Applying Aspect Oriented Programming to Distributed Storage Metadata Management

Dimple Kaul, Aniruddha Gokhale, Larry Dawson, Alan Tackett and Kelly McCauley
Dept of EECS and ACCRE, Vanderbilt University, Nashville, TN 37235
{dimple.kaul, a.gokhale, larry.dawson, alan.tackett, kelly.mccauley}@vanderbilt.edu

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Abstract

High performance computing applications often must handle on the order of peta bytes of data during their operation. Such large data sets inherently require distributed storage. Emerging distributed storage solutions in this realm, such as our L-Store framework, virtualize the distributed nature of the storage by offering the notion of a single file system to applications. These virtualization schemes must manage substantial amount of metadata to handle the data sets across the distributed storage. Apart from the primary concern of virtualization, a number of secondary but crosscutting concerns related to transaction and persistence control, database connection pooling, authentication and authorization, and logging must be addressed in these metadata management schemes. This paper describes our investigations into discovering these secondary and crosscutting concerns in metadata management for large-scale distributed data storage. We describe how we have applied AspectJ to address these crosscutting concerns in metadata management for the L-store distributed storage framework.

Keywords: Real-World Applications, Software, Tools, Aspects.

1. Introduction

High performance computing (HPC) applications, such as Physics simulations, require data storage in the order of tera to peta bytes over a span ranging from a few seconds to weeks or even years for their correct operation. Such large data sets are inherently stored at distributed sites. Prior to the advent of enabling technologies, such as Logistical Networking (LN) [2] and the Internet Backplane Protocol (IBP) [1], HPC applications were required to manage the distributed storage themselves. LN provides new capabilities to schedule data movement and storage on a global scale while IBP provides a middleware for managing remote storage and data objects of varying sizes.

Recent advances, such as the Logistical Store (L-Store) [14], leverage LN and IBP by masking the distribution of the storage and instead providing a single file system abstraction to applications. In order to manage distributed data and to provide a single file system abstraction, L-Store is required to maintain metadata information for the distributed data. With increasing number of files that store these large distributed data sets, the corresponding amount of metadata also increases. With an explosion in the size of the metadata itself, the problem of metadata management must be resolved for enabling technologies like L-Store to be successful. When the amount of metadata is relatively small, L-Store manages it on a single metadata server, but can scale it to larger sized systems by leveraging the Chord distributed hash table architecture [13].

Our experience working on the design and implementation of L-Store revealed a number of secondary design concerns for L-Store, such as database connection pooling or transaction management, which were tangled across the code. These additional design concerns were found to be crosscutting with respect to the primary design concerns of L-Store, which is to provide a single file system abstraction. The primary sources of these crosscutting concerns stemmed from the need to assure transactional and persistence control, database connection pooling, authentication and authorization, and exception handling and logging, which are deemed orthogonal to the primary goals of L-Store.

To improve the maintainability and extensibility of code, these sources of code tangling must be resolved, which requires the use of aspect-oriented software development (AOSD) techniques. One of the AOSD techniques is Aspect Oriented Programming (AOP). This paper describes our R&D and practical experiences gained using AspectJ [8] in the context of resolving the crosscutting concerns in L-Store. The rest of the paper is organized as follows: Section 2 describes the L-Store architecture and the crosscutting challenges we discovered during the design and implementation; Section 3 describes how we have used AOSD techniques to address the crosscutting challenges; Section 4 describes related research; and Section 5 presents concluding remarks, lessons learned.

2. Design Challenges for Metadata Management in Distributed Storage

In this section we focus on the tangled secondary concerns in metadata management that arise in the context of the L-Store architecture. To elucidate the crosscutting nature of these concerns better, we first outline our L-Store metadata management system architecture. We then illustrate how different crosscutting concerns make the design of such systems complex.

2.1 Logistical Storage

Logistical storage (L-Store) is a Java-based distributed file system providing a virtualization of a single file system to the applications

that use it. It is used for storing or writing arbitrary sized data objects and at high speeds. L-Store was created primarily to assist campus researchers who have accumulated large datasets. It stores metadata information of stored files in a database server for relatively smaller sized metadata but has the ability to leverage the Chord DHT [13] architecture for scalability.

L-Store has the ability to transfer huge amount of data for storing and access between remote labs and between different data centers. It provides real time data transfer across geographically isolated data stores. L-Store is a conceptually designed complete virtual file system. It uses the Internet Backplane Protocol (IBP) as the underlying abstraction of distributed storage, distributed hash tables (DHT) as a scalable mechanism for managing distributed metadata and software agent technology for implementing a distributed architecture.

In Internet Backplane Protocol (IBP) [1], *exnodes* are the pointers to allocations. IBP is a service that allows users to store data in the network. IBP allows allocations up to 4 GB in size. When a user requests an allocation, a depot (which is an IBP server) returns a capability (or key). It is safer to use these capabilities than FTP or HTTP for file distribution since the allocation key provides a secure access to the files. Unlike ftp and http, the key does not reveal details about the underlying file system. The IBP protocol transfers data between IBP depots by treating the entire data as a big chunk and transferring individual smaller slices. IBP provides fault tolerance and recovery features in a transparent fashion. This protocol is used by L-Store for the storage of files distributed across different storage sites.

2.2 Crosscutting Concerns in Logistical Storage

Our experience with the design and implementation of the L-Store metadata storage management system revealed a number of sources of crosscutting concerns that affect the maintainability, flexibility, extensibility and in some cases even performance. Below we describe these crosscutting concerns and how they manifest themselves in the L-Store architecture. Section 3 then describes how we resolved these design challenges.

2.2.1 Maintaining persistence in database transactions

Maintaining correct transaction control and persistence is vital for database consistency. A transaction is a logical unit of work that may include any number of database updates. During normal behavior, the issue of transaction consistency arises only in a few cases, such as before any transactions have been executed, between the completion of a successful transaction and before the next transaction begins, when the application terminates normally, or the database is closed. However, in the case of failures, without proper rollback mechanisms, transaction processing can result in inconsistent data.

L-Store internally maintains database tables for access control management and other functionalities it provides. There are some tables to store the IBP exnodes, exnode mappings, user to exnode mappings, protected rights of exnodes, among others. L-Store database transactions are executed during application operations, such as an upload of a file, which requires L-Store to update the corresponding metadata information stored across different database tables.

For example, database entries that may need to be updated based on an application action include updates to the exnode, `exnode_mappings` and some access control related tables like `protected_objects` and `protected_rights`. Thus, during this transaction if any exception is raised or an error occurs, and the transaction is aborted, there is a need to roll back the partially executed transaction. If not handled, a user may see inconsistencies such as a file being listed as available but cannot be accessed. This can prove to be a bottleneck for the application if it is not responsible to handle these failures. Handling these database transaction failures is a crosscutting concern since each different operation supported by L-Store will require handling these cases in order to maintain consistency of metadata. Thus it is necessary to make transaction persistent so that rollbacks or other failure handling can be seamlessly implemented.

2.2.2 Conventional methods used in database connection pooling

As alluded to earlier, L-Store uses database servers internally for the metadata management. Database connection management is an important parameter that dictates resulting performance. Database connection management involves a number of steps. First the connection to the database server is established over the network. Next the user trying to connect is authenticated with the database. Finally a connection is established and operations are performed. Once all activities are performed, the connection is closed resulting in the connection and server resources being freed.

Owing to all these steps, database connection management can be a bottleneck for applications using L-Store, whose main objective is to provide real time access to large quantities of distributed storage virtualized as a single file system. Thus it is important to optimize database connection management in L-Store.

Database Pooling is a process of obtaining and managing database connections faster in an application. Conventional connection pooling maintains a pool of connections in which a connection is allocated to an application when it requests a new connection and this connection is returned to the pool once the application closes the connection.

There are several conventional database connection pooling drivers like JDBC 2.0 which provide a rich set of features to the applications. They provide a standard way of creating and disposing off database connections. They reduce time to obtain new database connections but may cause extra memory and resource constraints. Moreover, the feature richness can become excessive for many applications since they must use all the functionality provided by these drivers even when they do not need them. And even if there is an option to configure some of these drivers, it is very difficult to configure them and then to test them.

In many application scenarios that use L-Store there is a need to bypass some features so that performance can be improved. In the current set of database drivers, this is not feasible and in most cases these standardized drivers may have to be replaced with proprietary drivers, which is not an acceptable alternative since the cost of developing and maintaining the code base increases. There may be times during the lifecycle of L-Store that the connection pooling feature may have to be toggled between on and off. With conventional pooling it may require changing most of the modules that use pooling [10]. These database connection pooling drivers provide a good database connection pooling solution for the application, but the application becomes tightly coupled to the driver for resource pooling. The tangling between the resource pooling and database connectivity concern is thus a big challenge needing resolution.

2.2.3 Authentication and authorization feature

Security is important in any software system. It is particularly an important challenge for distributed systems and by nature it tends to crosscut other design issues in any application. It consists of many components like authentication, authorization, auditing, and cryptography. In L-Store there is significant sharing and storing of data across geographical distributed locations. So in order to provide secure access and proper protection to the data and resources there should be a security aspect for L-Store. In order to provide security in L-Store based application we focused on the two main components – authentication and authorization.

Authentication is a process that verifies that user's credentials are valid at the time of login or in subsequent sessions. Authorization determines if the authenticated users have permission to access some system resource. For example user 'A' cannot download a file which has been uploaded by user 'B' unless it has been permitted to do so. Using conventional methods of providing security including different API's like OpenSSL, x.509 and JAAS leads to changing multiple modules in the code base of the application. The access control of L-Store was designed based on the entity relation of the various database tables.

To add security to the architecture would have forced a change to a large number of modules in our code base. After analyzing our design we found out that the authentication part was straightforward and was not really an orthogonal concern. L-Store's core functionality was designed in such a way that it was better to use an object-oriented approach to implement it. However, after designing authorization we found that it was going to affect all the important modules of L-Store. There were many file related functionalities like upload, download, list, make directory and stat among others, which needed verifying of access control. These challenges stem from the conventional object-oriented design of applications, which are tailored to meet the primary concerns but cannot accommodate secondary concerns such as authorization seamlessly in the same object oriented design framework and instead leads to a scattering of decision [10] i.e., the decision for operations to be checked against permissions is scattered throughout the system, and therefore any modifications to it can cause invasive changes.

2.2.4 Lack of consistency in exception handling

Exception handling and logging are an integral part of almost every application. Making applications *exception safe* is the responsibility of the application developer. Logging may be necessary for accounting or debugging. Often times, however, application developers ignore these secondary concerns and concentrate on the core design challenges of the application. The secondary concerns, such as exception handling and logging, become an afterthought in the design of complex systems.

We observed that the design of L-Store suffered from the same weaknesses. There are various logging techniques and toolkits that can be used for logging. For any logging toolkit, such as log4j, developers are still required to write log statements wherever logging is needed. Similar arguments hold for exception handling. Logging and exceptions are interrelated to each other. Logging of exceptions is a important part of the system. Whenever an exception is thrown, applications need to log it so that system failures and problems can be recorded and monitored. This type of logging is also called tracing or monitoring.

Logging and exception are fundamentally secondary concerns that crosscut the application code base. Due to code tangling, any changes to the logging or exception handling policy will affect large portions of the code base requiring most often manual changes.

2.3 Solution Approach: Aspect Oriented Techniques in L-Store

We used Aspect Oriented programming to provide an elegant solution to address the outlined crosscutting concerns in L-Store. AOP is an advanced programming technique used to separate crosscutting concerns in a modularized fashion. For example, since authorization is to be uniformly implemented in all the units of application, it is better to use aspect oriented techniques so that any changes to authorization are done at one place. In future if this application may expand or change access control functionality it will be very easy if we make it a separate concern.

In traditional object oriented programming languages if we add this type of concern on top of existing system core functionality we have to convert these secondary concerns into a class and then use them in primary concerns. These classes would not be reusable and they cannot be inherited and refined properly. They will ultimately scatter across the program and will be very difficult to manage and work with.

Since access control is a feature which tends to change with the evolution of application, it is always a good idea to use aspect oriented technique to design it. That way we can easily modify and understand security aspect very clearly.

AOP provides many powerful techniques to enhance code but sometimes it creates problems because it does not directly affect source code. Reading through code and understanding it becomes difficult but then even comprehension of object oriented programming is also difficult often times. Also we have to make sure that the code added or the changes made by AOP to the application should be orthogonal in nature. But sometimes aspect can be deeply crosscutting, and this happens when the application state, structure and the logic influence the aspect code in such a way that the aspect is only applicable in one specific application context [15].

3. Applying AOP Technologies to L-Store Design

Section 2 described various secondary and crosscutting concerns that make designing complex systems, such as L-Store challenging. In this section we illustrate how we resolved these challenges using AOP techniques. While addressing these secondary concerns we took care of the changeability and extensibility issues of the code. Since L-Store application is a Java based application hence we used the AspectJ AOSD technology to resolve the challenges. In the remainder of this section we first briefly describe the AspectJ which is a AOSD technology and then show how we used AspectJ to resolve the challenges outlined earlier.

3.1 AspectJ AOSD Technology

AspectJ [7] is a general purpose aspect-oriented extension to Java. The aspect-oriented constructs support the separate definition of crosscutting concerns that affect several units, of a system. This separation of concerns allows better modularity, avoiding tangled code and code spread over several units thereby improving system maintainability. AOP [8] does for crosscutting concerns what OOP has done for object encapsulation and inheritance by providing language extensions and mechanisms that explicitly capture crosscutting structure. This makes it possible to program crosscutting concerns in a modular way and achieve the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and that has greater potential for reuse.

We have applied AspectJ to resolve the crosscutting concerns in L-Store. We used the AspectJ Development Tool (AJDT) on the Eclipse IDE for our R&D.

3.1.1 Transaction Control and Persistence

L-Store is a Java based distributed file storage application. This application needs to store file information i.e., meta data information of the stored files into database server. This metadata server is used by large number of users and is designed to support millions of transactions. As we described in Section 2.2.1, because of lack of persistence there could be loss of updates, inconsistency of data and dirty reads. So it is essential for a database transaction to be persistent and all database dependent applications to guarantee the ACID properties [4] i.e., atomicity of operations, data consistency, isolation when performing operations, and data durability even if the system fails.

To address these challenges, there was a need to make some modifications to some part of original L-Store core code to implement transaction control. Originally in every operation provided by

L-Store, there was a call to database connect and release. The coupling with the primary concern was such that in order to provide transaction control we had to modify some of the methods which ended up establishing and releasing connection to the database. In the code snippet below we show parts of the original L-Store code before the secondary concerns were modularized.

```
1:  public void HandleRequest() throws Exception {
2:    String parent = br.readLine();
3:    String newDir = br.readLine();
4:
5:    try {
6:      Connection dbConn = null;
7:      dbConn = DbUtil.getDBConnection();
8:      DbLstore.insert_directory(dbConn,
9:        parent, newDir);
10:     bw.write(LStoreRequests.ALL_OK);
11:     bw.write(LStoreRequests.EOR);
12:     bw.flush();
13:   } catch (SQLException sqle) {
14:       throw new Exception ("Error creating
15:         directory: " + sqle);
16:   } finally {
17:       DbUtil.releaseDBConnection(dbConn);
18: }
```

In the above code we see that for every method there is a separate `getDBConnection()` method call (line 7). This method call is used to creating database connection and here it is used in `insert_directory` method (line 8) and then `releaseDBConnection` (line 17) is called. The modularization of transaction control as an aspect is required for lines 7 through line 17 since otherwise any intermediate failures will result in inconsistencies. So to avoid this database inconsistency for every call to database we introduced an aspect called transaction control as shown in the code snippet below.

```
1: /**
2:   * This aspect is for transaction control
3:   * of database connection
4:   */
5:  public aspect TransactionControl {
6:
7:  /**
8:    * On call of methods that match this pointcut
9:   */
10:   pointcut transactionMethod
11:     (Connection conn)
12:      :call(public static * *.*.*.DbLstore.*(..))
13:      && args(conn, ..);
14:
15: /**
16:   * Placeholder for transaction policies
17:   */
18:   Object around(Connection conn)
19:     :transactionMethod(conn){
20:     Object res = null;
21:     try{
22:       conn = DbUtil.getDBConnection();
23:       res = proceed(conn);
24:       commitTransaction(conn);
25:       DbUtil.releaseDBConnection(conn);
26:     }catch(SQLException qle){
27:       rollbackTransaction(conn);
28:       System.out.println ("Rolled back
29: transaction");
30:     }
31:     return res;
32:   }
33: }
```

In the above code snippet line 5 is the `TransactionControl` aspect created to handle transaction control of database. Line 10 is the pointcut named `transactionMethod`. It picks out the set of join points i.e., the well defined points in the program flow where the database connection is required. It will pick all the methods of `DbLstore` library having arguments as database connection. DbLstore is a database library used by L-Store application for database related connections. So, whenever these methods of DbLStore library are called, we need a database connection.

Whenever a method needing database connection is called in the application code it is detected by the aspect and provides the necessary database connection. For example, this aspect code (line 22) will establish database connection. This database connection is passed on to the methods of DbLstore using 'proceed(conn)' (line 23) where 'conn' is the database connection. This database connection is used in the method being called and then if everything is fine it will commit the transaction (line 24) and then release the database connection(line 25).

But if any kind of failure or any exception is raised it is caught in the same advice and the database transaction is rolled back (line 27). This common algorithm is modularized into an aspect and woven into the code base automatically by the weaving tools. In `TransactionControl` aspect, `rollbackTransactions()` and `commitTransaction()` are the methods that invokes the `java.sql.Connection.commit()` and `java.sql.Connection.rollback()` methods.

3.1.2 Database Connection Pooling

In order to optimize database connection pooling we used AspectJ to add database connection pooling. As discussed in Section 2.2.2 there are various constraints in bypassing traditional database connection pooling drivers when not required, and these issues can be overcome by using aspectized database connection pooling [10].

In this approach, a pre-existing driver-supported connection pooling will act as the secondary pooling strategy because AspectJ will be used to override the default connection pooling strategy. This type of connection pooling is easy to use. Database connection pooling functionality generated by AspectJ is customized according to needs of application. Using AspectJ we can provide connection pooling for only those modules where the benefits of improved speed outweighs the cost of extra space [10]. This implementation of database connection pooling is based on [10]. The advantage of this scheme is that only selected clients will be impacted by the new strategy, which can be driven by modifying a pointcut to select any number of packages and classes in an application. At any time, if the specialized strategy is not needed, an advice can nullify the effect.

Two types of pointcuts are designed in this case:

Connection creation: This pointcut (see code snippet below) is used to capture all the join points where an L-Store primary concern needs a database connection from the pool instead of creating a new one.

```
1:  pointcut connectionCreation()
2:    : call(public static Connection
3:    org.lstore.util.DbUtil.getDBConnection());
```

Connection destruction: This pointcut (see code snippet below) is used to capture all the join points where the connection is returned to the pool of database connections instead of destroying it.

```
1:  pointcut connectionRelease
2:    (Connection connection)
3:    : call(public void org.lstore.util.
4:    DbUtil.releaseDBConnection(Connection))
5:    && target(connection);
```

The above two pointcuts will be used in the following manner. First, we create an advice for the connection pooling logic for any database connection to use it from a pool instead of creating a new one. This advice is called `connectionCreation` and is shown below.

```
1:  Connection around()
2:    : connectionCreation() {
3:
4:    Connection connection= null;
5:    try{
6:      connection = connPool.getPoolConnection();
7:      if (connection == null) {
8:        connection = proceed();
9:        connPool.registerPoolConnection
10: (connection);
11:     }
12:   }catch(SQLException e){
13:      //Handle exception
14:   }
15:   return connection;
16: }
```

The next advice is to put the connection back to the pool after using it as seen in the pointcut `connectionRelease` below. In this database connection release aspect we use the "around" advice indicating the condition when the aspect must be applied.

Whenever core methods or the transaction control aspect try to release any database connection this advice will try to put the connection into the connection pool and on failure, it will use "proceed" to release database connection.

```
1: void around(Connection connection)
2:   : connectionRelease(connection) {
3:
4:   if (!connPool.putPoolConnection(connection))
5:     proceed(connection);
6: }
```

3.1.3 Authentication and Authorization:

Section 2.2.3 describes how security is a one of the major crosscutting concern which can be addressed by aspect oriented techniques very well. For authentication we did not use any AOSD techniques so we do not discuss this issue, however, aspects were required for authorization.

We implemented a very basic preliminary access control. To authorize users and to keep track of what are the users' rights we decided to use the Policy Machine model. A Policy Machine model (PM) [5] is a standardized access control mechanism and requires changes only in its configuration in the enforcement of arbitrary and organization specific attributes-based access control policies. Some of the enforceable policies are combinations of different access control policy instances like Role-Based Access Control (RBAC), Multi-Level Security (MLS) and Identity-Based Access Control (IBAC).

To address the crosscutting challenges with authorization, as a proof of concept, we started with the basic idea of Identification Based Access Control (IBAC) policy in L-Store. In future we plan to implement the entire PM. IBAC is a very straightforward access control mechanism where the owner of the resource can set access control. Most of the file systems like Unix and NTFS use this type of access control. For authorization, most of the access control was done using proper entity relation between database tables. The database table relationship is designed in such way that it follows IBAC. In Figure 1 we can see that a user can have read or write permissions for files. If the user is the owner of the directory by default it can upload, stat, list or download file. If a user is not the owner of the file it cannot perform all these operations. A user can grant permissions to any other user to access file for either read or write.

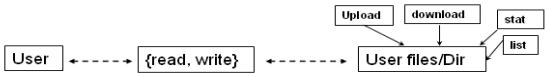

Figure 1. Identification Based Access Control configuration

For this secondary concern we had to add some code directly into core code. Following are the code snippets showing some part of code to check permissions of the user logged in. This aspect was called every time a user performs some system call like make directory, add user, change permission, grant permissions of object, etc.

```
1:  Object around(BaseTransaction tran)
2:    :execution(public * org.lstore.core.
3:     BaseTransaction.perform(..)) && this(tran){
4:
5:     try {
6:       if(tran.verifyAccess())
7:         return proceed(tran);
8:
9:     } catch( Exception ex) {
10:      ex.printStackTrace();
11:    }
12:    return LStoreRequests.createBasicReply(false);
13: }
14:
```

In this code snippet we see an advice which is called on execution of the "perform" method of any transaction. For all transaction we verify access rights (line 6) and if authorized, the transaction proceeds with the actual functionality but if the authorization fails, a reply (line 12) to client is sent indicating unauthorized access. As a side effect of addressing these challenges exception handling is also addressed as in the `verifyAccess` method (line 6) or in proceed (line 7) which is the call for original functionality.

3.1.4 Logging and Exception Handling:

Logging and Exception Handling are the most common examples to use aspects. They are an inherent crosscutting concern and tend to spread across entire application code. Exception handling was already provided in L-Store but everywhere these exceptions were implemented differently and inconsistently since they were implemented by different people at different stages of development.

In order to generalize all the exceptions we used softened exception handling of AspectJ. For exceptions which are not handled those exceptions are caught by using 'after throwing' advice. We also created some aspects to trace, profile and debug application code.

4. Related Work

The implementation of secondary concerns like logging, exception handling, transaction control, security as crosscutting concerns is not new. All these crosscutting concerns have been modularized in prior research, however, predominantly applied to enterprise computing. Our R&D demonstrates how these principles can be seamlessly applied to the domain of high performance computing.

There has been substantial prior research using AspectJ [9] for various crosscutting concerns but most of these research artifacts concentrate on one or at most two crosscutting concerns at a time on a particular system. The domain we are concentrating on requires us to manage multiple crosscutting concerns simultaneously.

In their papers [3] and [11], aspect oriented techniques have been used to modularize security aspects in an object oriented application. Other important concerns, such as transactional control, have also been addressed using AOSD techniques. For example, in their work [12] the authors implemented distribution and persistence aspects in a web based information system. The transaction control concern we addressed in this research is similar to these related works except that our work is demonstrated in the context of high performance computing and moreover, in the context and presence of multiple, simultaneous and different crosscutting concerns.

In the field of parallel-distributed systems, the use of AOSD is very limited. In the work [6] the authors have used AspectJ for separating crosscutting concerns like concurrency and parallelization concerns from core functionality. The authors demonstrate how the tangling of such concerns directly into scientific core functionality leads to increase in development complexity and decrease of code reuse. If readily portable parallel code is desired, it must be easy to change the scheme employed for achieving high-performance e.g., adapting the code to suit computer clusters or supercomputers; code tangling, of any form, makes this difficult.

5. Conclusion

This paper presented a case study illustrating how aspect oriented software development (AOSD) is useful to resolve the tangled concerns in the L-Store distributed storage management system used by high performance applications. We demonstrate the use of AspectJ, which is a Java-based AOSD tool, to address the crosscutting challenges arising from issues that were orthogonal to the primary design concerns of L-Store, such as persistence of transactions, database connection pooling, authorization, and exception handling and logging.

Often times application developers do not think "ahead of time" and miss some important but secondary design considerations, which when addressed later result in them getting tangled across the entire code base. Such code tangling makes it extremely difficult to maintain and extend the software. AOSD techniques provide the means to factor out and modularize these crosscutting concerns, which can be seamlessly and selectively woven into the fabric of the software.

Lessons Learned

There are many concerns like logging and exception handling which are perfect examples of concerns that can be cleanly separated out from the primary concerns, and plugged into the fabric of the application code base. There are however other secondary concerns that cannot be cleanly separated out from the core logic because of the tight integration with the core functionality. For example for security and transaction control we had to modify the system code to some extent.

Some of the limitations of aspect oriented programming we learned during this work are that it can sometimes increase the complexity in the design of the basic architecture since factoring out some secondary concerns is hard due to the need for minor but invasive changes in existing code base.

The problem is even more prominent when the modularization of secondary concerns and additional development of primary concerns goes on in parallel. In our case we had to deal with a situation where application developers were restructuring the code base as we were modularizing the secondary concerns, which impacted our effort since it affected the conditions when the aspects were to be woven in.

Irrespective, AOSD helps in the overall reduction of code tangling and increases the separation of concerns.

Acknowledgments

This research was supported in part by a grant from the National Foundation (NSF) CNS-SMA-0509296.

References

[1] A. Bassi, M. Beck, G. Fagg, T. Moore, J. S. Plank, M. Swany, and R. Wolski. The internet backplane protocol: A study in resource sharing. In *International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002.

[2] M. Beck, Y. Ding, T. Moore, and J. S. Plank. Transnet architecture and logistical networking for distributed storage. In *Workshop on Scalable File Systems and Storage Technologies*, San Francisco, CA, USA, September 2004.

[3] B. De Win, B. Vanhaute, and B. De Decker. How aspect-oriented programming can help to build secure software. *Informatica*, 26(2):141–149, 2001.

[4] R. Elmasri and S. B. Navathe. *Fundamentals of database systems (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[5] D. F. Ferraiolo, S. Gavrila, V. Hu, and D. R. Kuhn. Composing and combining policies under the policy machine. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 11–20, New York, NY, USA, 2005. ACM Press.

[6] B. Harbulot and J. R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 122–131. ACM Press, Mar 2004.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[10] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*, chapter 13. Manning Publications Co., Greenwich, CT, USA, 2003.

[11] V. Shah and F. Hill. An aspect-oriented security framework: Lessons learned. In B. De Win, V. Shah, W. Joosen, and R. Bodkin, editors, *AOSDSEC: AOSD Technology for Application-Level Security*, Mar 2004.

[12] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj. In *Proceedings of OOPSLA'02, Object Oriented Programming Systems Languages and Applications.* ACM Press, November 2002.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[14] A. Tackett, B. Brown, L. Dawson, S. de Ledesma, D. Kaul, K. McCaulley, and S. Pathak. Qos issues with the l-store distributed file system, Oct 2006.

[15] B. Vanhaute, B. D. Win, and B. D. Decker. Building frameworks in aspectj. Budapest, Hungary, 2001.