BTech Project Final Report

Shelley Lowe
4950257
slow056

2011

Abstract

This report explains the work done for the final year BTech project at Kiwiplan. It is a database based project implemented in Java. The main goal of the project is to extend an existing Distributed Service Profiler to integrate the database layer to intercept queries made to a database.

The profiler is used by developers working for Kiwiplan to help them analyse and find bottlenecks in their code. The profiler currently used by Kiwiplan developers only analyses service calls made between various services, not within. Queries to the database are not specified, even though the time taken for those queries are included in the information returned. This makes it difficult to find problem areas and information about database calls without developers manually going through the log files.

I have done a lot of research to get a better understanding the current profiler and find what tools I can use to complete this project. Research into Hibernate, Interceptors and logging has been done and explained in this report. Since the End of Semester report, I have done a lot more research into how the Distributed Service Profiler works and in particular, how MBeans work and how the profiler uses them to communicate with the various services.

Before going into the actual implementation of the profiler extension, I created some smaller standalone applications for testing and increasing my knowledge on the different frameworks. This included a simple Interceptor which I tested on a simple MySQL database. I also needed to create some new classes in order to achieve the goals of this project as well as modify some existing source code to make it possible to integrate the new components.

The system will still need to be improved and this will be included my plan of the future work.

Contents

Abstract **iv**

1 Project Description **1**
1.1 Problem Overview and Motivation 1
1.2 Company Information 2

2 Distributed Service Profiler **3**
2.1 Background Information 3
2.2 Related Work 4

3 Development Plan **6**
3.1 Solution Overview 6
3.2 Requirements 6
3.3 Architecture 7
3.4 User Interaction 8

4 Research **10**
4.1 Distributed Service Profiler 10
4.2 Hibernate 11
4.3 Hibernate Interceptor 11
4.4 EmptyInterceptor 12
4.5 Hibernate Event System 13
4.6 Logging 14
4.7 Hibernate Query Language (HQL) 14
4.8 Java MBeans 15
4.9 Notifications 16
4.10 JConsole 16
4.11 Database Changes Logger 17
4.12 Jini API 17
4.13 Maven 18

5 Design/Implementation **19**
5.1 Simple Hibernate Interceptor 19
5.2 PersistenceWrapper 20
5.3 LoggerInterceptor for Profiler 21
5.3.1 Original 21
5.3.2 Revised 23
5.4 LoggerManager for Profiler 24
5.4.1 Original 24
5.4.2 Revised 26
5.5 JMXClient Modifications 26
5.6 ClientListener Modifications 27
5.7 MBeanCreator Modifications 27
5.8 ProfilingManager Modifications 27
5.9 Matching Service Calls with Database Queries 28

6 Final Solution **31**
6.1 Solution Overview 31
6.2 Future Work 34

7 Knowledge Gained **36**
7.1 Knowledge on Various Frameworks/Tools 36
7.2 Development Process/Work Environment 37
7.3 Eclipse Extensions 37
7.4 Documentation/Commenting 37
7.5 Planning a Solution 38
7.6 Time Management 38
7.7 Presentation/Report Skills 38

8 Conclusion **40**
8.1 Summary 40
8.2 Conclusion 40
8.3 Acknowledgments 41

Bibliography **43**

Chapter 1

Project Description

This chapter describes the project and the company Kiwiplan.

1.1 Problem Overview and Motivation

Kiwiplan currently uses a Distributed Service Profiler to help analyse their code. This profiler can check calls and communication on a higher level to analyse the communication between the various services. An example of this communication is when a particular service calls another service, which then queries the database. Currently, developers must manually check the queries by going through the log files, similar to how they used to go through the code before using the profiler. The plan is to add database integration and automate the process to help Kiwiplan be more efficient in creating new solutions.

The goal of this project is to extend the capabilities of the Java distributed performance profiling tool by adding integration with the Hibernate ORM and the database layer. The new profiling tool should intercept calls made by the services to the database and display this information in the window. The information extracted from the database queries should be displayed as an extension to the call tree created by the current profiler.

Most of the services developed by Kiwiplan must communicate with each other and with the database. Some service methods take a lot of time to process, where the majority of these times are queries to the database. Queries to the database are not specified, even though the time taken for those queries are included in the service method information returned. This is typically the bottleneck in the system so Kiwiplan developers would like to know which methods are taking the longest so they can try to fix these problem areas.

Since the current service profiler only displays the process time under the

service method with no information on the database calls, developers must manually check the database logs. Checking where and what these database calls are is important to help developers find the bottlenecks and improve their code.

Without the database integration, the developers can only see which service methods are taking the longest. Any number of factors could cause this delay in computation time. If this information could be extended to display information on database queries, it would help the developers narrow down the problem areas and find out whether the long runtime is due to the actual service method or because of queries to the database.

1.2 Company Information

With over 30 years of expertise in developing software for the corrugating and packaging industry, Kiwiplan provides systems that deliver the high standard required for enterprise-wide, fully automatic integrated systems for Web-enabled supply chain management, sales order management, scheduling and planning, manufacturing, and inventory control.

Kiwiplan first developed over three decades ago by a visionary management and engineering team in a corrugated box plant, Kiwiplan's sophisticated software systems reflect the intimate knowledge of the packaging industry in every menu and function. Kiwiplan systems exhibit special characteristics and options that only industry participants would be aware of.

With a true global presence, Kiwiplan continues to be the world's premier leading provider of software for the corrugated and packaging industry [1].

Chapter 2

Distributed Service Profiler

This chapter describes the Distributed Service Profiler currently used by developers in Kiwiplan.

2.1 Background Information

The initial Distributed Service Profiler was created by another BTech student a few years ago. It connects to different services by connecting with specified port numbers. Any methods called by the services to communicate with each other are intercepted and the method information displayed in the profiler window.

The calls made by the various services are stored into nodes and a call tree is built up. This is displayed in the window to show which calls are initiated by another method. Each node displays the information on the method call. This information includes the method name, the average time taken for the method to be completed, the argument and return bytes and the number of hits (number of times this method was called). The time shown on the nodes is the average time taken to process; this is calculated from the total number of hits and time taken.

Leaf nodes are typically where database queries are made and the majority of the time for some longer method calls are due to queries to the database. These nodes are usually where the time taken to process is much higher. The methods which take much longer than others is what the developers are interested in as these usually indicate the problem areas of the system. However, since no information is given on the actual database calls, it is hard for developers to improve the code without going into the database query logs.

An example of the distributed service profiler running is shown in (Figure 2.1).

Each service is represented with a different colour. Options on the right of the window allow the user to connect to different services by entering the desired port number. The call tree is built as calls made between services are captured and each service method node is coloured in the colour of that particular service. This colour is dependent on the connections on the right and extra connections can be added.

2.2 Related Work

A profiling tool is a program typically used in software engineering or computer science for optimization tasks to run a performance analysis on an application. "A profile of the program's dynamic behaviour under a variety of inputs is presented by the profiler and represents the program's behaviour from invocation to termination" [2]. Profiling is important for understanding program behaviour. It can be a statistical summary of the events caused by the application, a stream of recorded events or an on-going interaction with the virtual machine manager.

JProfiler is an award-winning all-in-one Java profiler. JProfiler's intuitive GUI helps you find performance bottlenecks, pin down memory leaks and resolve threading issues [3]. It is a commercial Java profiling tool developed by ej-techonologies and can be used as a stand-alone application or an Eclipse plugin to analyse Java code.

Most profilers (like the JProfiler) typically analyse the performance of applications to find bottlenecks and memory leaks for that particular program. They usually analyse method calls, memory usage, runtime and frequency of calls and plenty of other issues to test a program's performance from start to termination.

The Distributed Service Profiler used by Kiwiplan analyses performance on a higher level to check service calls made between programs, not within.

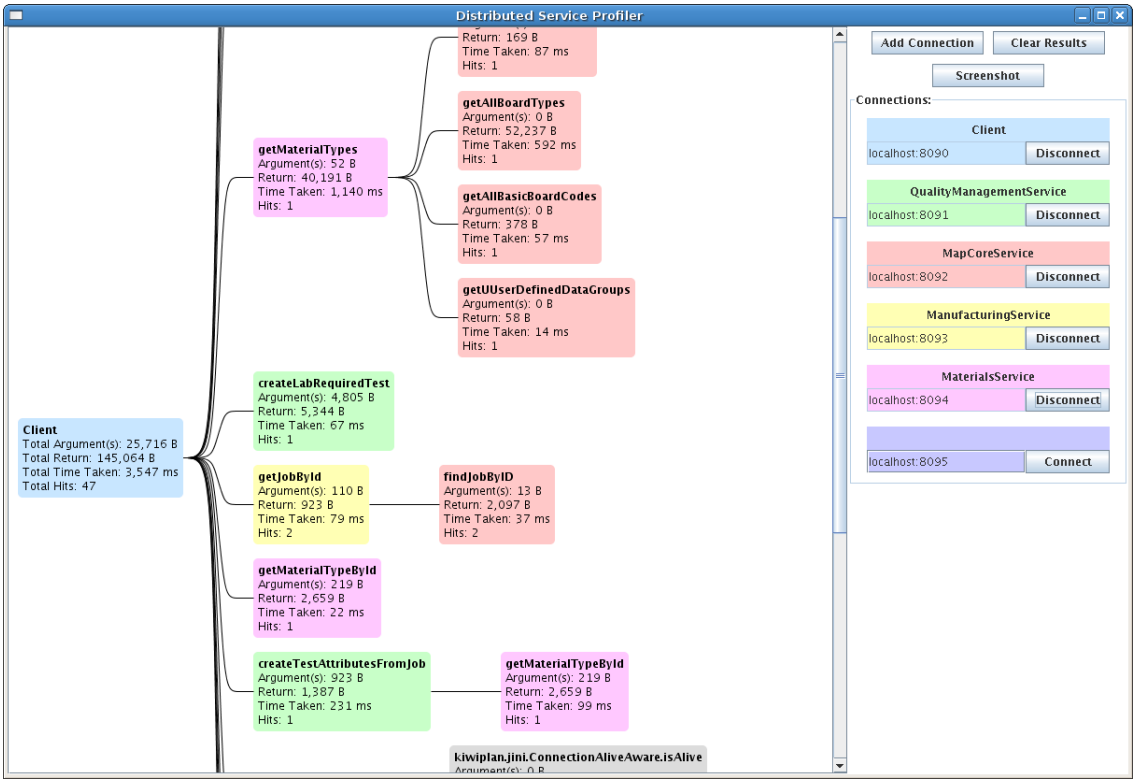

Figure 2.1: Distributed Service Profiler Screenshot

Chapter 3

Development Plan

This chapter describes the development plan I created which includes specifications and details of the project.

3.1 Solution Overview

The final solution should be an extension of the Distributed Service Profiler which can analyse queries made to the database and display the details on screen. The current profiler intercepts calls made between services and shows the details of each method call. This needs to be extended to calls made by the methods to the database and display their details as well as this seems to be where most processing time is spent.

3.2 Requirements

The aim of this project is to extend the existing Distributed Service Profiler so developers are able to find any bottlenecks that exist in the service, which could be in the method calls between services or calls made to the database. To achieve this, there are several requirements to be fulfilled.

The first requirement is that the extension must be able to successfully intercept calls made from the service to the database and extract the necessary information from them. This would obviously need to be done during runtime such that calls are captured as they are made by the services.

Another requirement is that the captured database calls must be correctly matched to the service method calls. This is an essential part of the project as it is an extension of the existing profiler so information from this part must be integrated with the current method calls. A major part of the motivation behind this project is that the current profiler displays the service method

calls but they are unable to see where the database calls are being made. With those calls being the ones which take up the most of the runtime, it is important to see which service methods are making these calls and therefore find out why some methods take such a long time.

Since this project is an extension of an existing application, it is important that the overall design remains consistent. This would help to avoid any errors as well as making it easier for maintaining. The extra implementation also shouldn't affect the how the existing profiler runs. It would be extremely unproductive if the new database interception causes the original method interception to not work properly.

The intercepted database calls will then need to be displayed as part of the current method call tree. This would involve first matching the database call to the original method call, and then creating new nodes to ad to the call tree.

There were also some extra specifications which could be included to improve the existing profiler. They were optional and suggestions made to possibly improve the current profiler.

Change current information in call nodes to display Average Time (currently Time Taken) and Total Time
Highlight the largest argument bytes for each call - show top 3
For long argument lists, show the largest argument (most important)
For database calls, display name, time, size, number of hits,
round-trip-time, SQL/HQL
Order nodes by time - could try ordering nodes or changing the tree edges of the longest three to a different colour
Search nodes in call tree by method name

3.3 Architecture

The current profiler will be extended and will use an interceptor to log calls made to the database. This information can then be displayed in the profiler. A Hibernate interceptor can be used to get the information from the database queries. This can be an extension of the EmptyInterceptor so only necessary methods are implemented. The profiler will need to configure the environment to use the Interceptor. The work done by the interceptor should be done on the server side. Database query information will be passed back and displayed in the call tree by the profiler.

The Interceptor part of this project will use MBeans to communicate with the current profiler. This will allow the Distributed Service Profiler to listen

for notifications from the Interceptor side. The profiler will act as the agent and the Interceptor as the server.

3.4 User Interaction

The GUI of the new profiler will be the same as the current profiler. It will have options to connect to different services and listen for calls made between the services and display the details on screen. It will also display calls made to the database from the services in the same window as an extension to the current call tree. A mock-up of the new GUI is shown in (Figure 3.1) and (Figure 3.2) which shows how the window will change once a database query node is selected.

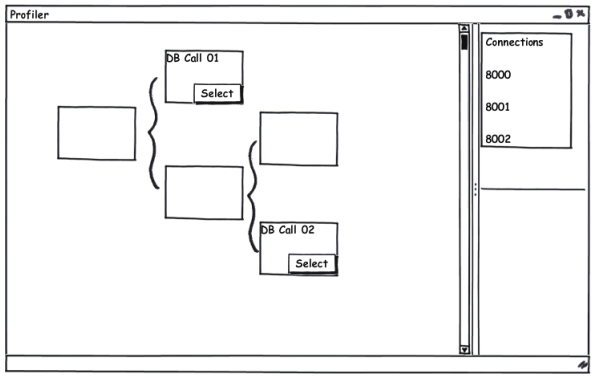

Figure 3.1: GUI Mock-Up Before Selecting Node

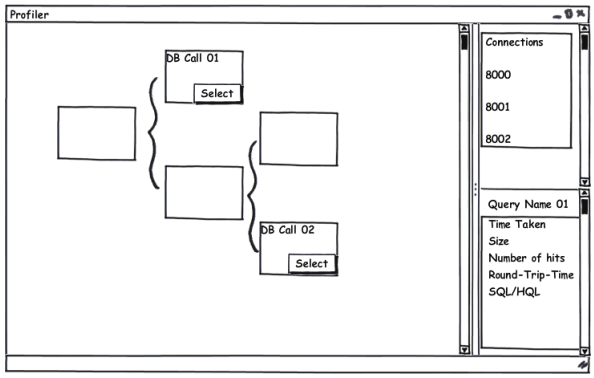

Figure 3.2: GUI Mock-Up After Selecting Node

Queries to the database will contain different information compared to

calls made between services. There could be extra information on the tables, attributes, values and other information. This extra information could be shown on the side of the window, in the space under the different connections (Figure 3.2). There will still be a node for each query which will be included in the call tree. These database query nodes will include a button which allows the user to click to show the extra details on the side of the page.

This is better than including all the information in the call tree as it could result in the tree becoming too large and database query nodes taking up too much space. This will keep the tree looking tidy and allow the user to choose which query they want to look at.

Another idea to display the extra information was to allow the user to select a node they want to see and that node would expand to display any extra information. However, expanding the selected node would mean the call tree would need to be re-computed and redrawn each time a node is selected.

Scrolling may be enabled in the different frames if the information can't all fit into the given space.

Chapter 4

Research

This chapter describes the research I have done for this project. Each section includes research on different areas and topics which helped me to understand the project and get an idea of how to carry out the final implementation. Each section includes explanations on a project or framework which I will be using. Individual classes will be discussed further in the next chapter.

4.1 Distributed Service Profiler

The first step of the project involved me looking into the source code of the existing profiler to find out how it works. This is essential as I will need to eventually extend this to include the database call interceptor.

A ServiceProfilerAgent(MBean) is used to send notifications to the profiler. This class is an MBean which allows several of its attributes and methods to be exposed. Two particular methods which are useful are the sendEntryNotification and sendExitNotification methods. These methods are uses to emit a notification which the Distributed Service Profiler can use. These notifications contain the information on the method calls such as the method name, time taken and size.

The JXMClient class in the profiler is used to query the server for the ServiceProfilerAgent MBean. When the correct MBean is found, a NotificationListener is added to the ClientListener class. The profiler uses a NotificationListener from the javax.management package. This listener is notified when a JMX (Java Management Extensions) notification occurs. When a notification is received, the ClientListener will pass this on to the ProfilingManager, where the majority of the work is done.

The profiler then takes this Notification object to extract the message from it. The message is then split so information can be extracted. This information includes the method name, the service name, argument bytes, return

bytes and the time taken. The hit count increments every time the same process is called.

All of this information is passed to the MethodCallContainer class which helps store this information for each method call. A DefaultMutableTreeNode containing a MethodCallContainer is created for each method call. The ResultsPanel class can then grab the information from each call to create a method call node and paint this component onto the profiler window. To keep track of which service calls which method, a method setInvokingIndex(i) is used to set which connection invoked that method, this is essential in building up the method call tree.

There is of course a lot of other processing involved in connecting to the services and building up this method call tree. However, the main technique used to get the information from these methods is described above. This is the main part of the program which I will be dealing with and will be using ideas from this to create my part. To try to keep things consistent, I would like to have my interceptor part integrated with the profiler and use a similar design.

4.2 Hibernate

Historically, Hibernate facilitated the storage and retrieval of Java domain objects via Object/Relational Mapping. Today, Hibernate is a collection of related projects enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object/Relational Mapping [4].

Hibernate is free and provides a framework to map Java classes to database tables. This means the object-oriented classes of a Java program can be mapped to traditional relational model databases. This makes it possible to map Java data types to SQL data types. Hibernate can also be used for database queries by generating the SQL code. This makes it easier for developers as they don't have to manually change the data types and can easily create applications which are supported with SQL databases.

In the case of the services created by Kiwiplan, Hibernate also helps to create query strings when database calls are made and translate these to traditional MySQL queries.

4.3 Hibernate Interceptor

The Hibernate Interceptor is an interface in Hibernate which can be used in applications to react to certain events occurring inside Hibernate. This is the

main tool I looked into and is what I will be using to extend the Distributed Service Profiler as it can be used to intercept calls made to a database.

The Hibernate Interceptor can be configured with an application and a database to intercept calls made to that database by the application. There can be two types of interceptors, global or session-scoped interceptors. Global interceptors are application-scoped. This means that when an application has several database sessions, the interceptor will affect objects in all sessions. Session-scoped interceptors, like the name indicates, are configured for each individual database session so they will only affect objects associated with that particular session.

Once an interceptor is configured and created, it will be invoked every time a insert, delete, update is made to the database. Methods in the interceptor can be modified to do different things based on the query. For example, if a user tries to insert an object into the database, the interceptor could be created such that it will check the properties to be saved to see whether they are valid. Another example could be for updates to the database, a log could be kept so each update is recorded to keep track of changes in the database.

There are three interfaces related to Interceptors are available in Hibernate, the Lifecycle and the Validatable interfaces and the Interceptor from the org.hibernate package [5].

The Lifecycle interface is used to encapsulate an objects phases of its lifecycle. Methods available from the Lifecycle interface include onLoad(), onSave(), onUpdate() and onDelete(). The Validatable interface only has a validate() method which is called during save operations to check the validity of the state of the object. The final Interceptor interface is what I am interested in.

4.4 EmptyInterceptor

The Interceptor interface is part of the org.hibernate package and includes over 15 methods. The EmptyInterceptor class implements the Interceptor interface and it typically what developers extend to implement their own customized interceptor. This allows the developer to implement only the necessary methods. Methods implemented in the interceptor are automatically invoked when a particular event occurs in the application.

A full list of the methods available from the Interceptor interface can be found on the Interceptor API page [6]. The methods I will be using include the onSave(), onDelete(), onLoad(), onFlushDirty(), afterTransactionBegin() and afterTransactionCompletion() methods. The onSave(), onDelete(), onLoad() and onFlushDirty() methods are invoked when query to the database saves an object, deletes an object, reads an object data and updates an object

respectively. The afterTransactionBegin() and afterTransactionCompletion() methods are invoked when a transaction begins and finishes so I will be using these methods to find the overall transaction time.

I know in Java there is a method System.currentTimeMillis() which return the current system time in milliseconds. This can be used in the afterTransactionBegin() and afterTransactionCompletion() methods and the difference taken to get the transaction time in milliseconds.

For a Hibernate Interceptor to work there must be at least two xml files to configure the interceptor. A hibernate.cfg.xml file is needed to configure the session factory with properties. These properties include the connection driver, URL, username and password and other properties which can set the logged SQL to a formatted output. This file will also specify the mapping files which are used to map the Java classes to the database tables. This mapping file will be named [mappingClassName].hbm.xml and specify the attribute names and types.

I chose to use this to create my final solution as it is fairly straightforward to implement and can easily intercept database calls. It will take some work to integrate it with the current profiler but using the Hibernate Interceptor will make it easier to take care of the query interception part of the project. I will be extending the EmptyInterceptor class as I will only need a few specific methods and this will make it more flexible for me to choose which methods to include and which to ignore.

4.5 Hibernate Event System

"If you have to react to particular events in your persistence layer, you can also use the Hibernate3event architecture. The event system can be used in addition, or as a replacement, for interceptors." **??**.

This would be an alternate way to intercept database queries. Event listeners would be used instead to listen for events generated by the Hibernate session. This implementation is considered the newer version of the interceptors and can be more robust flexible in terms of events that can be listened to. The event listeners have a different set of methods that can be implemented compared to the interceptor, but overall they serve a similar purpose. Event listeners are actually used in Hibernate to invoke any interceptors implemented by developers.

I chose to use the interceptor for this project as it seems like the simpler approach and has the methods I would need. Since the goal of this part of the project is to simply intercept a database query and pass on any relevant information, the simpler approach would be the preferred option. The event system is simply an alternate design choice, one to keep in mind if any future

development and testing shows that the interceptor is too limited for the application.

4.6 Logging

I also looked at how HQL (Hibernate query language) and SQL could be logged. I found something called SLF4J (Simple Logging Facade for Java) which serves as a simple facade or abstraction for various logging frameworks [7]. SLF4J is what Hibernate uses for logging SQL. However, it should have a binding framework such as Logback or Log4J.

I compared the difference between Logback and Log4J to see which binding I should use. Logback is basically the new version of Log4J and it seems pretty straightforward to use. I initially decided to use Logback with my interceptor. However, Kiwiplan typically uses Log4J so I decided to use Log4J for my final logging interceptor to keep things consistent.

Log4j is a tool used by developers to output log statements so that problem areas can be located. Logging behaviour can be controlled by using a configuration file and may be assigned to levels. The possible levels are TRACE, DEBUG, INFO, WARN, ERROR and FATAL. Since these levels are ordered, if the logging level is set to INFO, then log requests at or above this level (INFO, WARN, ERROR and FATAL) will be enabled and the log statements printed. Levels below the set logging level (in this case, TRACE and DEBUG) will be disabled and log statements of that level are not printed.

I am currently unsure about logging for HQL; it is hard to find sources which describe how HQL can be logged. Most articles I found were for SQL logging. I will need to do further research for this.

4.7 Hibernate Query Language (HQL)

I have used SQL queries before but I had no previous experience with HQL so I had to do some research on that. HQL is similar to SQL but is fully object-oriented and understands notions like inheritance, polymorphism and association [8]. It is used to write queries which are similar to SQL queries but for Hibernate objects. I found that HQL is much easier to use than SQL. A simple example of HQL is ”from Contact”. This short query will return all instances of the class Contact. Compared to SQL, where the same query would need to be ”Select * from Contact”.

HQL also has a feature called criteria, which can be used to set restrictions to a query to select specific rows or attributes from a table. This makes it similar to object-oriented programming and can help make code neater when

using long and complex queries which would require many ”where” and join conditions if traditional SQL was used.

4.8 Java MBeans

MBeans are managed beans, Java objects that represent resources to be managed, and have a management interface [9]. MBeans represents objects or resources in Java and implement getter and setter methods for attributes and properties of that object. The special management interface associated with an MBean allows access to attributes, operations that can be invoked, notifications that can be emitted and constructors for that object. There are also five types of MBeans - Standard, Dynamic, Open, Model and MXBeans.

Using the class name plus ”MBean” is the typical format of creating an MBean to expose certain attributes and methods. This is how the Standard MBean is created. Dynamic MBeans can publish a description for each of its exposed methods (compared to Standard MBeans which can't). This makes it easier for users as they can see what existing MBeans can use and do. The Open and Model type MBeans are both dynamic types. Open MBeans follow some conventions, making them less powerful and convenient but portable. Model MBeans are not done in a typical Java class, but instead created by using a Model MBean from the JMX and modifying it. Lastly, the MXBean is a special type, and references a predefined set of data types. This makes them usable by any client as no special configuration is required.

MBeans can be used to invoke operations on the application being monitored, as well as receive notifications about events. This is what the current Distributed Service Profiler does. A property change listener can also be implemented to detect changes made to a property of an object.

I will be using the Standard MBean, as I will only need to use it for communication with the profiler and the ability to emit notifications. The Standard type will be the simplest one to use and is what the ServiceProfilerAgent(MBean) extends. Using the same type means I can use a similar format and keep the design consistent with the existing profiler.

To actually be able to use an MBean, it must first be registered to an MBeanServer. This is a repository of MBeans which a client can later browse. To be registered, MBeans must be given a unique object name. This allows clients to be able to query the server to find a specific MBean by supplying some criteria such as an object name.

4.9 Notifications

Once an MBean is created and registered, it can be used to emit notifications. Notifications can be used to signal many things such as changes, events or problems. Notifications all contain a source (the object name of the MBean) and a sequence number (used to order notifications from the same source). They can also contain a timestamp and a message (which is simply a String type). They can also contain other attributes depending on the type of notification they are. There are 7 main types - AttributeChangeNotification, JMXConnectionNotification, MBeanServerNotification, MonitorNotification, RelationNotification, TimerAlarmClockNotification and TimerNotification.

For an MBean to emit a notification, it must implement the NotificationEmitter interface or extend NotifcationBroadcastSupport. A notification instance can then be created with the appropriate parameters and the method sendNotification() can be used to emit the notification.

The client application can then use a class to implement the NotificationListener and use the handleNotification() method to receive notifications. Once a notification listener is added by calling addNotificationListener with the client listener and specific MBean on the MBeanServer, the client can start receiving notifications from the MBean. The handleNotification() method will then be invoked every time a notification is emitted from the MBean.

4.10 JConsole

"JConsole is a JMX-compliant GUI tool that connects to a running JVM, which started with the management agent." [10]. It includes a graphical interface to allow easy monitoring of the Java Virtual Machine (JVM) and both local and remote applications.

Once connected to a specific process, there are six tabs in the interface to look at different information from that process - Overview, Memory, Threads, Classes, VM Summary and MBeans. The MBeans tab can be used to look at the information on all MBeans registered in the target JVM and allows access to the full set of the platform instructmentation. Information on each MBean can be displayed such as the exposed attributes and operations as well as allowing the user to use the MBean to emit a notification. Exposed operations will allow the user to invoke methods to test how they run. Emitted notifications can also be stored for users to see.

This MBean information page is what I used when testing my LoggerManagerMBean (described in a later chapter). It is extremely helpful in seeing whether the MBeans are being registered and emitting notifications correctly.

4.11 Database Changes Logger

The Database Changes Logger is part of the Hibernate project and is a tool developed another BTech student a few years ago used to record changes to Hibernate databases used in Kiwiplan services. It uses JMX MBeans and the Hibernate Interceptor to intercept database queries and is similar to what I need to create to extend to the Distributed Service Profiler.

The interceptor in this changes logger implements onSave(), onDelete() and onFlushDirty() and writes the changes to a log. The logging level can be changed to log a different amount of detail for each query (NONE, BASIC and FULL). This changes logger also uses LoggerManager class which manages the interceptor and creates it.

This changes logger will use configurations files to configure Hibernate if they are provided, otherwise default values are used. The logging levels to choose from are NONE, BASIC and FULL. If NONE is used, nothing will be logged. BASIC means all save and delete operations will log a single line with the username, action, objects id and the objects toString() method result. Updates are only logged if there were actual changes. At the FULL logging level, all properties will be shown for save and delete operations and for updates, the object will be logged even if there are no changes made to it.

This changes logger uses the service name passed into the LoggingManager to associate each different logging manager with a particular service. This service name is then used to create service properties class which stores the properties for a service and provides methods for retrieving them. This class can then be used by the interceptor.

The LoggingManager class registers itself as an MBean, this means the logging level can be changed after initialisation using a JMX management application. Logged messages will be written to the appropriate log, specified by the configurations in the log4j.properties or log4.xml file.

4.12 Jini API

Jini API is one of the projects that the service profiler requires to run. There are three particular classes in this project that the service needs which I looked into - MBeanCreator, ServiceProfilerAgent and ServiceProfilerAgentMBean. One of the things that confused me as to how the profiler works was that I initially though the profiler was only contained in a single project. Looking into these three classes helped me to understand how the profiler runs and where the notifications came from.

The ServiceProfilerAgent(MBean) is the class used to emit notifications containing the method call information. These notifications are then used by

the profiler to build up the method call tree. The MBeanCreator is used to connect to the service using the specified port number. A ServiceProfilerAgent(MBean) is then created and registered to a specific MBeanServer for the profiler to find.

4.13 Maven

"Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information." [11]. It helps in providing a uniform build system as well as quality project information. It can be used for any Java-based project.

Maven is the tool that Kiwiplan uses to build and manage Java projects. It allows a project to be imported easily into Eclipse and makes building the project simple as dependencies are managed. Since this was all set up, I could simply run Maven from shell to build a project and get a jar file from the build.

Chapter 5

Design/Implementation

This chapter describes what I have actually implemented, including experiments I did to test parts of the code, the classes which will be combined with the Distrributed Service Profiler and any modifcations made to existing classes.

5.1 Simple Hibernate Interceptor

To experiment with a Hibernate Interceptor without having to integrate it with the service profiler, I implemented a simple Interceptor class which extends the EmptyInterceptor. In this Interceptor, I implemented the onSave(), onDelete(), onLoad(), onFlushDirty(), afterTransactionBegin() and afterTransactionCompletion() methods. At first, the onSave(), onDelete(), onLoad(), onFlushDirty() methods would only print out whether it was saving, deleting, updating or loading an object. This was just to let me see whether the interceptor was working and whether the methods were being invoked at the right time.

I also created the hibernate.cfg.xml and mapping file to configure the interceptor and map it to the MySQL table. To test this interceptor, I created a Contact class in Java which has attributes id, first name, last name and email. I also created a main class HibernateTest which configures the session factory and creates an application-scoped Interceptor. In here, I created a few Contact objects and saved, deleted, updated and read them from the database to check if the Interceptor worked.

The hibenate.cfg.xml file contains information on the properties needed to configure the session factory for Hibernate to use. These properties include the connection driver class, connection URL, dialect, connection username and password and the names of the mapping files. There can be a lot

of other properties to configure the session factory but many of them are not necessary for the program to work. The properties I used in this simple interceptor test were the minimal amount needed to get it working. Once I start to integrate the real interceptor with the Distributed Service Profiler, I will need a lot more information. I expect most of this information can be obtained from Kiwiplan as they already use Hibernate for their code.

The same can be said for the mapping files. Since mapping files include information to map the attributes from a Java class to a database table, I would expect I can get this information from Kiwiplan otherwise it would be extremely difficult for me to create one from scratch. These were some of the initial problem I encountered when implementation fo the actual extension started. These issues will be discussed more in section 5.3.

I also added a SLF4J with Logback into this application and tried the different logging levels. I used this in the Interceptor class to print out when a method is invoked as well the entity class (in this case, Contact) and the id.

Up until this point, the afterTransactionBegin() and afterTransactionCompletion() methods were empty. I added a System.currentTimeMillis() in afterTransactionBegin() and stored this in a global variable. When afterTransactionCompletion() is invoked, System.currentTimeMillis() is called again and the difference is taken between the returned value and the stored time from the beginning of the transaction. This is the total transaction time in milliseconds.

Once the Interceptor was working with the saves and deletes, I decided to test it with some more complicated queries. I had only been using session.save(c) or session.delete(c) to test it so far. I used some HQL and SQL queries to select certain contacts from the Contact table. This included using the HQL criteria and setting parameters into the queries to run on the database. I also created traditional SQL query strings. Both were fine in running on the database and the interceptor successfully intercepts the queries and extracts the information from the queries.

5.2 PersistenceWrapper

The PersistenceWrapper class is a generic interface used by the services for persistence operations. This class can be used to configure the Hibernate sessions, load properties and mapping files and many other operations. There are three getInstance methods which returns a singleton of the PersistenceWrapper. These methods will create a configuration and set any required properties to it. One of these getInstance methods takes in a databaseName, Configuration and serviceName as parameters. Only when this getInstance is used will a DBChangesLoggerManager be created and logging set up for the

DBChangesLogger application. When this getInstance method is called, an interceptor will be set for the configuration using the getInterceptor method from the DBChangesLoggerManager.

A rough overview of how the PersistenceWrapper works is that it creates a session using the session factory. Any Hibernate mapping schemas are read in and added to the configuration and mappings (list of mappings to avoid duplicates). The session factory is re-created after any new mappings are added. A transaction can be started using the given session and Hibernate properties files can also be loaded. These are the main methods required to start the session and perform database queries.

I had started creating a PersistenceWrapper class with only the specific methods I would need for my project. I was also confused as to where I needed to create the PersistenceWrapper in the profiler. I had originally wanted to intialise it when a service is first connected. However, I later found out that the services actually have a PersistenceWrapper in their constructor and I could simply make changes to the existing one to account for my service profiler interceptor.

5.3 LoggerInterceptor for Profiler

This section is split into two parts as the implementation for this class was later revised and changed to create a better overall design of the project. The first section will discuss the initial implementation as well as the testing and reasons that led to the design change. The second part will explain how the final version was implemented.

5.3.1 Original

It was good that I had created the simple Interceptor for testing as it allowed me to understand how the Hibernate interceptor works and reuse the same format and some of the code from that for the actual LoggingInterceptor class.

The LoggingInterceptor is initialised with a service name and the LoggingInterceptor class has the onSave(), onDelete(), onLoad(), onFlushDirty(), afterTransactionBegin() and afterTransactionCompletion() methods.

I also included methods in the LoggingInterceptor which are called when the onSave(), onDelete(), onLoad() and onFlushDirty() methods are invoked. These methods take in the parameters passed into the methods which have information on the entity being passed to the database for saving, deleting, reading or updating. This information can be used in interceptors to validate or change the incoming information before passing it on. I simply take these

parameters to format them and print out the entity being changed and its properties.

As mentioned earlier, I was concerned about how to obtain the mapping and configuration files need to use the interceptor. All of this information is in the PersistenceWrapper so that made things a lot simpler. To make sure I had the interceptor working properly, I took the LoggerInterceptor class and injected it into the PersistenceWrapper for all services to use. I changed the code in the PersistenceWrapper to use this interceptor instead of the DBChangesLoggerManager and Interceptor. I ran into some problems trying to get the interceptor working with the services and after some looking around, it turned out that the services were calling a different getInstance method. I simply set my interceptor in the configuration of this other getInstance method and that fixed the problem. I launched the services and used to the LoggerInterceptor to print out intercepted queries. Figure 5.1 shows some sample output from the interceptor, and it is clear that database queries are successfully intercepted. This was also helpful in seeing what kind of information is being captured and what information I should pass onto the profiler.

Once the LoggerInterceptor was successfully working, I integrated it with the LoggerManager. The LoggerManager is used to initialise the LoggerInterceptor and is passed as one of the parameters into the LoggerInterceptor constructor. The LoggerInterceptor can then use this to get the service name and logging status (to switch logging on and off).

An important part of this project is to be able 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 an accurate call tree. There were several alternate approaches do this. The first was to use the stack trace. However, there was no guarantee this would go as far back as the service calls. If this was true, the method matching would need to be done in another place (not in LoggerInterceptor). One idea was to use the PersistenceWrapper. The stack trace approach would be the simplest as it could all be done in a single class without the need to use something like the PersistenceWrapper, processing it and then passing the information back to the LoggerInterceptor.

The first thing I needed to do was check the stack trace to see what kind of information was in there. To get the stack trace,
Thread.currentThread().getStackTrace() can be used to get an array of StackTraceElements. Each element in this array represents a single stack frame and is a method invocation containing the declaring class, method name, file name and line number. This method was called in the LoggerInterceptors startQuery() method, which is called every time a database call is intercepted.

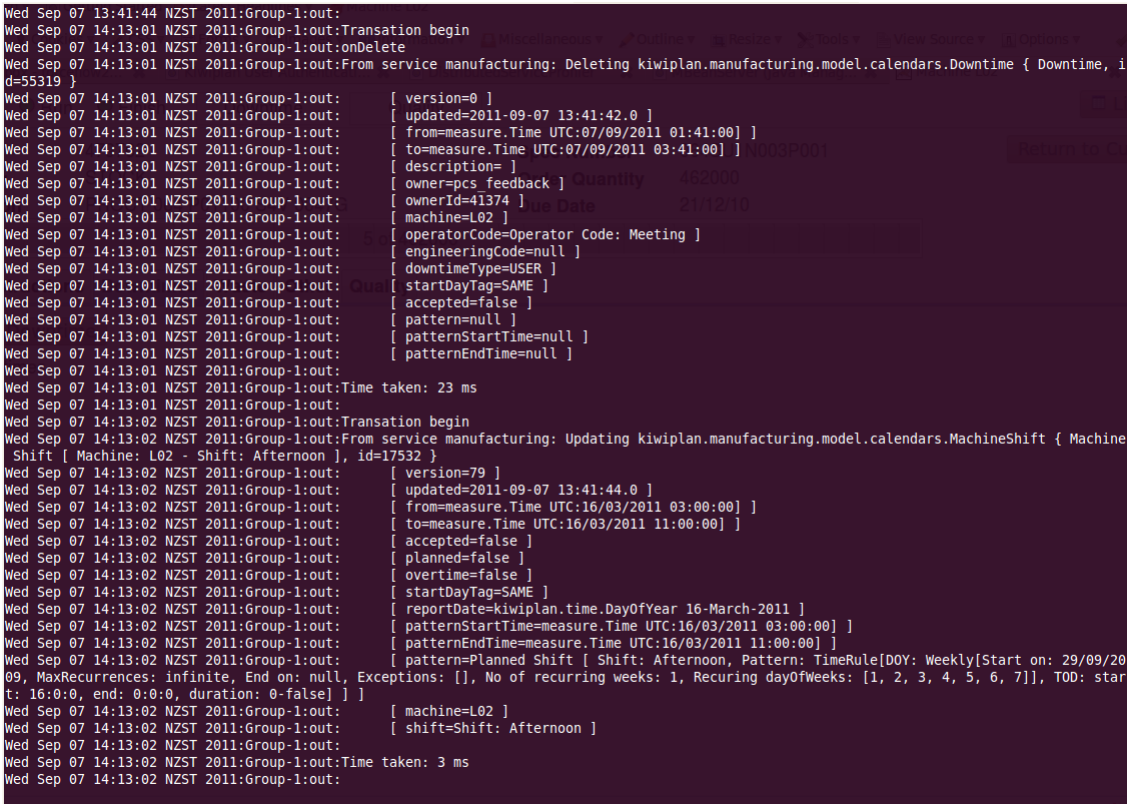

Figure 5.1: Intercepted queries from LoggerInterceptor

As shown in figure 5.2, it can be seen that the stack trace does in fact go back to the original service method call. I have removed some of the output to save space, they were simply methods used by Hibernate such as event listeners which is done for interceptors to work. Typically the methods invoked by classes from the Hibernate package are helper methods done in the background and not explicit calls done in the services. As we move down the stack trace, it can be seen that methods from the PCS service (one of the services created by Kiwiplan) is making calls (runServiceUpgrade, runTask etc) that lead to the database query.

5.3.2 Revised

To keep the dependencies between different projects consistent, the Logger-Interceptor and LoggerManager (explained in section 5.4) were modified. The same methods and concepts are used for the actual interception of the database calls and communication with the profiler.

Previously, the LoggerInterceptor required an instance of the LoggerManager as a parameter in the constructor. This was used to initialise attributes in the LoggerInterceptor such as the service name and logging status. An instance of the LoggingManager is also stored in the LoggerInterceptor to use when emitting notifications to the profiler. As these two classes had to be in two different projects - LoggerInterceptor needed PersistenceWrapper and LoggerManager required MBeanCreator (section 5.7) - it caused some dependency issues. These other classes will be discussed in their respective sections.

The main change the LoggerInterceptor had was that the dependency between it and the LoggerManager was reversed. The LoggerInterceptor was modified so a LoggerManager was not needed for initialisation. Instead, a call is made to create a new LoggerManager inside the constructor and this instance is the one used to send the notifications. This meant no dependencies needed to be changed as this Hibernate project already had a dependency on the Jini API.

The only other change needed to make this work was in the PersistenceWrapper. Instead of creating the LoggerManager and using that to create the LoggerInterceptor, the LoggerInterceptor is created straight from the PersistenceWrapper and set to the configuration.

5.4 LoggerManager for Profiler

This section is also divided into two parts and structured like Section 5.3.

5.4.1 Original

Using the format from the DBChangesLogger project, I created a LoggerManager class and a LoggerManagerMBean class. I initially had just the attributes serviceName and loggingStatus exposed. The loggingStatus is simply a Boolean variable used to switch logging on and off. The serviceName will help to associate the interceptor with a specific service.

The LoggerManager is initialised by passing in a serviceName and a PersistenceWrapper instance. To set the interceptor for the configuration, the getInterceptor method can be called. This will actually create a WrapperInterceptor class before creating a LoggerInterceptor class. They both extend the EmptyInterceptor and implement the same methods (such as onSave, onDelete etc.). The WrapperInterceptor will call the methods in the LoggerInterceptor when database queries are intercepted. This is to prevent any logging errors from halting interaction with the database.

To test out the LoggerManager(MBean), I added a simple main method so that it could run as a standalone application without the PersistenceWrapper. I could then run the application and use JConsole to check that I had the MBean registering properly and was able to switch logging on and off.

After some testing and more research, I needed the LoggerManager to be able to send notifications to the profiler where the query information could be used and displayed in the call tree. I implemented two methods - sendEntryNotification and sendExitNotification - which creates an AttributeChangeNotification and emits it for the profiler to grab. Inside this notification was only the method name, time stamp and message tag. The message tag is simple a string to track whether it is the start of a query or the end. Again I used JConsole to check whether the notifications were emitting properly.

For the LoggerManager to actually be able to communicate with the profiler, the profiler must be able to find the LoggerManager MBean to register a listener. However, I ran into some problems here where the profiler was unable to find the correct MBean, despite the fact that the LoggerManager(MBean) had been successfully registered and I was able to see it using JConsole. After some research, it turned out that because the two applications were separate (LoggerManager in one project, service profiler as another), it meant they were in different JVMs. As I was only using ManagementFactory.getPlatformMBeanServer() and registering the LoggerManager to this MBeanServer, the profiler was unable to find it. To fix this, I needed to register the MBean in the same place as the ServiceProfilerAgent. This would make it easier for the profiler to find as they would appear on the same MBeanServer.

Figure 5.3 shows a screenshot of the JConsole application. The MBean tab is open and it can be seen that the LoggerManager(MBean) has been successful registered into the MBeanServer. On the left hand side, the attributes and operations of the MBean is listed. Currently displayed is the information page, which shows the unique object name of the MBean, class and a short description. The object name is the one given to the MBean when being registered and is what will help the profiler find the correct MBean.

As the MBeanCreator class is used to create and register the ServiceProfilerAgent(MBean), I needed to use the same MBeanServer created in that class. Once I had the LoggerManager using this class to register itself, the profiler was able to find the LoggerManager(MBean) and handle notifications emitted from it.

With the profiler successfully catching notifications emitted by the LoggerManager, the next step was to send information about the queries to the profiler. This was done in the LoggerInterceptor and the query information, including the stack trace was sent to the profiler using the sendEntryNotifi-

cation method (as described earlier).

5.4.2 Revised

Since the LoggerInterceptor was now initialising the LoggerManager (not the other way around), some small changes needed to be made to the LoggerManager class. For one, the method getInterceptor() was no longer needed. This was the method the PersistenceWrapper initially called in order to get an interceptor to set to the configuration.

5.5 JMXClient Modifications

The JMXClient class is used by the profiler whenever the user wishes to connect to a service. This class connects to the JMX server and queries the MBeanServer to find the ServiceProfilerAgent(MBean). Once found, a notification listener is added to the ClientListener (section 5.6) to receive emitted notifications. This is the class in which I added the connection with the LoggerManager(MBean).

As mentioned earlier, I ran into some problems with this part as the profiler had trouble find the LoggerManager MBean. This is because I had originally been using LoggerManager in the Hibernate project and registering it with the MBeanServer obtained from the method getPlatformMBeanServer(). In the JMXClient, I tried to find the MBean by calling getPlatformMBeanServer() and looking in that. While the MBean was registering correctly, it was not showing up on the profiler side. I found that this was due to the classes being in different applications, and essentially different JVMs, therefore different MBeanServers. This is why the LoggerManager(MBean) had to be moved to the Jini API project and be registered to the same MBeanServer as the SerivceProfilerAgent.

After the classes had been rearranged, I could then use the same MBeanServer found in the JMXClient and find the LoggerManager(MBean) by supplying an object name. With this done, another notification listener was added to the ClientListener, but with a specific handback string - ”loggerManager”. This is so the ClientListener is able to differentiate between the notifications received from the original ServiceProfilerAgent and the LoggerManager. I had initially considered creating a new class, similar to ClientListener, for the LoggerManager but combining it with the existing listener was a better design choice as the implementation did not need to be changed.

5.6 ClientListener Modifications

The ClientListener class implements the NotificationListener to receive the notifications sent by the ServiceProfilerAgent(MBean). It contained a single method - handleNotification - which is invoked whenever a notification is received. Once invoked, it will pass the notification along with the associated connection index onto the ProfilingManager for processing in order to build up the method call tree.

The handleNotification() method takes in two parameters - the notification and a handback. The handback can be defined for the listener and MBean when the addNotificationListener() method is called. This was defined as null for the ServiceProfilerAgent but as I needed to be able to differentiate between notifications from the two MBeans, I gave the LoggerManager(MBean) a handback "loggerManager". Again, the connection index is passed along with the database query to the ProfilingManager as this will be needed when building up the method call tree.

When handleNotification() is invoked, I included a check for the handback string and based on that, passed it onto the ProfilingManager with different methods.

5.7 MBeanCreator Modifications

The MBeanCreator class is a part of the Jini API project and is used to create and register the ServiceProfilerAgent(MBean). It first connects to the service using the provided port number and then creates an MBeanServer (if one hasn't been defined) and registers the MBean to it.

The only change needed to this class was an extra method for registering the LoggerManager(MBean). Since the LoggerManager was initialised by the LoggerInterceptor, the only thing needed was the same instance of the MBeanServer. The LoggerManager can then simply call the method upon initialisation, and pass itself and an appropriate object name along with the method. In MBeanCreator, a MBeanServer will be created if it hasn't already, and the method mBeanServer.registerMBean(loggerManager, mbeanName) can be used.

5.8 ProfilingManager Modifications

The ProfilingManager class is where the majority of the work in the profiler is done. This class is responsible for processing the information received from the service method calls, creating tree nodes for each one and then building

up a method call tree. It first takes the notification from the ServiceProfiler-Agent and checks the message tag to see what kind of method it was. This could be invoked, called, returned or completed and helps to build up the method call tree. Depending on the message tag, the notification will then be passed on to a different method and processed. This will involve creating tree nodes containing a MethodCallContainer with all the method information inside.

To integrate the database information, I created a new method queryNotificationReceived() which is called by the ClientListener whenever a database query is intercepted. In here, the notification can be passed onto different methods depending on the message tag - queryStart or queryEnd. With the stack trace contained in the notification, the matching can be done to find the initial service method call that invoked the database call.

Since the current profiler only displays information about the method calls between the services, the database calls would original from the leaf nodes of the call tree. This is where some methods have a much longer runtime, since it causes calls to the database to be made, that time is included in the service method. As the current profiler stores each call node in a list, with one of them being the open nodes, I can simply use that list and compare the method names of each node with the method names in the stack trace.

5.9 Matching Service Calls with Database Queries

As explained in the previous section, the stack trace was used to match the methods from the list of open nodes to the method names in the stack trace in order to find the corresponding service method call to the intercepted query. Before this was implemented, it was uncertain whether that woudl be an appropriate approach as the stack trace may not have contained all the information and gone back to the point of the original method call.

The basic flow of the database calls is that a service will invoke a database call by using some HQL statement. This HQL in the PersistenceWrapper uses Hibernate to convert the method into a traditional SQL statement and then passed onto the database. The alternative approach that could have been used would have involved using the PerisistenceWrapper and grabbing the information from the HQL to find the original service method. While this approach would definitely work, it would have involved a lot more difficult compared to using the stack trace. The intercepted query would need to be used along with PersistenceWrapper and after being processed, the information would need to be passed back through the LoggerInterceptor to the profiler.

```
java.lang.Thread
getStackTrace
kiwiplan.persistence.servicelogger.LoggerInterceptor
startQuery
kiwiplan.persistence.servicelogger.LoggerInterceptor
onFlushDirty
...
org.hibernate.event.def.DefaultFlushEntityEventListener
handleInterception
...
org.hibernate.transaction.JDBCTransaction
commit
kiwiplan.persistence.PersistenceHelper
update
kiwiplan.pcs.dao.HibernateLineupEntryDAO
store
kiwiplan.pcs.service.upgrader
.StepEstimatedQuantityCalculationUpgradeTask
runTask
kiwiplan.pcs.service.upgrader.PcsServiceUpgraderImpl
runServiceUpgrade
kiwiplan.pcs.service.PcsServiceFactory
createAndrunPcsServiceUpgrader
kiwiplan.pcs.service.PcsServiceFactory
createServiceImpl
kiwiplan.pcs.service.PcsServiceFactory
createServiceImpl
...
```

Figure 5.2: Example of stack trace using getClassName and getMethodName

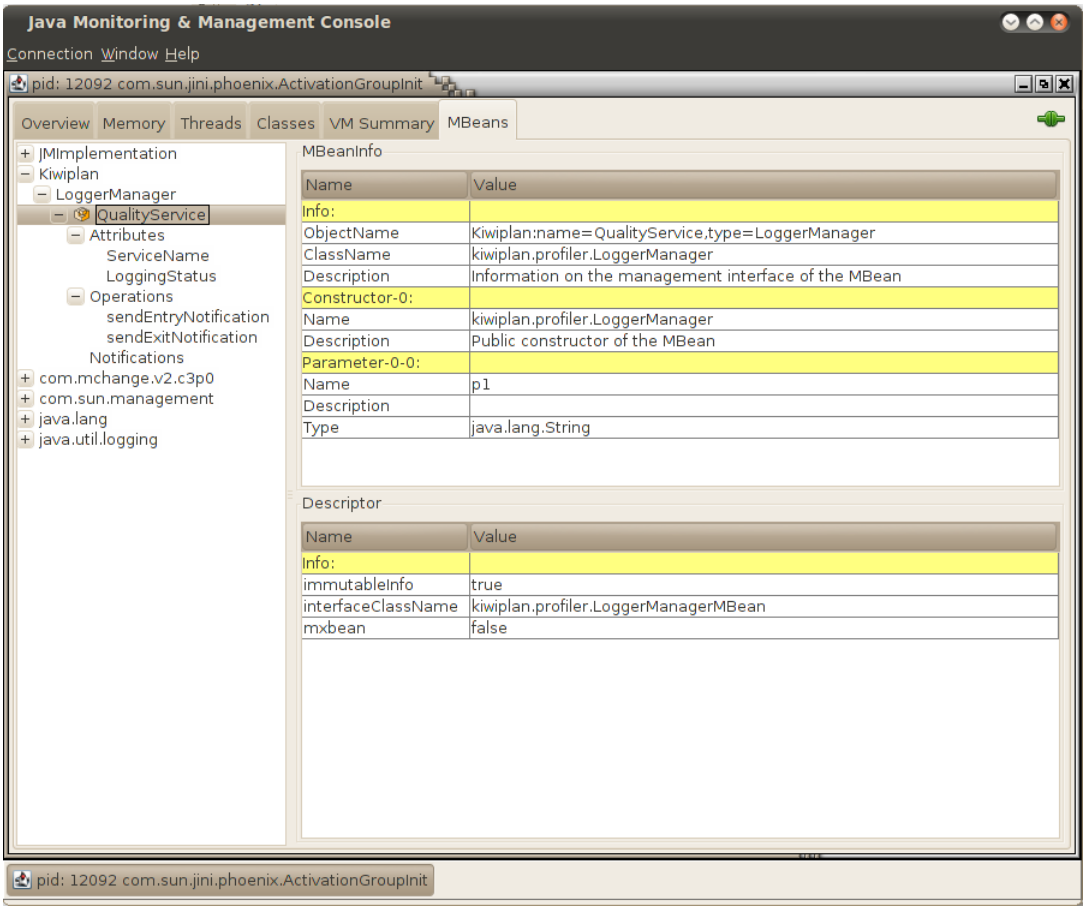

Figure 5.3: JConsole View of MBean

Chapter 6

Final Solution

This chapter gives an overview of the final solution and future work.

6.1 Solution Overview

The final solution of the project will follow a similar design of the current profiler. Figure 6.1 shows a high level view of the solution, and how it fits in with the existing profiler. The red boxes show the main changes made to the overall profiler. When one service makes a method call to another, that method is intercepted existing profiler. When a database query is made, it will be intercepted in the Hibernate project and passed onto the JIni project. In here is the MBean used to communicate with the profiler, and a notification can be sent containing the query information.

As the original Distributed Service Profiler has a dependency on another project (Jini API), I ended up modifying my original design to keep dependencies consistent. My original solution of the interceptor portion required classes from both the Hibernate (Kiwiplan project, not the framework) and Jini API. As explained in earlier chapters of this report, the LoggerManager(MBean) must be registered to the MBeanServer and the Interceptor must use the PersistenceWrapper to be configured with the services. As there was already a dependency on the Jini API from the Hibernate project, this created a circular dependency, something that could cause problems with the other projects in Kiwiplan.

The cause of this circular dependency was that the LoggerInterceptor needed the PersistenceWrapper (Hibernate), the LoggerManager needed the MBeanCreator (Jini API) but they also needed each other. My initial idea to solve the errors I ran into while building the solution was to add a dependency into the Jini API to use the Hibernate project. While this helped me

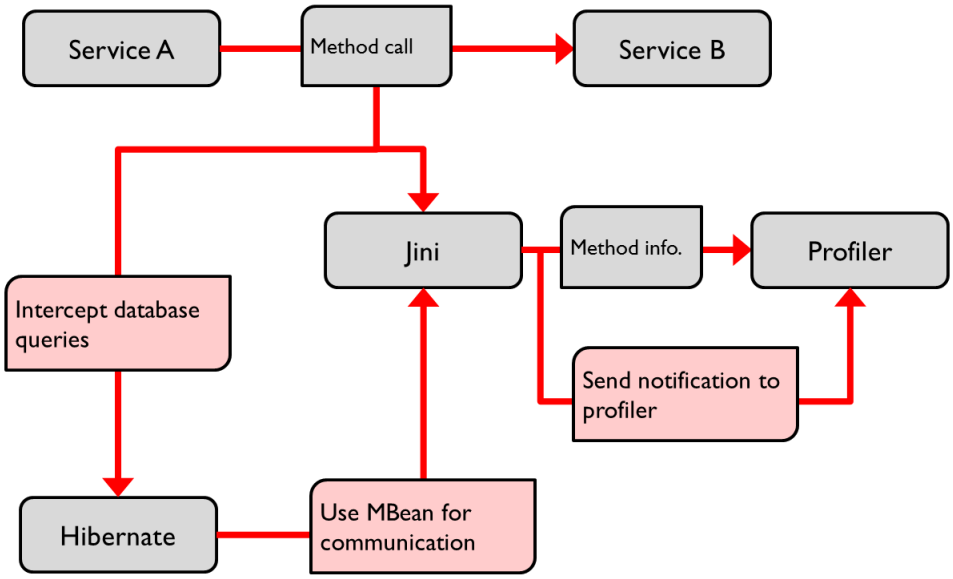

Figure 6.1: Solution Overview

get the project running without errors, it would affect the existing projects Kiwiplan develops once committed to the system.

To fix this dependency problem, I changed the way the LoggerInterceptor and LoggerManager were created. My initial approach followed the design of the Hibernate project and had the PersistenceWrapper create an instance of the LoggerManager, which would then create an instance of the LoggerInterceptor. Since these two classes had to be placed in different projects - the interceptor in Hibernate and manager in Jini API, I switched the way they were initialized. As the LoggerInterceptor uses the LoggerManager(MBean) to emit notifications, there was no need for the LoggerManager to have an instance of the LoggerInterceptor. The new solution has the PersistenceWrapper create the LoggerInterceptor (both in Hibernate), which then creates an instance of the LoggerManager (in Jini API) for communication with the profiler.

Figure 6.2 shows the final solution of this extension to the profiler. This is a lower level view of the solution and shows the main classes used in each stage. Each service is constructed with a PersistenceWrapper which will then set an interceptor into the configuration. When the LoggerInterceptor is initialised, it will create an instance of LoggerManager. In this LoggerManager, it will register itself on the MBeanServer using the class MBeanCreator - the

same class which is used by the profiler to create and register MBeans. This will keep them registered on the same server, making it easier for the service profiler to find. The profiler can then add a notification listener for the LoggerManager. When the profiler finds the LoggerManager(MBean), it will be associated with the connection index from this instance of the ClientListener. This will help to keep track of where the methods are intercepted from and help with the later parts of the project where the database queries need to be integrated with the service method calls. With this design, I did not need to add any new dependencies to the projects.

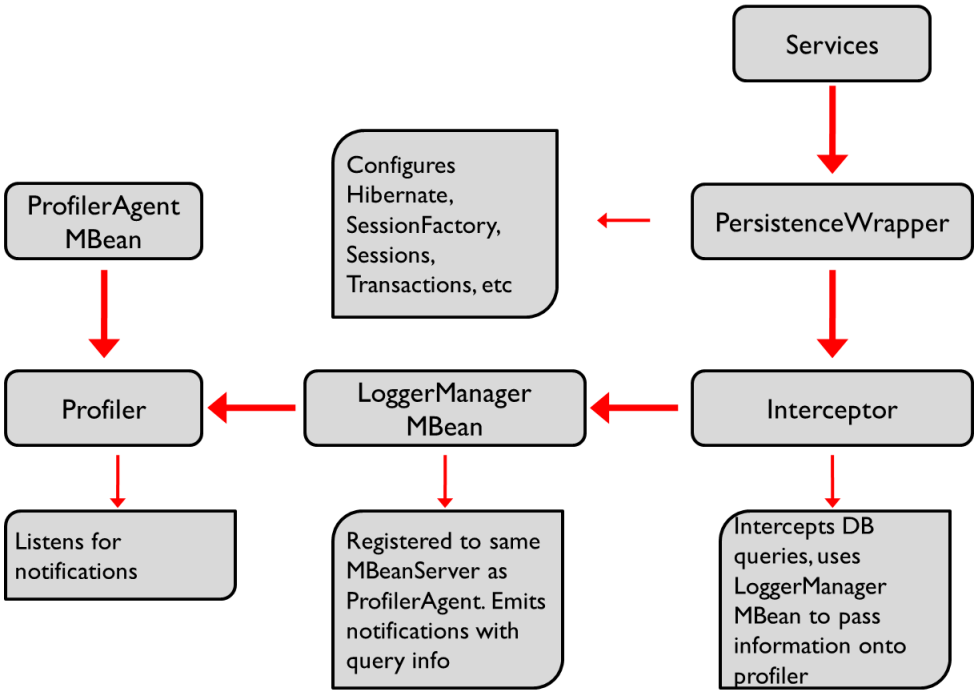

Figure 6.2: Solution Overview - Low Level

The design of this solution is effective and addresses most of the requirements stated in chapter 3.2. It can successfully intercept calls made by the services to the database and this was shown with the LoggerInterceptor testing (figure 5.1). This was all done using the Hibernate interceptor and done during runtime as the calls were made by the services. The stack trace of the can then be used to do the matching between the intercepted database query and the original service method calls.

This design also followed the original profiler design, using MBeans and

notifications to communicate and using the connection index to keep track of the method calls. No changes were made to the original source code. I only added new methods or sections to the source code, but no existing code was modified or removed. This was to ensure the original profiler would still work.

The requirements I did not manage to address were displaying the new database calls and the extra specifications. This was due to the implementation part of the project being incomplete and is explained in the next section.

6.2 Future Work

Unfortunately, I did not complete all the implementation of the design so there are still a few missing parts to this profiler extension. Currently, the LoggerInterceptor is successfully configured with the services and able to intercept the database queries made by those services. The profiler is also able to query and find the LoggerManager(MBean) and listen for notifications emitted by the LoggerManager. This notification contains the stack trace of that thread up to the point of query interception. The profiler can then compare the current node of the service method calls with the methods called in the stack trace to find which method is the cause of this database query - something that is essential to build up the method call tree.

The parts which still need to be implemented include creating a node for the database query. This could involve creating a new class which extends the MethodCallContainer class in the profiler. The MethodCallContainer is simply a class containing all the information from the method calls and is an object contained inside each node of the call tree. The database query will most likely contain some slightly different information so a new class will need to be created. However, it will still be in a similar format so this old class can be used as a basis for the query container.

Once this is done, the new database nodes will need to be integrated with the existing call tree. As each method and database call has an associated connection index (from the client listener) and method names will be matched from the stack trace, it should not be too difficult to combine the database nodes into the call tree. The way the nodes are drawn will need to be modified to account for the new database nodes.

The final part to implement, although is not too important, is to create some form of switch on the display window to allow the user to turn database interception on and off. As the LoggerInterceptor contains a Boolean variable to keep track of the logging status, the modification will just need to change this attribute. When set to false, no notifications are emitted by the LoggerManager. One thing to note would be how the profiler deals with

not receiving any database notifications when this status is changed. This would only cause problems if the new way of displaying the call tree required database nodes from the service method call leaf nodes.

Testing would also need to be done in order to iron out the bugs and find out any limitations the extension has. This would also help to find any problem areas the current profiler has. I ran across some problems occasionally connecting to the services and it would be useful to try address some of the existing issues.

The optional requirements specified in chapter 3.2 could also be incorporated. These were mostly small changes to the original profile concerning how the call tree is displayed. Since I did not complete the implementation of the database section, I did not try to address these issues during this project. More research into how to log HQL would also be beneficial. Logging SQL was discussed in section 4.6 but as explained, logging HQL is more difficult but is something that would be helpful for developers to see.

Chapter 7

Knowledge Gained

As this project was part of my final year of my degree, it was useful in teaching new ideas. This chapter describes what I have gained from this project, whether it was some specific concept or experience of working on an actual project.

7.1 Knowledge on Various Frameworks/Tools

This project involved a lot of technologies and frameworks that I had never used before. While this made the project itself quite challenging and the progress slow at first, it was extremely helpful in teaching me a range of new ideas and concepts.

Many of the tools used by Kiwiplan (such as Hibernate, Maven and even Linux) were completely new for me. Working on this project allowed me to use these tools and experiment with how they work to see how useful they can be when developing. I would expect many software companies to use similar tools when developing software, so this was a great chance to become familiar with them before moving on to work in the industry.

This project also involved a lot of research in frameworks such as JMX, MBeans and the Hibernate Interceptor. While I did not complete the profiler extension, I did get to implement and use these packages and did a lot of testing with smaller standalone applications. This gave me a chance to experiment with how they work and understand how they can be useful for Java projects.

7.2 Development Process/Work Environment

As all of my previous projects were done individually or in small groups (of at most 4), it was very different working on a project that is a part of such a large company. There are many issues to keep in mind such as how different projects depend on each other and there can be many different versions to work on.

As there can be many dependencies between projects, changes to one could easily affect how another runs. This is very different to my previous programs, where everything is kept together in one project and every project is built individually. With this project, as I had to keep classes in different projects to avoid changing any project dependencies, it often involved building one project to remove errors occurring in another.

Working on this project also allowed me to learn about the development process in a software company. Kiwiplan is a large company, with many different services in development. Updates/patches are often created, making newer versions of the projects. This means projects being worked on should be updated through the repository. Again this is very different to how I usually implement something, so it was a great experience to see how companies develop before going into the industry.

7.3 Eclipse Extensions

Even though Eclipse is the developing environment I have been using for the past three years, I was able to learn a lot more about the functions it has. As discussed previously, tools like Maven and the development process (using the repository) were very new to me and I was able to see how Eclipse can be used together with these ideas. Maven can be used to import and build Java projects. A link can also be made to Eclipse to the repository. This can be used to compare code written on the local computer with the code in the repository to see what kind of changes have been made (and revert changes if needed). Code can also be committed through Eclipse.

7.4 Documentation/Commenting

While commenting is something I've learned is important since my first year of university, it was always something I did to keep track of what attributes/methods are for or to comment out unnecessary code that I thought might be needed at a later date. This was simply a way of helping me and not specifically something for other people to use. I know that working in a company would

involve others looking my code so I usually comment my methods descriptively.

During this project, I found that it was important to use the official Javadoc style when describing attributes and methods. This allows proper API documentation to be generated and makes it possible to link descriptions between classes using tags like @link or @param. It was also important to keep the code clean and structured properly using specific code styles.

This is all very important as the code written by one person must be committed to the repository to allow others to continue with the project or do testing with it. As I only completed a part of this project, another person will be continuing this using the code I have written; therefore everything must be described clearly.

7.5 Planning a Solution

Another important thing I learnt is the need to plan a solution. Typically the programs I've written are small and although I have in mind a rough plan of the solution, I didn't need to come up with a development plan and really get into the details. With this project, it was important to create a plan of how I wanted to approach this problem and how I would solve it. This involved creating a timeline of how I wanted the project to go (included in earlier chapter). Although I was unable to follow through on the entire plan, it was definitely helpful to set small milestones and goals to make the final solution more achievable. Breaking the project down into smaller tasks also made it easier to research and do what was needed for each part and slowly build towards a final solution.

7.6 Time Management

Given that this project took place in the final year of my degree, it meant it had to be done alongside my other courses as well. Since the papers I took this year were all post-graduate courses, they were much more challenging and involved a lot more work. This meant I had to manager my time carefully in order to get the work done for my courses as well as spending time on this project every week.

7.7 Presentation/Report Skills

The requirements for this project, in terms of evaluation, included giving three seminars and writing two reports. Working on this project definitely

helped me to develop my presentation skills as well as report writing skills. Getting feedback from both my industry and academic mentors allowed me to improve on areas where I was lacking and I am sure these skills will be beneficial for me in the future.

Chapter 8

Conclusion

This chapter simply concludes the report and summarises the project.

8.1 Summary

The goal of this project was to create an extension for the existing Distributed Service Profiler to integrate the database layer. The majority of this project involved research into various concepts which helped to create the final solution. The main areas I researched were Hibernate/Hibernate Interceptors, JXM MBeans and the code from the original profiler. Using the hibernate interceptor I was able to intercept queries made to the database and with the use of MBeans, pass this intercepted information onto the existing profiler. This was all tested with smaller standalone applications or using JConsole. By using the stack trace from the intercepted query, the original service method call can be found and used to integrate the new database information with the current profiler services information.

Since I did not complete all the implementation, future work for this project would involve displaying the new database calls in the call tree as well as making some small changes to the GUI.

8.2 Conclusion

The solution I created is able to intercept database queries and communicate with profiler and I managed to complete most of the specified requirements. Even though I was unable to complete the all the implementation of the extension to the Distributed Service Profiler, this project was definitely a great learning experience. I gained a lot of knowledge on various frameworks and tools commonly used by software companies, something which I'm sure will

come in handy when I begin work in the industry. It was also great getting a chance to see how a major company runs and the kind of things I will need to know in the future.

8.3 Acknowledgments

I would like to thank everyone involved in this project - Dr Sathiamoorthy Manoharan, Tim Walker, Dr Xinfeng Ye and Edward Chen. I am especially grateful for my academic and industry mentors Xinfeng and Edward for their feedback on presentations and reports and also advice given to me during the year which helped to steer me in the right direction in order to do this project.

Bibliography

[1] "Kiwiplan — contact info." http://www.kiwiplan.com/site_about/index.cfm?abbr=en, 2011.

[2] "Profiling (computer programming)." http://en.wikipedia.org/wiki/Profiling_(computer_programming), 2011.

[3] "Java profiler - jprofiler." http://www.ej-technologies.com/products/jprofiler/overview.html, 2011.

[4] "Hibernate - jboss community." http://www.hibernate.org/, 2011.

[5] Raja, "Introduction to interceptors in hibernate orm framework." http://www.javabeat.net/articles/9-interceptors-in-hibernate-orm-framework-an-introducti-1.html, 2011.

[6] "Interceptor (hibernate api documentation)." http://www.dil.univ-mrs.fr/~massat/docs/hibernate-3.1/api/org/hibernate/Interceptor.html, 2011.

[7] "Simple logging facade for java (slf4j)." http://www.slf4j.org/, 2011.

[8] "Hql: The hibernate query language." http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html, 2011.

[9] "Overview of monitoring and management." http://download.oracle.com/javase/1.5.0/docs/guide/management/overview.html, 2011.

[10] "Using jconsole to monitor applications." http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html, 2011.

[11] "Apache maven project." http://maven.apache.org/, 2011.