Database Connection Monitoring for Component-based Persistence Systems

Uwe Hohenstein and Michael C. Jaeger
Siemens AG, Corporate Research and Technologies
Otto-Hahn-Ring 6, D-81730 Munich, Germany
Email: {uwe.hohenstein|michael.c.jaeger}@siemens.com

***Abstract*—Server applications often use a relational database management system (RDBMS) for storing and managing their data. Storage and management is not limited to the RDBMS itself, but involves also other software forming a persistence system. Such system has become complex. Besides the RDBMS it includes drivers, connection pools, query languages, the mapping between application logic and a database data model and it involves the optimisation of resources. One important resource is the connection from applications to the database system, because the acquisition of a connection is a very expensive operation.**

This work introduces monitoring facilities for the use or misuse of connections in component-based applications. In particular, it explains how the monitoring can take place in order to configure connection pooling for a set of different components. The implemented solution uses conventional programming methods as well as an aspect-oriented approach. The described facilities are integrated into the development of an enterprise-scale application implementing a communications middleware.

***Keywords*-persistence systems; O/R mapping; connection pooling; performance;**

I. Introduction

When it comes to persisting application data, relational database management systems (RDBMSs) are still the most used technology. If the application is written in an object-oriented programming language and a RDBMS is chosen, a mapping between the object data and the relational data model takes place. For this purpose, object-relational (O/R) persistence frameworks exist that implement the orthogonal persistence of data objects [1]. Examples are Hibernate [11], or OpenJPA [22] for the Java platform. By using such a technology, the data model can stay purely object-oriented. Then, programming can be done on an abstract object-oriented level, i.e., operations for storing and retrieving Java objects are provided. The O/R framework translates those object-oriented operations into SQL statements.

In principle, these O/R frameworks ease the work with RDBMSs. However, once the application becomes more complex, once it involves different components and when it processes huge amounts of data, the allocation of resources becomes an important issue for providing optimal performance. An intuitive use of persistence frameworks is not sufficient anymore; the developer must know implications of the used technology and must understand the basic principles such as caching and lazy/eager fetching strategies in order to preserve a high performance level.

Functionality and operations that involve persistence involve also the use of heavy-weight objects or can require the handling of large amounts of data. For example, careless setting of eager fetching can multiply the amount of datasets retrieved. With O/R frameworks and database communication, several heavy-weight objects are used, such as an entity that manages the mapping during run time, or the entity that handles the connection to the RDMS. As a consequence, developers must be aware of settings such as fetching strategies as well as of the proper use of API with regard to the creation of heavy-weight objects.

This work will explain the technical mechanisms managing the connections from the application to the RDMS in the context of a large-scale application middleware developed by Siemens Enterprise Communications (SEN). This middleware implements a service-oriented, server-based platform for common services in the communications domain.

The next Section 2 will explain the particular problems with database connections for component-based systems. The architecture of the SEN middleware is explained in more detail in the next Section 3. Section 4 describes the persistence subsystem of this middleware. Then, the monitoring of connections for this middleware will be explained in Section 5. The presented approach makes use of the recent technology of aspect-orientation (AO) and is applicable to common O/R frameworks such as Hibernate or OpenJPA. The subsequent Section 6 discusses the results of the connection monitoring. The paper will end with our conclusions and future work in this area in Section 7.

II. Common Problems with Database Connections

One important resource that must be handled carefully in large-scale applications is a database connection. A connection object is generally considered a heavy-weight data structure. A database system requires a lot of resources for setting up the communication and to keep space for query results etc. Hence, acquiring and releasing database connections are expensive operations.

A. Amount of Database Connections

In general, the number of connections is often limited by a configuration parameter. Moreover, the underlying operating system, or the RDMS itself has its own limits with regard to a maximum amount of open connections. This means that if one component missed to release a connection, not only resources are wasted, but also other components could be blocked when the overall number of connections is exhausted. O/R frameworks abstract from database connections by offering the concept of a session. A session includes a connection to the database, but does not solve those problems.

Moreover, large applications, having a lot of components and serving a lot of users, require a large number of connections. But in most cases, it is not reasonable to keep connections for a long time. Especially service-oriented architectures (SOAs) are characterised by short-living operations where it does not make sense to hold connections for a long time. Thus, such system has typically a high rate of connection acquisitions.

And also the best practices for O/R frameworks such as Hibernate and OpenJPA suggest using one connection for each transaction and to release it afterwards. This is inconsistent with general recommendations for using JDBC connections, since requesting and releasing a connection is time-consuming. But this is necessary, because each session has an associated object cache that becomes out-of-date at transaction end.

B. Connection Pooling

The efficient handling of short-living connections is supported by connection pools. A connection pool is an entity that allows multiple clients to share a set of connection objects each one providing access to a database. This becomes important, if a large application will serve many clients that each require connection objects. Most O/R frameworks such as Hibernate can be configured to work with a connection pool such as c3p0 [4] or DBCP [7]. Sharing connections in a pool leads to less initialisation attempts of this data structure and thus significantly reduces the time when accessing the RDBMS.

However, a connection pool does not solve all the mentioned problems. It basically reduces the number of required physical connections by means of sharing. Whenever a logical connection is closed, the underlying physical connection is not closed but put back into the pool for further usage. Anyway, closing a logical connection can still be forgotten. Moreover, the parameterization of this pool is not trivial. If the pool contains too few connections, components will have to wait until connections will be released by others. If the pool maintains too many connections, the pool itself, the RDBMS and the operating system will consume unnecessary resources for keeping connections open that are not effectively used.

In order to cope with this case, a pool can be configured to release pooled items after a certain period of inactivity. However, also this feature requires special attention, because if the connection pool might shrink too fast so that new connections must be acquired again; the advantage of a pooled connection is lost. Finally, it is not easy to understand the entire semantics of configuration parameters. Details about these parameters are not scope of this work, but interested readers are advised to compare the behaviour resulting from setting particular DBCP parameters with similarly appearing parameters from c3p0.

C. Issues with Component-Orientation

In a component-based system and also in SOA-based systems, the persistence system is usually provided as an individual instance for each unit of deployment, in most cases for each component or service. This is required, because the O/R mapping information must be given at the persistence systems initialisation time. Otherwise the O/R framework would not know which mapping to perform between objects and relational tables.

As a consequence, also an individual connection pool is maintained for each unit of deployment. Therefore, in a dynamic runtime environment, one can expect different connection pools for each persistence system that covers a domain of O/R mapping definitions. It becomes clear that for a proper use of a connection pool an appropriate parameterization is required in order to ensure optimal performance: Each unit of deployment must provide sufficient connections for the highest throughput, but otherwise just as few connections as possible to allow appropriate settings for other connection pools as well.

Configuring one connection pool is difficult, because it requires an appropriate load model and also sufficient measurements facilities. Besides the number of used connections, it must be also tracked, how long a component or thread is blocked when obtaining a connection. Considering a component-based system, it becomes clear that configuring several pools is even more difficult: How can we obtain an appropriate load model that also resembles the required parallelism?

Moreover, services call each other, which implicitly relate their connection pools somehow implicitly. Even if the connection pool of a service is large enough for its purpose, performance is degraded if the service is calling another service the pool of which is congested. It is also clear that connection information from the RDBMS is not sufficient, because the RDBMS does usually not provide information about the originating component for a connection. Such information can only be obtained by intercepting JDBC drivers or the internals of O/R mapping frameworks.

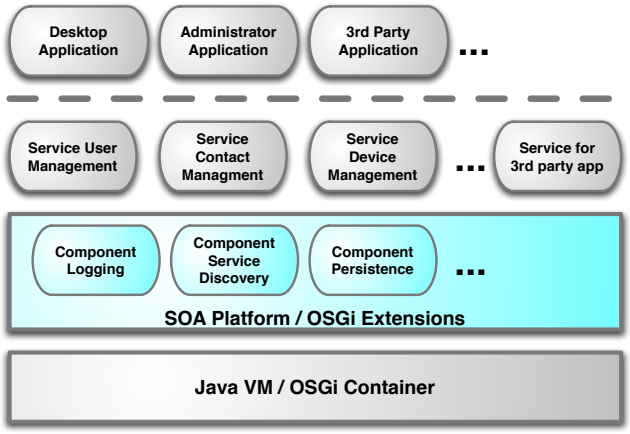

Figure 1. OpenSOA Basic Architecture

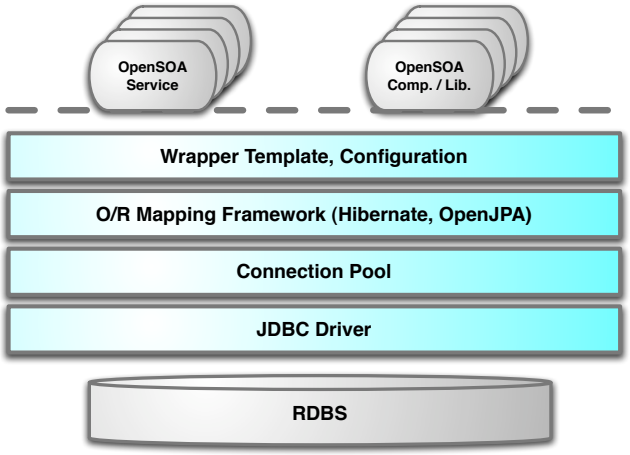

Figure 2. Persistence System Architecture

III. The OpenSOA Middleware

The application case where the monitoring was applied is a middleware called OpenSOA [28]. OpenSOA is implemented in Java and provides an open service platform for the deployment and provision of communication services. Examples for such services are the capturing of user presence, the management of calling domains, notifications, an administration functionality for the underlying switch technology, and so forth. One business case is to sell these services for the integration in groupware and other communication applications along with the Siemens private branch exchange (PBX) solutions. The technical basis for OpenSOA is an OSGi container [23]. This specification was chosen over JEE [8] because of its smaller size and focused functional extent.

In order to establish an infrastructure for the provision of services, custom built or common off-the-shelf components (depending on the availability) were added. For example, they implement the discovery and registration of services among containers on different computers for a high-performance messaging infrastructure. Figure 1 outlines the basic architecture: as the foundation, the software runs on Java and an OSGi container. Then, a tier provides components and extensions in order to implement a service-oriented architecture (SOA). On top of that, different services implement common functionality of a communication system. Finally, different application can communicate with this system using service interfaces.

The size of the entire code base (infrastructure and communication services) has similar dimensions as the distribution of the JBoss application server with regard to the defined classes (in the range of 5.000 to 10.000). It involves about 170 sub-projects among which about 20 projects use the persistence system.

IV. OpenSOA Persistence System

The previous overview OpenSOA did not explain the persistence system of OpenSOA which is subject of this section. Figure 2 shows the persistence architecture as an outline. The persistence system is divided into different tiers. Each of the tiers accesses the functionality provided by the tier below. An OpenSOA service using persistence services generally accesses the wrapping template.

A direct access of the O/R mapping framework or JDBC is not desired, however, cannot be prevented. The O/R mapping framework obtains connection from a connection pool. And the pool accesses the database via the JDBC driver. Again, the direct access to the database cannot be prevented by using the JDBC driver. But in practise it is a set rule that such calls are forbidden.

In the beginning, the Hibernate framework was used as implementation of the O/R mapping. Then, because of a patent infringement claim against Red Hat (the vendor of Hibernate), the persistence system was migrated to OpenJPA distributed by the Apache Software Foundation.

The entire persistence system allows OpenSOA for running on several RBDSs. Various business reasons require the support of solidDB from IBM: solidDB provides high performance paired with high reliability because of its hot-stand-by feature. In addition, MySQL and PostgreSQL are also supported. Moreover, the persistence framework makes programming easier by offering an object-relational mapping in order to store and retrieve the data of Java objects.

A. Working with O/R Mapping

An O/R mapping framework like Hibernate or OpenJPA requires the information about which Java classes and which fields in them are persistent, how classes map to tables, and how fields map to table columns and how pointers are mapped to foreign keys. OpenJPA supports the definition of

mappings by using an XML-document or by annotation in the Java code (with Java 5). In JPA, a self-contained set of mapping definitions forms a so called persistence unit.

The APIs of Hibernate and OpenJPA are very similar. Both use a factory object that maintains connection objects to the storage. The connection object is called Session in Hibernate and EntityManager in the JPA specification. Accordingly the factory objects are called SessionFactory and EntityManagerFactory respectively. An EntityManagerFactory exists for each persistence unit. Consequently, a connection pool is by default maintained for each persistence unit.

For programming actual operations, the developers needs to obtain a Session or an EntityManager object. From a conceptual point of view, a connection to the database is opened then (Actually, OpenJPA, for example, can be configured to actually defer the opening of a connection in order to tune for a shorter period of connection obtainment). The O/R mapping framework obtains database connections from a connection pool implementation, while the pool obtains concrete database connection from the JDBC driver. In addition, a connection pool such as DBCP or c3p0 is set between the JDBC driver and the persistence framework. The JDBC driver is used to communicate with the RDBMS.

A general recommended programming practice is to begin a transaction as well at this point. When a session is opened, the developer can perform various object operations, such as creation, deletion, modifications that are translated to according SQL statements. If a set of operations is finished, the developer should commit the transaction and close the session. From a conceptual point of view, the connection to the database is closed then. In conjunction with a connection pool, the connection is free for the next connection acquisition then. Listing 1 shows an example use with OpenJPA.

Listing 1. Example Use of OpenJPA

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("myPerUnit");
 EntityManager em = emf.createEntityManager();
 EntityTransaction tx = em.getTransaction();
 Customer c = new Customer(4711);
 em.persist(c);
 Query q =
    em.createQuery("SELECT␣c␣FROM␣Customer␣c");
 List result = q.list();
 Customer c2 = mySession.find(Customer.class, 42);
 tx.commit();
em.close();
```

B. Wrapping Template

The OpenSOA middleware extends the persistence framework with a wrapping template implementation, which standardises the use of Hibernate or OpenJPA. This template works in a similar way as the JDBC abstraction in the Spring framework [26]. It offers different functionality upon the O/R mapping framework. It basically abstracts from concepts such as OpenJPAs EntityManagerFactory (in principle a database with a schema and an associated connection pool), EntityManager (a database connection) and persistence units (logical database name). It ensures the appropriate and standardised use of this framework. When a developer uses the persistence system via the wrapping template, the following functions are provided:

1) Standardised parameterization of the persistence framework. This includes the RDBMS URL, the access credentials as well as other settings performed by the framework.
2) Standardised allocation of resources. A general setting for opening and closing connections and transactions is performed.
3) Standardised error handling. In case of recoverable errors, exceptions returned by the persistence framework or by the JDBC driver are caught and a retry is initiated, e.g., in case of concurrency conflicts. Furthermore, exceptions are converted into OpenSOA-specific ones for coverage in the layers above.

Although it is theoretically possible for developers to create, for example for the OpenJPA case, an EntityManager directly, this is generally a not allowed practice. The subsequent discussion will focus on OpenJPA and solidDB although the principles have been applied to the Hibernate framework and the other RDBMSs as well.

V. Connection Monitoring

The general aim of the monitoring is the proper configuration and usage of connections and the connection pool. Main pool parameters are the initial number of connections, the maximum and minimum number possible and the idle time after which a connection is closed after inactivity. It is a complex task to determine appropriate settings for the pool. Moreover, it is important to monitor the correct usage of connections, e.g., to avoid that connections are not released.

The monitoring functionality has been implemented at different levels in order to cover the different ways a client to the persistence system can obtain connections. Generally, the developer of a service should implicitly obtain connections by using the wrapping template. That is the first place to integrate monitoring. However, this corresponds to a logical acquisition of connections since not always real physical connections are requested thanks to pooling. The overall goal is also to get information about any physical open/close connection activity in order to keep track of the number of currently used connections for each functional unit. However, there is major a problem to obtain this information: in principle, the JDBC driver needs to be intercepted, but for example in case of solidDB, its source code is not available. To this end, the recent technology of aspect-oriented programming AOP provides an adequate solution.

A. Aspect-Oriented Programming with AspectJ

Aspect-oriented programming (AOP) provides a solution, which is not immediately obvious. AOP has originally been proposed for developing software to eliminate crosscutting concerns (CCCs) [10]. Those CCCs are functionalities that are typically spread over several classes, thus leading to code tangling and scattering [5], [20] in conventional programming. AOP provides special extensions to Java to separate crosscutting concerns. Recent research has shown usefulness to this respect, e.g., [12], [25], [29], [13], [19].

AOP is ideal for monitoring purposes. So far, some tools are implemented with AOP to monitor the system performance [3], [9]. We applied the AO language AspectJ [18], [16] to monitor the usage of database connections.

Programming with AspectJ is essentially done by Java and newly aspects. An aspect concentrates crosscutting functionality. The main purpose of aspects is to change the dynamic structure of a program by changing the program flow. An aspect can intercept certain points of the program flow, called join points. Examples of join points are method and constructor calls or executions, attribute accesses, and exception handling. Join points are syntactically specified by means of pointcuts. Pointcuts identify join points in the program flow by means of a signature expression. Once join points are captured, advices specify weaving rules involving those joint points, such as taking a certain action before or after the join points.

As AspectJ is a new language, which offers no syntactic constructs such as aspect, pointcut and advice, it requires a compiler of its own. Usually, the AJDT plug-in will be installed in Eclipse. However, a new compiler requires changes in the build process, which is often not desired. Then, using Java-5 annotations is an alternative: Aspect code can be written in pure Java; we could rely on standard Eclipse with an ordinary Java compiler, without AJDT plug-in for AspectJ compilation etc. The Listing 2 is an example that uses Java classes with AspectJ annotations.

Listing 2. Example Use of OpenJPA

```
@Aspect
class MyAspect {
 internal variables and methods;
 @Before(execution(* MyClass*.get*(..)) )
  public void myPC() {
    do something before join point
  } ... }
```

An annotation @Aspect lets a Java class MyAspect become an aspect. If a method is annotated with @Before, @After etc., it becomes an advice that is executed before or after join points, resp. Those annotations specify pointcuts as a String. This aspect possesses a @Before advice that adds logic before executing those methods that are captured by the pointcut myPC. In order to use annotations, the AspectJ runtime JAR is required in the classpath. To make the aspect active, we also have to start the JVM (e.g., in Eclipse or an OSGi container) with an javaagent argument referring to the AspectJ weaver. Annotations are then evaluated and become really active, because a so-called load-time weaving takes place: Aspect weaving occurs whenever a matching class is loaded.

If loadtime weaving cannot be applied, e.g., if AspectJ should be applied to code running in an OSGi container, another pre-compilation approach can be used which requires the iajc compiler. We can take the aspect class and compile it into a JAR file with an ordinary Java compiler. Then, the aspects JAR can be applied to classes or existing JARs, particularly, 3rd party JARs such as Hibernate or JDBC drivers. There is an iajc taskdef in ANT to make both steps easier. That is the approach we are pursuing. This is only a brief overview of AspectJ; examples are given later in the code samples.

B. Monitoring Component

All the connection information is collected by a central persistence statistics component. This component implements an OSGi component and acts as a subpart of the persistence system. This component offers two basic interface parts:

One part provides the parameterisation of the statistics functionality. It covers switching certain functionality on and off. And it defines how detailed the monitoring results should be. The output is written to a log with a definable frequency.

The other part receives the notifications of various connection attempts to obtain and release connections. The component receives such notifications and saves this information in a counter-based local data structure. Since during the life cycle, the entire application uses millions of connections, a comprehensive logging of every attempt would not make sense. Hence, the persistence statistics tracks for only currently open entities and stores an operational maximum number in addition.

Furthermore, the persistence statistics component can place an alert, if a connection remains open without any closing action, or if an attempt to close a non-existent connection happens (where no preceding obtainment has taken place). It is important to determine the origin of connection acquisitions, which could be resembled by the database user or by the persistence unit. In our case, there are only few database users are configured. Extracting the persistence unit is a better choice.

The data for these events is also separated by the functional components. That is, the persistence statistics stores all attempts separated by the user management services, all by the contact list service, etc. Depending on the parameterisation of the persistence statistics component, reports of this data can be written at different levels of granularity at different time intervals.

C. Connection Pool Monitoring: Wrapper Template

As already mentioned, the persistence system introduces a wrapping template that encapsulates database requests at a central place. This template is a first place to add monitoring functionality. It covers the notification of following events: First, the wrapping template notifies the obtainment and the release of an EntityManager of OpenJPA (which is a Session in Hibernate). Then, the template notifies the persistence statistics component when such a logical connection is obtained or released. Depending on the parameterisation, the persistence framework can automatically initialise a connection when a Session/EntityManager is obtained, or it can defer this step to later phases. For example, OpenJPA allows for three settings of connection initialisation:

a) a connection is automatically obtained for an obtained EntityManager,
b) a connection is obtained if a transaction is opened and
c) a connection is opened on demand when a connection is actually used by the persistence framework.

This represents the acquisition of logical connections. Due to a connection pool, this does not necessarily acquire a new physical JDBC connection. Similarly, closing an EntityManager is handled. The Listing 3 presents an excerpt of the template implementation.

Listing 3. Control Points in Operation Flow

```
start = System.currentTimeMillis();
session = sessionFactory.openSession();
if  (OpenJPAWrapper.getStatistics()
  .isNotificationsGenerallyEnabledYN()) {
 EntityManagerFactory emf =
   this.sessionFactory.getEMF();
 persistenceUnit = (String)
   emf.getConfiguration()
     .toProperties(false).get("openjpa.Id");
 OpenJPAWrapper.getStatistics()
     .notifySessionOpened(
        persistenceUnit, getActionId());
}
...
EntityManager oem = session.getEM();
trans = session.beginTransaction();
if (OpenJPAWrapper.getStatistics()
  .isNotificationsGenerallyEnabledYN()) {
    connectionId = jdbcConn.hashCode();
  OpenJPAWrapper.getStatistics()
    .notifyTemplateConnectionObtain(
      persistenceUnit, getActionId(),
        connectionId, elapsed);
}
stop = System.currentTimeMillis();
elapsed = stop   start;
if (elapsed > THRESHOLD) {
  printTimingWarning("connection",
       action, sql, elapsed);
}
start = System.currentTimeMillis();

if (OpenJPAWrapper.getStatistics()
  .isNotificationsGenerallyEnabledYN()) {
    OpenJPAWrapper.getStatistics()
      .notifyTemplateConnectionRelease(
persistenceUnit, getActionId(), connectionId);
OpenJPAWrapper.getStatistics()
      .notifySessionClosed(
           persistenceUnit, getActionId());
}
session.close();
```

D. Connection Pool Monitoring: JDBC Driver

The overall goal is also to get information about any physical open/close connection activity in order to keep track of the number of currently used connections for each functional unit. Since the JDBC driver needs to be intercepted the source code of which is generally not available, we apply the monitoring aspect as listed in Listing 4.

Listing 4. Aspect for JDBC Driver: Origin

```
@Aspect
public class ConnectionMonitorAspect {
  @AfterReturning(
   pointcut = "execution(Connection
solid.jdbc.SolidDriver.connect(..))",
   returning = "ret")
  public void monitorOpenJDBC
  (final Connection ret) {
   if (monitoringIsPossible) {
    String unit = determineFunctionalUnit();
    SolidConnection conn =
                  (SolidConnection) ret;
    theStatistics.notifyJDBCConnectionObtain
                     (unit, conn.hashCode());
  } } ...}
```

At a first glance, ConnectionMonitorAspect is a Java class that possesses a method named monitorOpenJDBC. However, an annotation @Aspect lets the Java class become an aspect. Since the method is annotated with @AfterReturning, it becomes an advice that is executed after returning from a method; the returning clause binds a variable ret to the return value. The method possesses a corresponding parameter Connection ret that yields access to the return value.

The @AfterReturning annotation also specifies the pointcuts as a String. Here, any execution (execution) of a method SolidDriver.connect with any parameters (..) returning a Connection is intercepted and the logic of the monitorOpenJDBC method is executed after being returned. In other words, this aspect monitors whenever a connection is opened. It determines the functional unit and uses the hashCode to identify the connection and pass both to the persistence statistics by using theStatistics.notifyJDBCConnectionObtain.

There are other forms of advices such as @Before (executing before) or @Around (putting logic around or replacing the original logic) which are handled similarly. Closing a connection via JDBC is monitored by the aspect shown in Listing 5.

Listing 5. Aspect for JDBC Driver: Counting

```
@Before("execution(*
solid.jdbc.SolidConnection.close(..))")
public void monitorCloseJdbc (final JoinPoint jp) {
```

```
  if (monitoringIsPossible) {
    String unit = determineFunctionalUnit();
    SolidConnection conn =
              (SolidConnection)jp.getThis();
    theStatistics.notifyJDBCConnectionRelease
                    (unit, conn.hashCode());
} }
```

The parameter JoinPoint jp gives access to context information about the join point, especially the object on which the method is invoked (jp.getThis()). The gathered information is sent to the central theStatistics object that keeps this information to determine the number of currently open connections. Moreover, it keeps track of the maximal value. This information is managed for each functional unit. The important point is that the solidDB JDBC driver is intercepted although its source code is not available. That is, the aspect code is applied to a 3rd party JAR file.

E. Detecting Misusage

Developers should use the template to access the database, however, there is no way to force them. Sometimes developers use JDBC to connect to the RDBMS directly, thus bypassing the entire infrastructure, e.g., for administration or setup purposes. When it comes to the monitoring of connections, all the attempts from different levels of the persistence system architecture must be monitored: Directly using JDBC connections has the inherent danger of programmers not releasing the connection, which would lead to an increase in connection usage.

Additional aspect advices allow for monitoring certain kinds of misusage when using the persistence framework directly. For example, it is monitored whenever an EntityManagerFactory is created by bypassing our wrapper template (creation is usually not done explicitly, but implicitly in the template). This means that an additional connection pool would be created for the same functional unit. The corresponding pointcut is:

```
@Before("execution(
   * *..*.createEntityManagerFactory
  (String, java.util.Map)) && args(str,map)")
```

Another pointcut detects any direct usage of JDBC connection handling beside the connection pool:

```
@Pointcut("execution(*
    org.apache.commons.dbcp.*.*(..))")
public void withinDbcp() {}
@AfterReturning(pointcut = "execution(*
    solid.jdbc.SolidDriver.connect(..))
        && !cflow(withinDbcp)", returning = "ret")
```

In both cases, an advice will issue a warning.

F. Connection Aquisition Times

Besides the amount of actual open connections and also besides the information of the origin, a problem remains: From the counting of connections, it cannot be seen, if the pool actually performs efficiently when providing connection objects. It is obvious that connection counting can happen only at specified intervals. However, an overload situation can easily slip through the measurement points.

Thus, another measurement facility has been added to the wrapping template, a time measurement, how long actually a connection obtainment takes place. If the obtainment takes place obviously the demand is higher than the number of connections that the connection pool provides.

In addition, we can guess at medium-ranged connection obtain times that the pool has freshly created the connection. Please note that it depends on the computer system used how many milliseconds the initialization of a connection object actually takes. This can have two reasons: Either the settings of the pool reduce the number of kept open connections too quickly. Or the integrity conditions for a connection are missed too often, which lets the pool replace an existing connection with a new one. Both cases also reduce the performance of the overall system. In the first case, the settings of the pool need revision. In the second case, the connection integrity conditions must be checked or the handling of connections must be evaluated for inappropriate operations on the connection object.

Although we issue a warning whenever the time to obtain a connection exceeded a certain threshold, we still do not know the reason why: Is it because the pool is exhausted and the DBS has to create a new one? Or has the connection pool some contention problems solving parallel requests? To get more diagnostics, we added an additional aspect that allows distinguishing whether a connection is newly created or taken from the pool, thereby passing a threshold. In addition, the current connection pool properties (recently active and idle connections) are printed.

G. Summary

The proposed environment offers the desired information about connections. It presents an overview of currently requested logical database connections and really used physical connections for any component a certain time intervals. Moreover, it keeps track of maximal values. If the logical number value is higher than the physical one, then the pool seems to be undersized. Similarly, an oversized pool can easily be detected. An example output send top the log files is listed in Listing 6.

Listing 6. Example Output of Monitoring

```
14:12:00,097 DEBUG [PersistenceStatistics]
  @b6d6ab says: sess opened ever/current: (1/1),
   template conn obtain
      ever/max at once/current open: (1/1/1),
   jdbc conn max same time/current: (35/35).
...
14:12:00,097 DEBUG [PersistenceStatistics]
 session referrer history:
---------------------------------------
domain:
 |_ com.siemens...createDomain(),
       ever: 1, curr: 1
   |_ max 1 session(s) in use at the same time.
---------------------------------------
```

```
 connection referrer history:
---------------------------------------
domain:
 |_ com.siemens...createDomain(),
       ever: 1, curr: 1
   |_ max 1 template conn(s) at the same time.
---------------------------------------
 1 current template connections (max at once: 1):
---------------------------------------
  12743356 origin: com.siemens...createDomain(),
---------------------------------------
 35 current jdbc connections (max at once: 35):
---------------------------------------
 |_ domain: 35
---------------------------------------
...
```

In addition to the introduced monitoring in the previous sections, also the RDBMS is queried for the connected users and the number of open connections. Figure 3 shows a comprehensive overview of the monitoring points in the architecture.

VI. Results

The introduced connection monitoring allowed insights on the use of connections for particular services or components. Such kind of information is important for the following reasons:

1) Some functionality is called very seldom, resulting in few database operations, for example a backup service, while other functionality is called in a continuous manner, for example, a user management service in several threads. The distinction by the functional units allows for an assessment of required parallel connections to the RDBMS. Over- and under-sizing can be avoided.
2) The OpenSOA platform and the services will be combined in different deployment scenarios resulting in different total amount of required connections. Such aggregated numbers are important for the proper parameterisation of the RDBMS, or the server running the RDBMS.
3) The monitoring of JDBC connections also verifies the proper use of the persistence framework. When using a persistence framework, avoiding direct connections to the RDBMS is very important. Otherwise, the object initialization, caching and the retry mechanisms can lead to inconsistent data. Using our monitoring, we could detect one component that uses JDBC directly.
4) The monitoring revealed that the number of connections oscillates within seconds with some settings of the DBCP: sometimes, even if components were waiting for a connection from DBCP, newly released connections were physically closed and immediately requested again because of the load. This is an important performance issue that leaded to severe performance degradation. Using our monitoring, it turned out that the maxActive configuration parameter defines the upper limit of connections. If another maxIdle parameter is lower, then DBCP immediately releases any exceeding connection - even if there are still requests!
   Furthermore, idle connections are evicted in a configurable interval until a minIdle threshold is passed. Again some kind of oscillation occurs, since a bulk of connections is released first, and then connections are acquired to satisfy the minIdle parameter. This strange and unexpected behavior of the DBCP could then be fixed.
5) If one component calls additionally internal operations within the same persistence unit and thus the same pool a deadlock situation can occur: At high loads the maximum amount of possible open connection can be reached by originating calls. Then a sub operation required cannot obtain further connections. However, the originating operation cannot proceed, because it waits for the sub operation. Thus, the combination of appropriate load models for testing and detailed monitoring facilities is required to systematically identify such deadlock situations.

The applied connection monitoring has clearly improved the knowledge about the deployment needs of the OpenSOA platform. We ask the reader for understanding that no detailed figures can be given as the OpenSOA platform is commercial software acting in competition to other solutions. Regarding related solutions, it must be noted that the presented implementation of such a monitoring was required for the following reasons:

1) Not every connection pool offers this information at the required grade: For example, the DBCP connection pool that we integrated into OpenJPA offers no logging at all in the current version.
2) The solidDB JDBC driver does not offer such logging facilities as required.
3) Even if some of the information were available with OpenJPA, Hibernate, or if an alternative connection pool like c3p0 would provide the information here and there, the analysis would require a cumbersome search inside the log files. In contrast, the a centralised persistence statistics component offers also the advantage of an aggregated reporting.
4) A RDBMS stores the information at system level. However, such information can be only used to verify the information gathered from the various other points. The RDBMS stores in most cases information about the number of connections obtained by a particular database user. Since different components of an application use the same database user, no detailed information can be found from there.

VII. Conclusions

Monitoring database connections in complex software is manifold and cannot be done by the database system only,

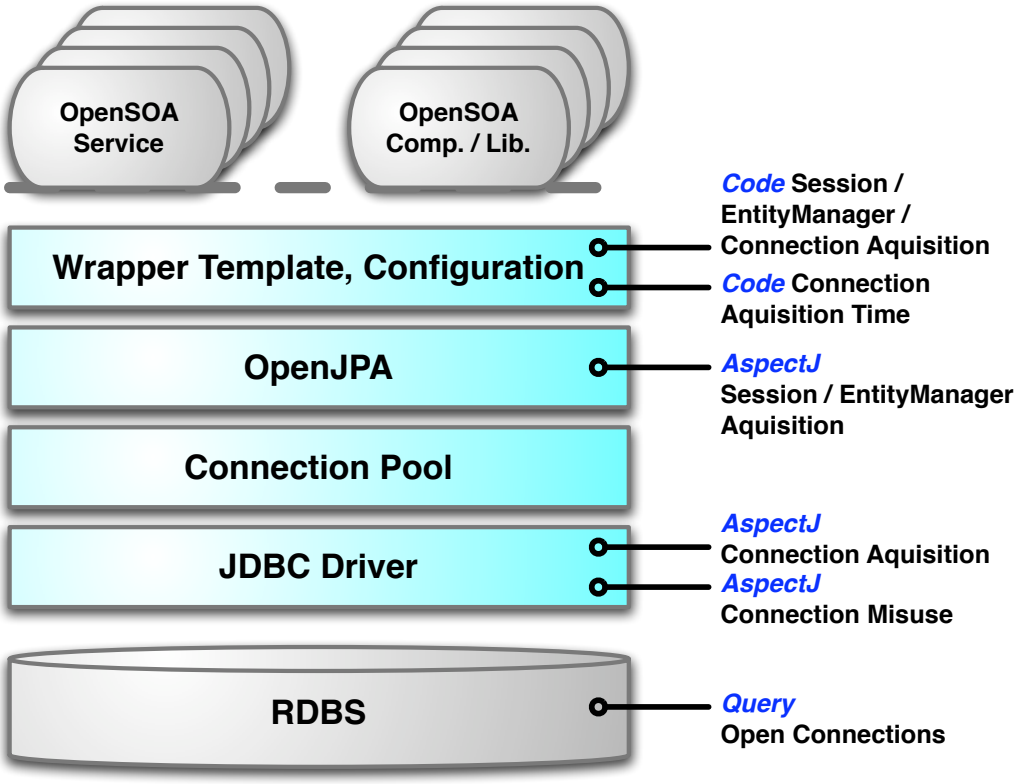

Figure 3. Monitoring in the Persistence System

if the application is distributed and involves many different components. Moreover, statistics over the counted connections is mandatory in order to precisely define parameters for optimal resource consumption/performance trade-off of the connection pool. The discussion has motivated that a precise monitoring of connections to the database and their origin is relevant in order to optimise different deployment configurations in terms of the parameters for a connection pool.

As future work, a self-parameterisation of the connection pool parameters can be implemented based on the monitoring functionality. For this purpose, dynamic connection pools exist already. However, a combination with the dynamic deployment of bundles in the OSGi container would improve adaptation of connection pool parameters instead of monitoring the resulting traffic on the connection pool level only. Such a mechanism requires additional research, because the optimal number of connections would result from a statistical model that includes also usage patterns and the number of served users.

REFERENCES

[1] M.P. Atkinson, R.Morrison: Orthogonally Persistent Object Systems. VLDB Journal 4, 3 (1995) pp 319-401.

[2] M. Aksit (ed.): Proc. of 2nd Int. Conf. on Aspect-Oriented Software Development, Boston 2003

[3] R. Bodkin: AOP@Work: Performance monitoring with AspectJ. http://www.ibm.com/developerworks/java/library/j-aopwork10/index.html

[4] c3p0 Connection Pool, Project site available at: http://c3p0.sourceforge.net/

[5] Y. Coady, G. Kiczales: Back to the Future: A Retrospective Study of Aspect Evolution in Operating System Code. In [AOSD03]

[6] M. Chapman, A. Vasseur, G. Kniesel (eds.): Proc. Of Industry Track 3rd Conf. on Aspect-Oriented Software Development, AOSD 2006, Bonn, ACM Press

[7] Commons DBCP Component. project website available at http://commons.apache.org/dbcp/

[8] The EJB3 Specification (JSR 220): http://java.sun.com /products/ejb/docs.html

[9] K. Govindraj, S. Narayanan et al.: On Using AOP for Application Performance Management. In [CVK06]

[10] R. Filman, D. Friedman: Aspect-Oriented Programming is Quantification and Obliviousness. Worksh. on Advanced Separation of Concerns, OOPSLA 2000

[11] Hibernate Reference Documentation. http://www.hibernate.org /hib_docs/v3/reference/en/html/

[12] J. Hannemann and G. Kiczales: Design Pattern Implementation in Java and AspectJ. In Proc. of the 17th Annual ACM Conf. on Obj.-Orien. Programming, Systems, Languages, and Applications, OOPSLA 2002, Seattle

[13] U. Hohenstein: Using Aspect-Orientation to Manage Database Statistics. In:

[14] JSR-000012 JavaTM Data Objects Specification. http: //jcp.org/aboutJava/communityprocess/first/jsr012/

[15] Java Persistence API. http://java.sun.com/javaee/technologies /persistence.jsp

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold: An Overview of AspectJ. ECOOP 2001, Springer LNCS 2072

[17] G. Kiczales: Adopting AOP. In Proc. 4th Conf. on Aspect-Oriented Software Development; AOSD 2005, Chicago, ACM Press

[18] R. Laddad: AspectJ in Action. Manning Publications Greenwich 2003

[19] N. Lesiecki: Applying AspectJ to J2EE Application Development. IEEE Software, January/February 2006

[20] G. Murphy, A. R. Walker, M. Robillard: Separating Features in Source Code: An Exploratory Study. In Proc. of 23rd Int. Conf. on Software Engineering 2001

[21] H. Masuhara, A. Rashid (eds.): Proc of 5th Int. Conf. on Aspect-Oriented Software Development. Bonn (Germany) 2006

[22] Java Persistence API. http://java.sun.com/javaee/ technologies /persistence.jsp

[23] Open Service Gateway Initiative. Home Page of the OSGi Consortium. http://www.osgi.org.

[24] Rashid, A.: Aspect-Oriented Database Systems. Springer Berlin Heidelberg 2004

[25] A: Rashid, R. Chitchyan: Persistence as an Aspect. In [AOSD03]

[26] Rod Johnson: Introduction to the Spring Framework. October 2007. http://www.theserverside.com/tt/articles/ article.tss?l=IntrotoSpring25

[27] S. Soares, P. Borba: Implementing Modular and Reusable Aspect-Oriented Concurrency Control with AspectJ: In WASP05, UberlŁndia, Brazil

[28] W. Strunk: The Symphonia Product-Line. Java and Obj.-Orient. (JAOO) Conf. 2007, Arhus, Denmark (2007)

[29] C. Zhang, H.-A. Jacobsen: Quantifying Aspects in Middleware Platforms. In [AOSD03].