 gives a detailed discussion on Hibernate architecture, class objects, sessions, properties, persistent class, the advantages of Hibernate, what databases are supported by Hibernate as well as Hibernate Query Language (HQL).

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 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 6 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 8.

Contents

Chaper 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.

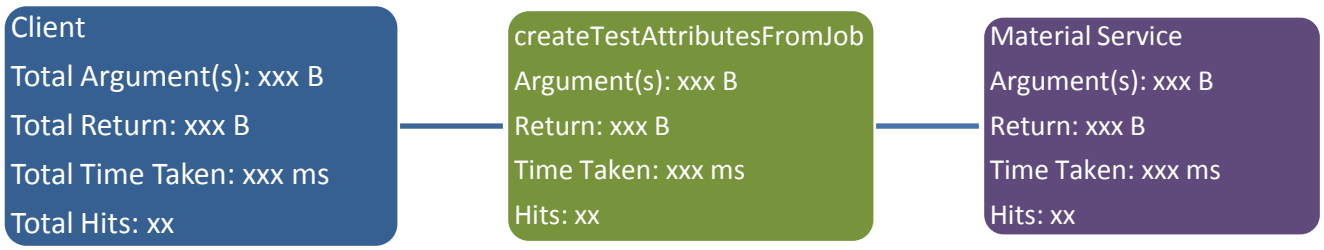

Figure 2.2 Illustration of the communication between services

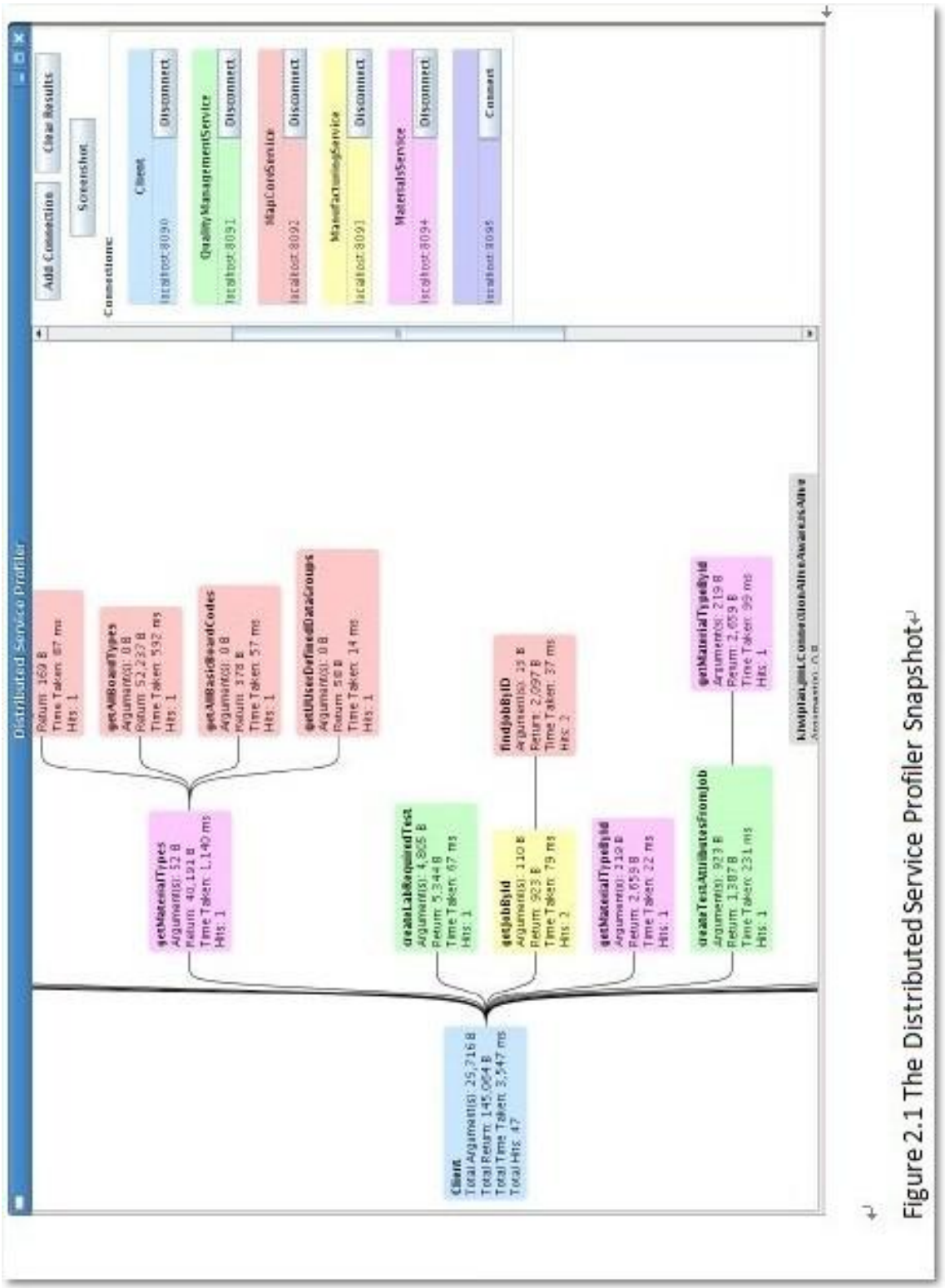

Figure 2.1 The Distributed Service Profiler Snapshot

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.

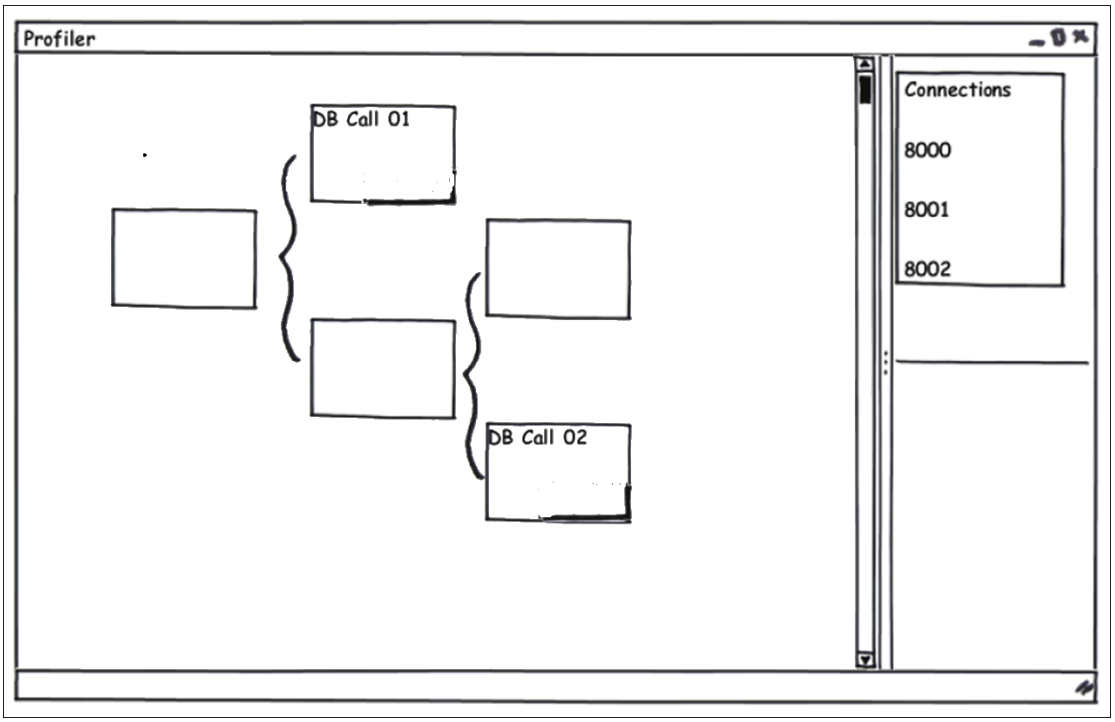

Figure 2.2 Mockup of the Distributed Service Profiler

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
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.

3.1 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 3.1 below.

Figure 3.1 The position of Hibernate

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

3.2 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 3.2 and Figure 3.3 below are a high level view of the architecture of Hibernate and a detailed view of the architecture of Hibernate respectively.

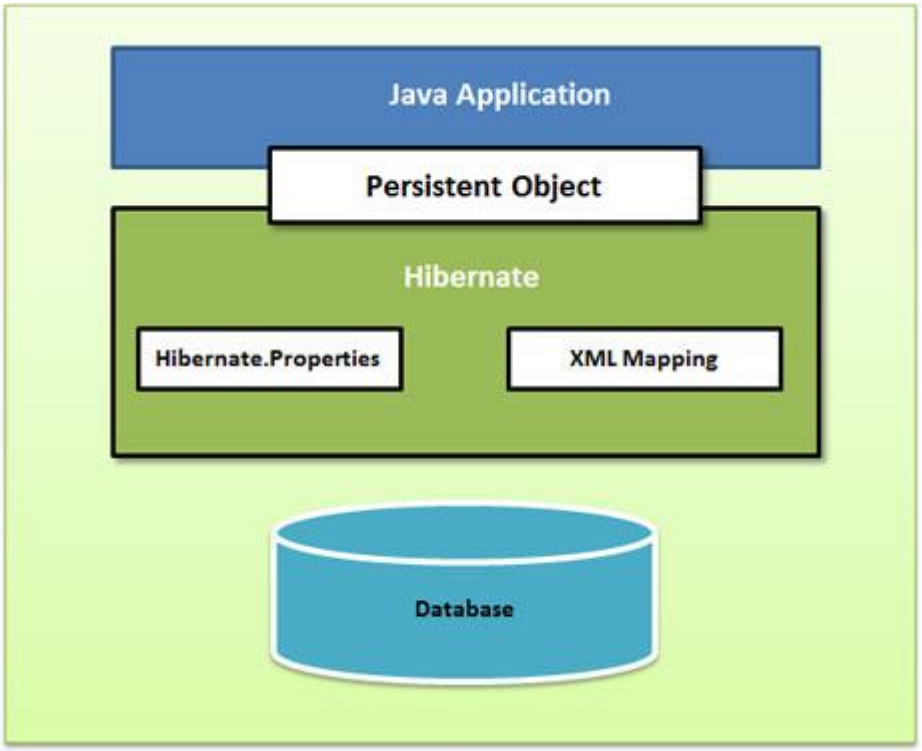

Figure 3.2 High level architecture of Hibernate

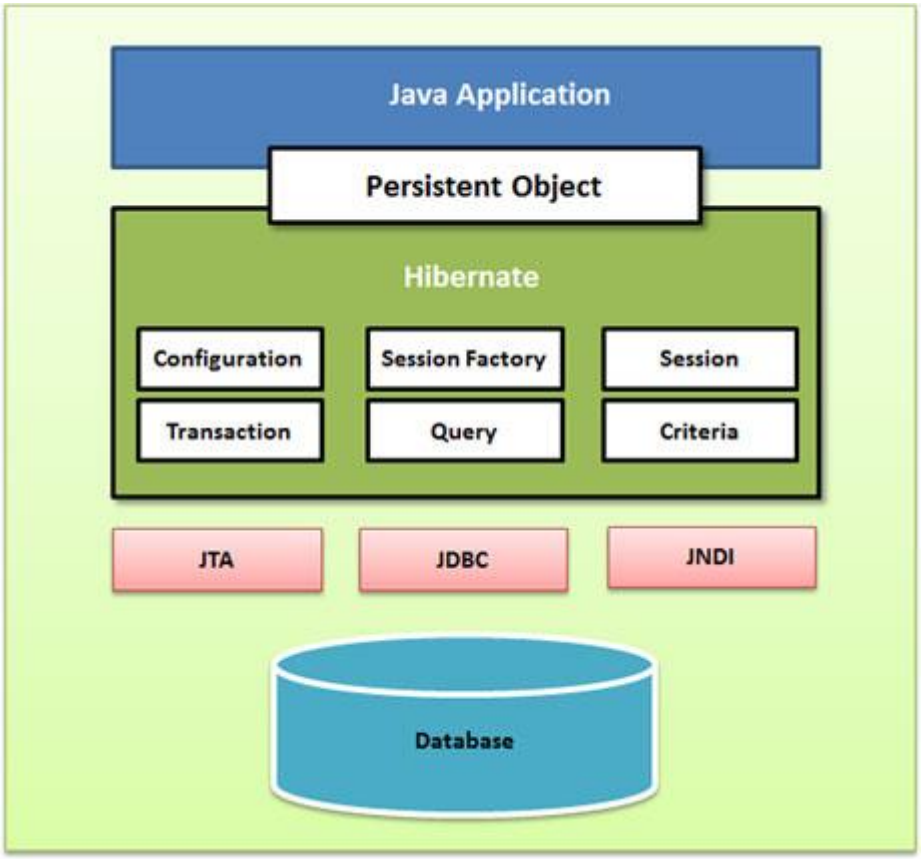

Figure 3.3 Detailed view of the architecture of Hibernate

In Figure 3.3, 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 3.3 are the Java objects used in Hibernate Application Architecture, and they will be explained in section 3.3 below.

3.3 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 3.4.

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.

3.4 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.

3.5 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 3.1 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 3.2 Hibernate Properties #2

3.6 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.

3.7 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.

3. 8 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

3.9 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.

Chapter 4
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.

4.1 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.

4.2 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());

4.3 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 4.1 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 4.1 Methods in EmptyInterceptor

Since this report is not the final report, other methods from the EmptyInterceptor interface but not in Table 4.1 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 5
Work So Far

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

5.1 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.

5.2 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 5.1 and it implements the methods as in the MyInterceptor.java in the Hibernate testing application, onSave(), onLoad(), onDelete(), afterTransactionBegin(), afterTransactionCompletion(), onFlushDirty(), preFlush() and postFlush().

Chapter 6
Future Work

This chapter describes what needs to be done and my plan for Semester Two.

6.1 Integrating the LoggerInterceptor with the Distributed Service Profiler

I have created the LoggerInterceptor, but it is still a standalone Java class. I need to integrate with the Distributed Service Profiler to put it into use.

Integrating the LoggerInterceptor with the Distributed Service Profiler will not be as easy as it is in my simple testing application, since the Distributed Service Profiler is quite a large application, and I need to identify where exactly to put the LoggerIntercepor in the Distributed Service Profiler. Therefore, more efforts and possibly research need to be made.

6.2 Extracting argument bytes and return bytes from queries

Currently, the LoggerInterceptor that I implemented only returns the time taken for each query to complete, and the number of hits. In the current Distributed Service Profiler, each node (includes each query node that I am going to add to the Distributed Service Profiler) needs to display the argument bytes, the return bytes, the time taken and the number of hits. To be consistent with the Distributed Service Profiler, I need to extract argument bytes that are passed to a query and the return bytes returned by the query as well. I need to look into the source code of the current Distributed Service Profiler to see how the profiler extracts the argument bytes and return bytes from the method calls. If I cannot use the same way for queries, then more research needs to be done.

6.3 Matching queries with their callers

Another important thing to do is to match queries with their callers so that we can display query nodes at the correct positions. As stated earlier, each query node needs to be attached to its caller. There are several approaches to do this, one of which is to

use the stack trace. Using stack trace is the simplest way, since it could all be done in a single class. Thread.currentThread().getStackTrace() can be used to get an array of StackTraceElements. Each element in the StackTraceElements represent a single stack frame.

6.4 Creating and displaying query nodes

Once all the above steps have been done, we need to create and display the query nodes. This involves me looking into the API of the current Distributed Service Profiler, since the design of the nodes will be the same. I have not done any research related to this part, since it can only be done after all the above steps are completed.

6.5 Researching on profilers for distributed systems

As suggest by Xinfeng Ye, my academic supervisor, I have decided to do more research on profilers for distributed systems. The Distributed Service Profiler is quite a large application, and it connects to all services provided by Kiwiplan making it even more complex. In order to fully understand how the Distributed Service Profiler works, I need to do expand my knowledge in profilers for distributed systems.

Chapter 7
Difficulties

This chapter describes the difficulties and concerns that rise in Semester One, and the actions that I am going to take to overcome them.

7.1 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. Semester One has been a busy semester, since apart from this project I am taking three other courses, and working part time at university. Most of the project is work is done at Kiwiplan, however, not much is done when I am at university or at home. 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.

7.2 Lack of Academic Knowledge

The biggest difficulty that I come across is the lack of academic knowledge. When I am examining the source code of the Distributed Service Profiler, there are a lot of things I do not understand and I have to look into the Java API, which decreases my efficiency significantly. The lack of academic knowledge can be complemented by doing more research and reading more academic papers related to profilers on distributed systems.

Overall, Semester One is the planning and investigating phase. There is still much to be done, both academic research and the actual coding, and there are difficulties to conquer. Since in Semester Two, 16 – 20 hours per week of work will be guaranteed, I am expecting to finish more work than Semester One.

Code Appendix

This appendix contains the code for my Hibernate testing application described in section 5.1.

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;
    }
}
```

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.