Aspectual Development of a P2P Secure File Synchronization Application

R. Bertan Gündoğdu, Kaan Onarlıoğlu, Volkan Yazıcı
Bilkent University Department of Computer Engineering
06800, Ankara, Turkey
{*gundogdu,onarliog,vyazici*}*@cs.bilkent.edu.tr*

Abstract

Development of a file synchronization system requires careful handling of several concerns such as transactions, fault-tolerance, and security. Very often these concerns are hard to modularize into first class elements using traditional software development paradigms. Such cross-cutting concerns decrease the cohesion and increase coupling of the components of the file synchronization system. Aspect-Oriented Software Development (AOSD) provides explicit abstractions to address these cross-cutting concerns. In this paper, we go through the development process of a secure peer-to-peer file synchronization application, FSync, and AOSD techniques to enhance separation of concerns.

1 Introduction

Peer-to-peer (P2P) architecture has emerged as a strong alternative to centralized network architectures that provides scalability, fault-tolerance, and high adaptability to dynamic changes. Contrary to a client-server system where communication is established between a central server and numerous clients, in P2P systems, there is no clear distinction between a client and a server. In other words, all the nodes in the system are considered ”peers” that can act both as clients and servers.

File synchronization process studies the concept of synchronous distribution of a data state between multiple peers. It is essential that all peers subscribed to a certain network have the same data state of the related network at a given time – excluding peers that could not finish their synchronization process for some reason and waiting in an inconsistent state.

One of the key points that must be accomplished during a synchronization process is guaranteeing the reliability of the committed state after data transfer that the state is transfered and committed on the peer side as is. Another one is atomicity of the process[1] – that after any failure a peer should not get stuck in an inconsistent state. It must also be noted that, encryption of the data traffic and authentication of the peers are essential in modern file synchronization systems.

The concerns considered above cannot easily be represented as first class elements by conventional software development paradigms. These concerns are inherently cross-cutting and are scattered among the components of the system, causing tangled code in the implementation. The reason behind this problem is that conventional development paradigms, such as Object-Oriented Software Development (OOSD), do not provide explicit abstractions to address and modularize cross-cutting concerns[9].

The recognition of such cross-cutting concerns that adversely affect the software quality lead to the emergence of a new programming paradigm: Aspect Oriented Programming (AOP). AOP aims to allow separation of cross-cutting concerns by identifying and expressing cross-cutting concerns as first-class elements in a software system. To this end, it provides additional language abstractions, called ”aspects” that explicitly encapsulate cross-cutting concerns.

In this paper, we are going to study the development of a secure peer-to-peer file synchronization application, FSync, by providing design and implementation details. We are first going to present an object-oriented solution for the design task and identify cross-cutting concerns and their implicit impacts that adversely affect the maintainability, understandability and complexity of the system. Next, we are going to apply AOSD techniques to concentrate on modularizing the identified cross-cutting concerns to enhance the overall quality of the previous design and discuss our experiences.

In the rest of this paper we first provide the preliminaries required to understand the concept of the peer-to-peer architecture and of a file synchronization system in section 2. Next, we present a high level object oriented design of the system together with explanations of core components in section 3. Then, we discuss each of the concerns ad-

[1] We assume that the transactions are not network wide, but peer wide.

dressed in the design in detail and identify the cross-cutting concerns and their impacts on the design in section 4. We move on to describe specific implementation issues and provide aspect oriented code samples in section 5 and finally, conclude with the discussions in section 6.

2 P2P File Synchronization

File synchronization studies the methodology of data state propagation among multiple endpoints. It ensures the synchronized identity of data state between distributed peers at a given point in time. Modifications applied to a peer in a synchronization network gets propagated to other peers subscribed to the same network automatically[12].

P2P communication is an popular architecture in modern network topology setups. P2P architectures introduce scalability and fault-tolerance concepts into the system built upon this architecture.

With the above considered architectures and methodologies in mind, file synchronization infrastructures implemented over P2P concepts serve as a successful platform for many real-world IT scenarios. Below you can find concrete usage examples in the field.

1. Mobilized data can be accessed from any peer where the data state synchronization consistency is supplied by the underlying mechanisms[11]. (E.g. Consider distinct employees working on the same project; each can access data from his/her own repository and propagate local changes to other peers in the network.)

2. It enables request load-balancing possible for the related data state. (E.g. It doesn't matter which peer serves an incoming request in a cloud of file servers where data states between servers are synchronized.)

3. It enables fault-tolerant architectures by making it possible to build high-available network of peers. (E.g. In above file server example, failure of a peer doesn't affect the general responsivity of the whole infrastructure.)

4. Synchronized state axiom can be used for backup purposes of sensitive data.

2.1 Implementation Approaches for P2P File Synchronization

In a P2P network, synchronization of the distinct peers implicitly turn out into an unexpectedly complex problem to be solved because of the unreliable nature of the communication environment and peers. There are various software implementation solutions to this problem each having its own advantages and disadvantages. Some of the major solutions that we want to underline are discussed in the rest of this section.

2.1.1 Master-to-Master Database Replication

In this model, multiple (R)DBMSes (Oracle[6], PostgreSQL[13], IBM DB2[8], Microsoft SQL Server, etc.) are configured to propagate each data modification query to other RDBMSes in the network. Such a design should cope with

1. Advanced locking issues (optimistic/pessimistic locking) where every node is accessed and modified simultaneously,

2. Complicated transaction atomicity methodologies. (E.g. 2-Phase commit.)

3. While this implementation makes a perfect fit for the problem, it achieves this goal by issuing complexity in the code and management sides.

2.1.2 Revision Control

Revision control models, keep the track of data state modifications in a structured manner; and to access to a data state in a point in time, it applies the modifications done till that time in an incremental order. A master peer keeps the main repository (with all of its underlying modification history) and slave peers can receive their copy of the data state of a specific point in time.

In this revision control scheme, any change must be reflected on the *master* peer and than manually propagated to slave peers.

2.1.3 Distributed Revision Control

In distributed revision control systems (git[4], darcs[3], bazaar[1], mercurial[2], etc.), every user of keeps a local copy of the repository. One can propagate changes belonging to an other repository by his/her own prompt.

While this scheme supplies a perfect environment for working with groups of individuals involved in distinct code parts (e.g. every user can have his/her own branching, tagging state), it makes it nearly impossible (and infeasible) to have each node at the same data state by design. To summarize, in this implementation it requires manual prompt of peers to receive modifications from a *specific* peer.

2.1.4 Distributed File System

While so far mentioned models run in the user level, distributed file systems[2] (e.g. OpenAFS[14], etc.) sit between

[2] Assuming file system peers are synchronized, not owners of distinct data parts.

user and kernel level[3], providing an easy to use traditional file system to the user.

This model has the same advantage, disadvantage and implementation complexities as master-to-master database replication. One more advantage of distributed file systems over master-to-master database replication is that the user isn't restricted to use an (R)DBMS, instead, he/she can benefit from the functionality of a traditional file system interface.

3 FSync: Secure P2P File Synchronization

3.1 Overview

FSync is a secure peer-to-peer file synchronization application that provides synchronization of data files across a virtual file sharing network, called a "sync network". Computers join sync networks to become "nodes" in that network and synchronize with the shared content on the network. A node can be in a single sync network or can share different content on several different sync networks; which means overlapping sync networks are possible. The system adopts a peer-to-peer architecture; in other words there is no master coordinator controlling the synchronization of the nodes; instead each node issues updates to the shared content and these local changes are propagated to the peer nodes in that sync network. Note that FSync is very different from a "versioning system" such as CVS in that it altogether avoids doing versioning, file locking through check in/out, etc. Moreover, in a CVS structure updates are pulled by clients from the server, whereas in FSync, updates are pushed onto the network by the peers. It is a lightweight application to synchronize personal content on a network in a secure and fault tolerant way.

3.2 Requirements Analysis

We have identified the core requirements of FSync as follows:

Sync Network Management: Application allows users to create sync networks, specify their names and set up passwords to control access to the network.

Joining/Leaving a Sync Network: Users can become nodes in a sync network by joining that network. Since access to the sync networks are controlled, a user must specify a valid password to be eligible to share content on that network. Users that have become nodes in network can leave that network.

File Synchronization: Users issue updates to the shared content and the rest of the nodes belonging to the corresponding sync network synchronize with the updated content.

Transactional File Transfer: During a synchronization process, file transfers are handled as a single transaction; that is to say, the local content on a node is updated only if all of the updated content is retrieved and verified through an integrity check.

Secure Network Association: Joining a network is performed through a cryptography based authentication protocol. The protocol does two-sided authentication; that is both the joining node and the node that accepts the new node into the sync network are authenticated to prevent malicious connections.

Secure File Transfer: Files are encrypted before being transmitted over a sync network. Encryption could be performed by a selection of popular symmetric encryption algorithms, which is specified while creating the sync network. The symmetric key to be used for encrypting the files is exchanged securely during the secure network association protocol.

Persistence: The application offers persistent storage of sync network management data together with the state of the synchronized folder content.

3.3 Object-Oriented Design

The class diagram of the final system, expressed in UML, is provided in Figure 1. Note that, this diagram is simplified to only include the essential details for the sake of clarity.

The core components of the system are briefly explained below:

FSync: The top-level component of the system. It manages the rest of components, acts as a bridge over several components of the system and provides an interface for the graphical user interface to interact with the application.

SyncNetwork: Represents a sync network and the local machine that acts as a node in that network. It stores all the essential data to identify a sync network, performs management of network nodes and issues update notifications to nodes during a synchronization request by the user.

Node: Represents a remote node in a sync network. It contains all the information to identify and access a node on a physical network connection. Note that the local machine is not represented as a Node object but as a SyncNetwork object, as explained above.

Encryption: The abstract parent class that represents an encryption box. It stores the encryption parameters such as the symmetric keys. This is class is sub-classed to implement encryption and decryption facilities using different cryptography algorithms.

ConnectionManager: Handles transmission of control messages. It is responsible for handling and issuing con-

[3] Actually it's also possible to implement file systems in user level using FUSE (Filesystem in Userspace) like APIs.

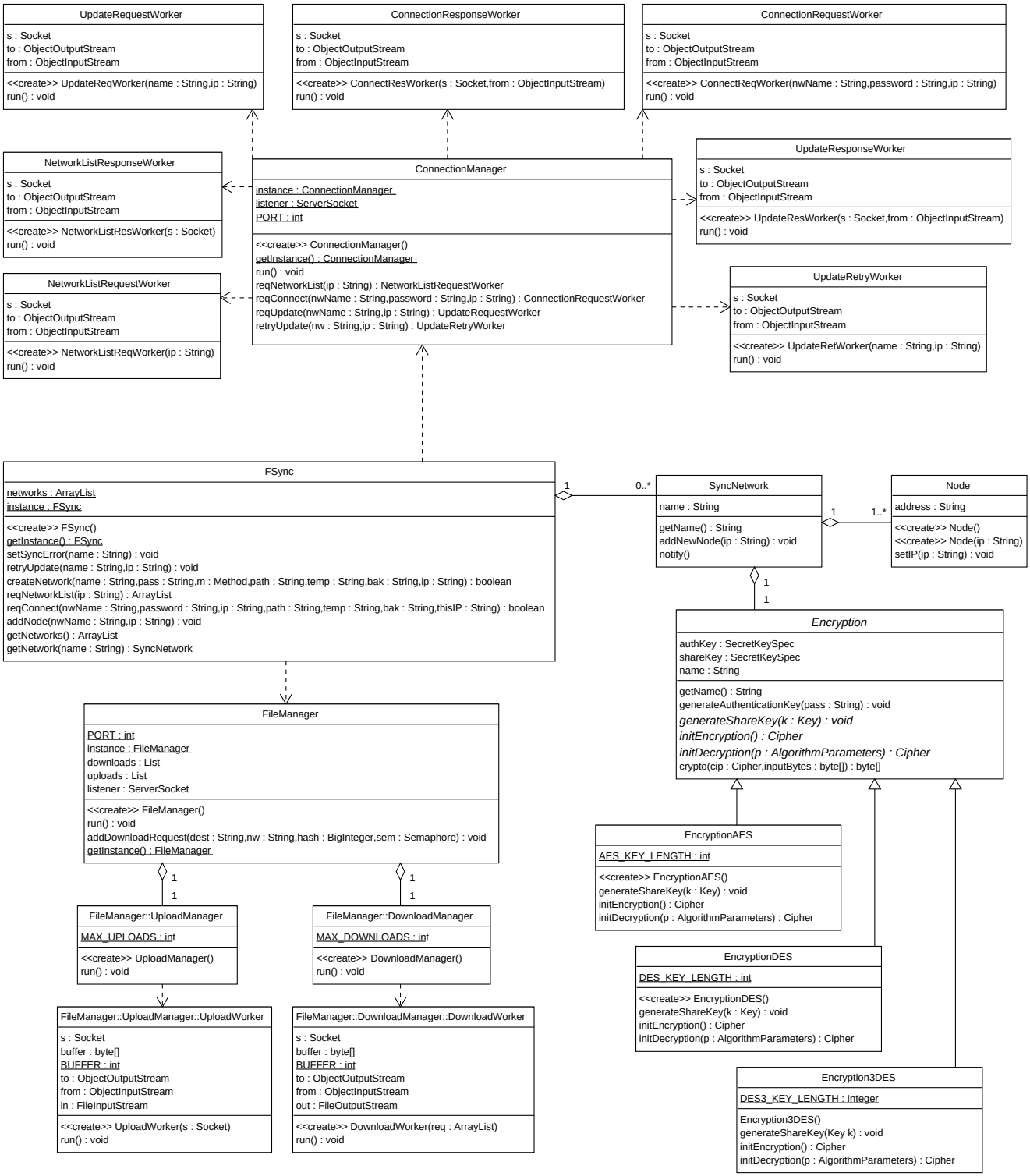

Figure 1. UML Class Diagram Depicting the Object-Oriented Design of FSync

nection requests, update notifications and retry requests. All the functionality provided by this class is performed by separate worker threads, as illustrated in the UML diagram.

FileManager: Handles transmission of files. It is respon-

sible for issuing download requests to remote FileManagers and serving uploads requests arriving from them. All the functionality provided by this class is performed by separate worker threads, as illustrated in the UML. FileManager is capable of downloading and uploading in parallel connections, number of which could be specified by the user.

The object oriented design benefits from a number of well-known design patterns.

Observer Pattern

An observer pattern is used for coupling SyncNetwork and Node objects in an abstract and oblivious fashion. Whenever the synchronized content is changed in a sync network, the notify() method of the corresponding SyncNetwork object is invoked, which then invokes the update() method of each of its Node objects. (See *Figure 2. Application of Observer Pattern in FSync*)

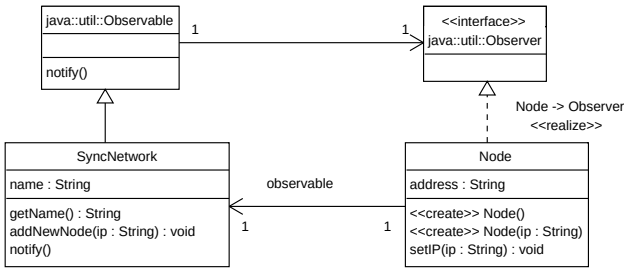

Figure 2. Application of Observer Pattern in FSync

It should be noted that the pattern applied in our case differs from the general case in that, the Node object represents a remote machine and thus it forwards the "Update" notification to the corresponding machine with the help of ConnectionManager. Then the following "Get State" message will be received again by the ConnectionManager and relayed to the SyncNetwork. In short, our case applies the observer pattern over a TCP connection, similar to Java Remote Method Invocation.

Strategy Pattern

The strategy pattern is used for providing several variant encryption algorithms with a uniform interface without affecting the rest of the system. It is used for providing DES, AES and 3DES encryption algorithms to be used in file transfer. (See *Figure 3. Application of Strategy Pattern in FSync*)

Singleton Pattern

FSync, ConnectionManager and FileManager classes are defined to be singleton classes; in other words only one instance of each can be constructed. This is essential since they use specific network and system resources that must only be accessed by a single thread of execution at a time[7].

4 Aspect Oriented Design

4.1 Problems with the Object-Oriented Approach

In the previous section we have presented an object oriented design of FSync that aims to support the quality of the system by applying the well-known object-oriented design principles: abstraction, decomposition, encapsulation, information hiding and separation of concerns. We have strived to minimize the coupling among the components of the system and provide high cohesion within. To this end, we have applied several object-oriented design patterns to support the quality factors affecting FSync.

At this point, we can safely assume that we have achieved a strong object-oriented design based on well-established object-oriented design techniques; however despite our best effort it is still possible to identify weaknesses in the system design. These are primarily caused by concerns that cannot be expressed as first class elements, in other words cross-cutting concerns, due to the inherent inadequacy of the object-oriented approach.

In the light of the requirements analysis we have provided in section 3.2, we can identify the concerns that must be expressed in the system as follows: Sync network management, file transfer & synchronization, authentication, encryption, transactions & data integrity, persistence and finally, graphical user interface updates and notifications. We shall now discuss whether each of these concerns can be expressed as first class elements in our design and demonstrate some of the problems that cannot be resolved even by high-quality object-oriented design through change scenarios on the system.

4.2 Concerns Expressed as First-Class Elements

4.2.1 Sync Network Management

Sync network management involves creating new sync networks and joining/leaving sync networks.

A sync network is expressed as a single component in the system by the SyncNetwork class; similarly nodes that belong to the networks are represented by the Node class.

Connection responsibilities are performed by ConnectionRequestWorker and ConnectionResponseWorker classes which are coordinated by the ConnectionManager. Each of these components only handles a single step of the connection process and they do not have interactions with each other.

We can conclude that, each physical computer using the system can be represented in the software by highly cohesive modules with a one way coupling link. Moreover, each

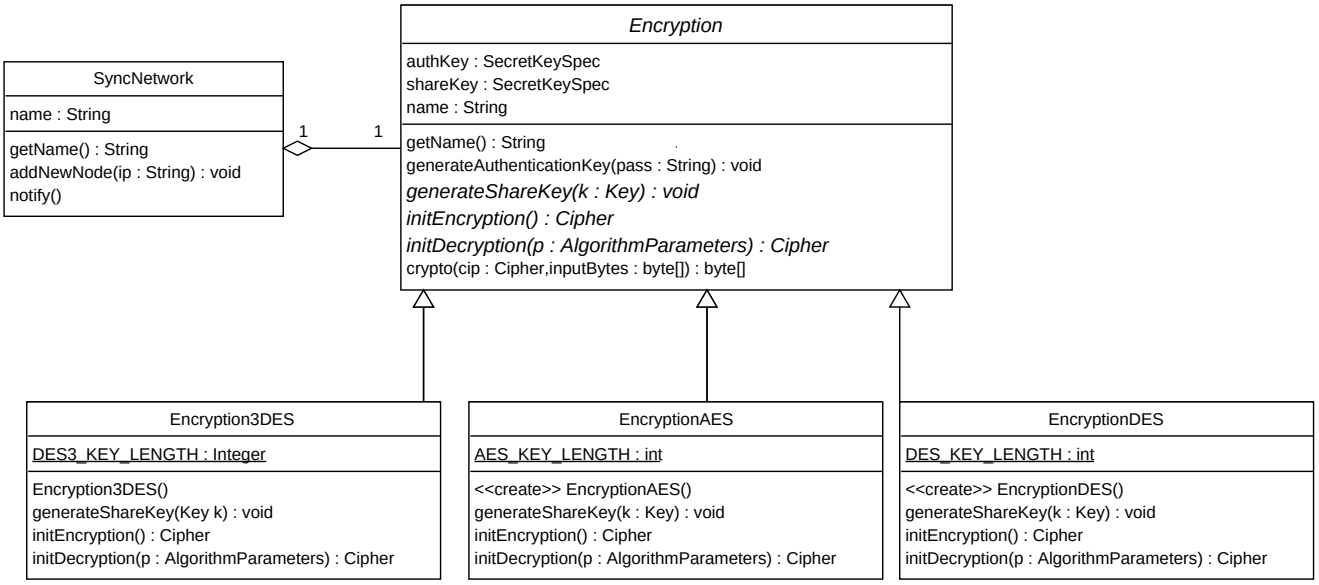

Figure 3. Application of Strategy Pattern in FSync

separate step of the connection protocol is handled by separate modules with well-defined responsibilities. Therefore, OOD can decompose this concern into sub-steps and can encapsulate each of them in separate modules with sufficient quality.

4.2.2 Synchronization & File Transfer

Synchronization & file transfer concern involves notifying nodes in a sync network of updates in the shared content and propagating the updated content to those nodes.

The notification mechanism is very well handled with the aid of the observer pattern, as discussed in section 3.3, so that it is already expected to be a strong part of the design. The communication of the modified or new content is performed by the UpdateRequestWorker and UpdateResponseWorker classes; which then issue download/upload requests to the FileManager.

File transfers are performed by the UploadManager and DownloadManager classes, which use a number of workers to transfer data in parallel. These are also coordinated by the FileManager class, so that they are lowly coupled components with a single specific task to perform.

Therefore, synchronization & file transfer is another concern that can be realized by using the OOD paradigm without violating the good design principles.

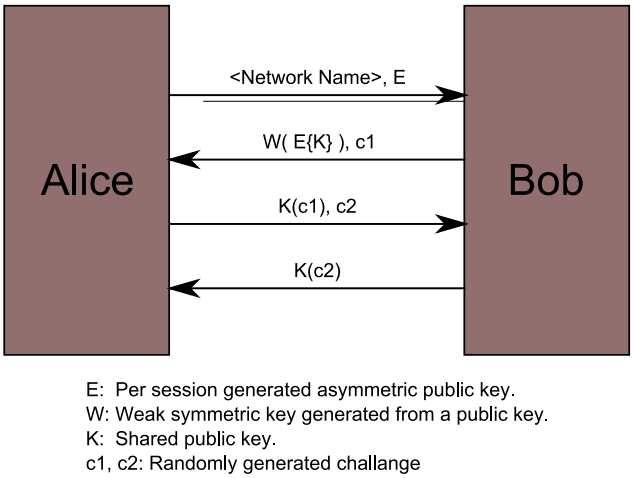

Figure 4. EKE (Encrypted Key Exchange) Protocol

4.3 Cross-cutting Concerns

4.3.1 Authentication

Authentication concern involves running a cryptography based protocol to ensure a node is eligible to join a remote sync network, and in turn ensuring that the remote point is a valid sync network. During this protocol the symmetric key to use for file encryption is also exchanged between the nodes. We have used an authentication protocol based on the well-establish Encrypted Key Exchange (EKE) protocol[5]. (See *Figure 4. EKE (Encrypted Key Exchange) Protocol*)

As demonstrated in *Figure 4* above, the EKE protocol requires two rounds of data exchange between the authenticating nodes. Since this is also a part of the sync network connection protocol, it naturally is implemented within the ConnectionRequestWorker and ConnectionResponseWorker classes, which means that the authentication concern is scattered among two components in the system. Additionally, since the original responsibilities of these workers was to handle connection requests but not to perform complex cryptographic protocols, introducing the authentication concern into their implementation causes tangled code in those modules. We shall demonstrate the effects of this problem with a change scenario:

Scenario: Replace the EKE protocol with a simple password based login protocol to improve performance in a trusted zone.

Impacts: run() methods in the ConnectionRequestWorker and ConnectionResponseWorker need to be changed dramatically. If we want to provide both protocols as options, then we need to clutter these methods with many conditional statements. The changes on the two modules must be strictly symmetric, otherwise we may break the connection protocol altogether by sending messages different from what the receiver expects next; which may give rise to security flaws such as unauthorized access as well.

4.3.2 Encryption

Encryption concern involves encrypting the files to be transfer over a physical network to maintain secrecy. It could be performed using an algorithm specified by the user among provided algorithms.

The Encryption algorithm is represented by its own class Encryption, and thanks to the strategy pattern switching between different algorithms is not an issue. However, this abstraction and the strategy pattern do not help prevent this concern from scattering. Since the files are stored in clear in disk, they must be encrypted just before the upload process. In a similar way, received files must be decrypted before being written to the disk. Consequently, UploadWorker and DownloadWorker must access the Encryption object and make use of the cryptography services it provides. Moreover, the upload worker must send a set of algorithm specific encryption parameters that are randomly generated each time a file is to be encrypted before transferring the file, and the corresponding DownloadWorker must receive this data and set up its Encryption object for proper decryption. Finally, since each sync network can possibly use a different encryption algorithm with different parameters, a SyncNetwork object must properly initialize its Encryption object upon creation. As a result, this concern is scattered among Encryption, SyncNetwork, UploadWorker and DownloadWorker classes, leading to tangled code in each of them. The following change scenario expresses the effects of this problem:

Scenario: Offer the option to bypass encryption for significantly faster synchronization of insensitive data.

Impacts: In order to support this feature we have to modify the run() methods of DownloadWorker and UploadWorker classes, to bypass invocations to the Encryption object and prevent transferring the encryption parameters. Again the changes should be strictly symmetric in order to keep the protocol running correctly. Another approach would be to introduce a dummy Encryption sub-class NoEncryption, but creating an Encryption object which actually does no Encryption would be a poor design choice. Moreover we may need to modify the constructor of the SyncNetwork class to properly initialize this new subclass.

4.3.3 Transactions & Data Integrity

This concern involves making the synchronization operations atomic and prevent going to an inconsistent state by first checking the integrity of the files received and then updating the local share.

Transaction operations are performed inside the UpdateResponseWorker since this is the worker class that checks for updated content and requests download of the modified or new files from the FileManager. After all downloads are finished, the integrity check is performed. If an error is detected, the downloaded content is discarded and the UpdateRetryWorker requests a new update by restarting the process. Not only is this concern scattered among these two components, it introduces heavy tangling code into the UpdateResponseWorker whose original task is to simply issue download requests to the FileManager. We shall demonstrate the problem with a change scenario:

Scenario: To support efficient transfer of large amounts of files, offer the option to adopt a best-effort download approach and in case of error, instead of retrying the whole update, only request download of erroneous files.

Impacts: Not only does this change require modification of the run() methods of UpdateResponseWorker and UpdateRetryWorker, but it also amplifies the problem of scattering and tangling by increasing the scope of cross-cutting: now UpdateRequestWorker should also be modified to support partial update requests.

4.3.4 Persistence

Persistence concern involves persistently storing the latest state of the synchronized folders and the list of sync networks when the system is shut down. Then, when the application is run again, these data are loaded.

The loading operation obviously must be the first task to perform once the system components are initialized and it

suffices to perform this task only once each time the application is launched. However, the saving operation is much harder to address, it must be performed each time a state change occurs. These include creating a new sync network, joining a new network, welcoming another node to a network, and committing or receiving synchronization updates. The components where these changes are observed greatly vary; FSync, SyncNetwork, UpdateRequestWorker, UpdateResponseWorker and Encryption are among the components where the persistence task is scattered among. The below change scenario expresses this problem:

Scenario: Add new fields to the SyncNetwork and make them persistent: last update date, last update issuer, version, etc.

Impacts: Apart from the task of adding these fields and methods to set/get their values, making these persistent requires populating these methods with persistence code and tangling the methods. Moreover, the load operation must also be modified to read and load the new data accordingly. The scope of cross-cutting in similar scenarios is not easy to guess, since any of the components could possibly have persistent data.

4.3.5 GUI Updates and Notifications

This concern involves updating the GUI elements according to the state of the application and display notification indicators when necessary.

This concern has a vast scope since there are numerous possible things that can be expressed to the user of the application. In our implementation, we chose to display the list of current sync networks the application is associated with, the details of these networks and details of the nodes in them. We also display notifications whenever synchronization starts, completes successfully or terminates with an error. It is easy to see that the components that are aware of these states vary greatly; for instance, only FSync knows when a new sync network is created, and only the UpdateResponseWorker can determine the end of a synchronization period. Consequently, in our implementation the task of updating the GUI needs to be addressed in almost all of the worker classes and additionally in FSync. As a result, this is probably the most scattered cross-cutting concern which causes serious tangling in each of these modules. The problem is demonstrated in the following change scenario:

Scenario: Add progress indicator bars to display the synchronization progress in percentages.

Impacts: This requires adding progress bar update calls in each of the UpdateResponseWorker, UploadWorker and DownloadWorker classes. As seen, adding a new GUI notification feature requires modification of multiple components and the scopes of the changes depend on what feature is added.

4.4 Evaluating the Object-Oriented Design and Identifying Aspects

The detailed examination of the core concerns handled in the system, in the previous part, shows that the object oriented approach alone is incapable of addressing cross-cutting concerns. This results in overall low quality and hurts complexity maintainability, understandability and re-usability. Tackling the problem of cross-cutting concerns is not possible using OOD techniques; because OOP paradigm fails to express cross-cutting concerns as first-class elements. Thus, making use of the AOP paradigm, which provides explicit abstractions to handle cross-cutting concerns, is necessary.

Taking into account the cross-cutting concerns identified in 4.3, we have defined an aspect for each of them: Authentication Aspect, Encryption Aspect, Transaction Aspect, Persistence Aspect and UIAspect. Specific implementation details of these aspects will be presented in the next section.

5 Aspect-Oriented Programming

We have refined our object oriented design to address the issues discussed in the previous section and implemented the system using aspect oriented programming techniques. Our choice of implementation language was AspectJ; thanks to its popularity, good support by development environments and simplicity as a tool for introduction to the concept of aspect orientation[10].

In this section we will briefly discuss the implementation details of two of the aspects identified above: Authentication Aspect and Encryption Aspect, together with simplified AspectJ code samples. Note that we will use `Java♯` syntax; which is an imaginary syntax specification that allows English statements preceded by ♯, where details of implementation need be avoided.

5.1 Authentication Aspect

This aspect implements the routines required to realize a successful authentication protocol during the sync network connection process. As we have already pointed out, the connection protocol is performed between a ConnectionRequestWorker and a ConnectionResponseWorker object, in the corresponding run() methods. Therefore, we need to specify pointcuts to capture the appropriate join points in those functions.

Listing 1. Authentication Aspect Pointcuts

```
public aspect AuthenticationAspect {
    // variable declarations
    ...
    // pointcut definitions
    pointcut newOInputStream(ConnectionRequestWorker w):
        set(InputStream ConnectionRequestWorker.from)
        && this( w);
    pointcut newOutputStream(ConnectionResponseWorker w):
        set(OutputStream ConnectionResponseWorker.to)
        && this( w);
    // advice definitions
    ...
}
```

Pointcuts in the *Listing 1. Authentication Aspect Pointcuts* capture establishment of communication pipes that are going to be used during the connection protocol. Then, before the rest of the connection protocol could be carried out between the ConnectRequestWorker and ConnectResponseWorker, the authentication protocol is run as an advice as in *Listing 2 Authentication Aspect Join Points*.

Listing 2. Authenticatio Aspect Join Points

```
public aspect AuthenticationAspect {
    // variable declarations
    ...
    // pointcut definitions
    ...
    // advice definitions
    after( ConnectionRequestWorker worker)
        : newInputStream( worker) {
            # send E;
            # receive W( E{K}) and  c1;
            # extract K, send K(c1);
            # receive and verify K(c2);
    }

    after( ConnectionResponseWorker worker)
        : newOutputStream( worker) {
            # receive E;
            # send W( E{K}) and  c1;
            # receive and verify K(c1), send K(c2);
    }

    // support methods
    // performs encryption operations
    private byte[] crypto( ... ){...}

    // encrypts a given key
    private byte[] keyWrap( ... ) {...}

    // decrypts a given key
    private Key keyUnwrap( ... ) {...}

    // computes a key from a password
    private SecretKeySpec deriveKey( ... ) {...}
}
```

Note that the advice contains several regular methods that are used in various stages of the protocols. Details of the advice and method contents are not provided since a cryptography background is beyond the scope of our intents.

This implementation localizes all the authentication operations inside the aspect, and results in the connection workers being unaware of the fact that an authentication protocol is running. This is a useful feature that helps us change the authentication protocol or remove it altogether without disrupting the intended connection operation.

5.2 Encryption Aspect

This aspect provides secrecy during file transfers by encrypting the transferred data. As we have discussed above, the task of sending and receiving files is performed by UploadWorker and DownloadWorker objects. Therefore, we have to define pointcuts to capture appropriate joinpoints during the file transfer operations.

The first task is to agree on the encryption parameters to be used. In our example, we will use a single parameter for this purpose, the Initialization Vector (IV). IV is a large random integer value that is roughly used for randomizing the encryption process to make it more secure. It is determined or generated by the encrypting side, and must be transferred to the decrypting side for proper decryption. The IV itself is not secret and could be transferred in plain text without affecting the security of the cryptosystem.

As a result, we need to initialize our cryptosystem, generate an IV on the encrypting side and then send it to the receiving side so that it can use the IV to initialize its own cryptosystem for decryption. This is done as demonstrated in *Listing 3. IV Initialization in Encryption Aspect*.

Listing 3. IV Initialization in Encryption Aspect

```
public aspect EncryptionAspect {
    // variable declarations
    private Cipher DownloadWorker.cipher;
    private Cipher UploadWorker.cipher;
    // pointcut definitions
    pointcut sendIV( UploadWorker w):
        set( ObjectOutputStream UploadWorker.to)
        && this( w);

    pointcut getIV( DownloadWorker w):
        set( ObjectInputStream DownloadWorker.from)
        && this( w);
    ...

    // advice definitions
    // initializes the encryption and sends IV
    after( UploadWorker w): sendIV( w) {
        w.cipher = getNetwork( w.name)
            .encyption
            .initEncryption();
        Parameters params = w.cipher.getParameters();
        w.to.writeObject( params.getEncoded());
    }

    // receives IV and initializes decryption
    after( DownloadWorker w): getIV( w) {
        byte[] param = (byte[])w.from.readObject();
        Encryption encryption = getNetwork( w.name)
            .encryption;
        Parameters ap = Parameters.getInstance();
        ap.init( param);
        w.cipher = encryption.initDecryption( ap);
    }
}
```

The pointcuts defined above take over the control as soon as the communication pipes between the workers have been established and realize the IV exchange. Note that the aspect also addresses static cross-cutting by declaring a member Cipher in each of the workers, which store the current encryption status and parameters.

The next task is to actually encrypt and decrypt the data. This is performed as in *Listing 4. Forward/Backward Encryption Methods in Encryption Aspect*.

Listing 4. Forward/Backward Encryption Methods in Encryption Aspect

```
1  public aspect EncryptionAspect {
2      // variable declarations
3      ...
4      // pointcut definitions
5      pointcut encryptBuffer( UploadWorker w):
6          call( * FileInputStream.read(..))
7          && this( w);
8
9      pointcut decryptBuffer( DownloadWorker w):
10         call( * ObjectInputStream.readFully(..))
11         && this( w);
12     ...
13     // advice definitions
14     // encrypt data buffer
15     after( UploadWorker w): encryptBuffer( w) {
16         SyncNetwork sn= getNetwork( w.name);
17         w.buffer = sn
18             .encryption
19             .crypto( w.cipher, w.buffer);
20     }
21
22     // decrypt data buffer
23     after( DownloadWorker w): decryptBuffer(w) {
24         SyncNetwork sn= getNetwork( w.name);
25     }
26 }
```

The above code sample captures every read file statement on the uploader and every receive data statement on the downloader. Then the filled data buffer is retrieved from the workers, contents are encrypted or decrypted, and the buffer is filled with the processed content. In this way, the file transfer operation is performed with absolutely no knowledge of the encryption and the aspect could be removed entirely without disrupting the normal file transfer. Moreover, all the encryption task is localized in the aspect.

Finally, we have said that each SyncNetwork object stores its corresponding Encryption object. We would also like this coupling and initialization of the Encryption object to be localized in this aspect, as demonstrated in *Listing 5. Introducing SyncNetwork in Encryption Aspect*.

Listing 5. Introducing SyncNetwork in Encryption Aspect

```
1  public aspect EncryptionAspect {
2      // variable declarations
3      private Encryption SyncNetwork.enc;
4
5      // pointcut definitions
6      pointcut newSyncNetwork( String pass,
7                               Encryption.Method m,
8                               SyncNetwork nw):
9          execution( SyncNetwork.new(..))
10         && this( nw)
11         && args( .., pass, m);
12     ...
13     // advice definitions
14     // instantiate an encryption object
15     after( String pass,
16            Encryption.Method m,
17            SyncNetwork nw):
18         newSyncNetwork( pass, m, nw) {
19
20         if( m == Encryption.Method.DES)
21             nw.enc = new EncryptionDES();
22         else if( m == Encryption.Method.AES)
23             nw.enc = new EncryptionAES();
24         # else if...
25
26         nw.enc.generateShareKey();
27         nw.enc.generateAuthenticationKey( pass);
28     }
29 }
```

This code sample demonstrates introduction of a new member variable Encryption to the SyncNetwork class through static cross-cutting and initializing it with the help of an advice that executes after a new SyncNetwork has been instantiated.

6 Discussion

- In this paper we have studied Aspect-Oriented software development on a real-world application and experienced its uses. While AOSD certainly contributed advantages to the separation of cross-cutting concerns into suitable modules, it also extended the flexibility of the application. For instance, it is sufficient to exclude the encryption aspect from source code to remove the encryption feature from the system and the application will still compile and run. Moreover, this flexibility applies to almost every aspect (e.g. Authentication, Transaction, User Interface, etc.) we used.

- It must also be noted that, one needs to be aware of the detailed uses of every aspect in the system to be able to not to conflict with pointcut definitions and to conform with aspect conventions. This obligation requires a highly strict and clear conventions for coding.

- During development, we experienced that pointcuts relying on the code fragments strictly dependent to the local conventions used at that point are more vulnerable to the programmer mistakes and result in hard to spot bugs. This issue can be solved by improving the functionality of the existing development tools (e.g. more verbose display of pointcut relations) and/or reconsidering the design of the pointcuts so that the lexical dependency is relaxed.

- User interface design and implementation requires relatively huge amount of work in the code place, and generally a majority of the graphical component actions and concerns tend to overlap. During development of FSync, we saw that such cases can be

addressed and implemented aspect-oriented programming with ease, resulting in cleaner code. For instance, close and cancel operations of dialogs can be shared, hence can be implemented within suitable aspects. Moreover, business logic does not have to contain any user interface related code. Consequently, gluing of the business logic and user interface can be implemented in aspects, instead of tangling the code throughout the system.

- Finally, we would like discuss a new alternative technology to AspectJ, that is JBoss. JBoss is a full-featured Java Application Server with enterprise quality support for different programming paradigms, one of which is AOP. Unlike AspectJ, JBoss does not introduce any new keywords to the language, but uses native Java syntax. Pointcuts are defined in XML and the weaving is dynamic and performed at run time. As an example, the pointcuts in our Authentication aspect would look like as in *Listing 6. JBoss Alternative* in JBoss.

Listing 6. JBoss Alternative

```
1 <?xml version="1" encoding="UTF-8"?>
2 <aop>
3   <bind
4     pointcut="set(ObjectInputStream
5                   ConnectionRequestWorker->from)">
6     <interceptor
7       class="ConnectionRequestWorker->doAuthentication" />
8   </bind>
9 </aop>
```

Note that, interceptors in JBoss replace the advises in AspectJ, and they are defined similar to regular Java methods. In this case, when pointcut matches the join-point the `doAuthentication` method in `ConnectionRequestWorker` class gets executed.

7 Acknowledgements

We would like to thank Asst. Prof. Dr. Bedir Tekinerdoğan, for his efforts and for organizing the Turkish Aspect-Oriented Software Development Workshop 2008.

8 Conclusion

In this paper, we have developed a peer-to-peer secure file synchronization system and studied the application of aspect-oriented software design. We first provided the object-oriented design and then elaborated on the process of identifying, specifying and implementing aspects to handle cross-cutting concerns and enhance the design quality of FSync.

As a result of our studies, we have observed that aspect-oriented software design paradigm indeed addresses the problem of scattering and tangling in the object oriented design of peerto- peer file synchronization applications by explicitly expressing the cross-cutting key concerns. Despite the discussed advantages, during the development we have found out that AOP requires dedicated awareness of the aspect joinpoints, otherwise pointcuts might easily result in hard to spot bugs in the program.

In conclusion, many core requirements of a peer-to-peer file synchronization system could be addressed by using AOSD to enhance modularization by seperating cross-cutting concerns and to provide a design that supports software quality factors.

References

[1] Bazaar version control. http://bazaar-vcs.org/.

[2] Distributed version control system. http://www.selenic.com/mercurial/.

[3] Free, open source source code management system. http://darcs.net/.

[4] Git - fast version control system. http://git.or.cz/.

[5] S. M. Bellovin and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. pages 72–84. IEEE, 1992.

[6] O. Corporation. Oracle real application clusters 11g technical overview. http://www.oracle.com/technology/products/database/clustering/pdf/twp_rac11 2007.

[7] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[8] L. J. Gu, L. Budd, A. Cayci, C. Hendricks, M. Purnell, and C. Rigdon. *A Practical Guide to DB2 UDB Data Replication V8*. IBM Redbooks, 2002.

[9] W. L. Hursch and C. V. Lopes. Separation of concerns. Technical report, 1995.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[11] D. Malkhi, L. Novik, and C. Purcell. P2p replica synchronization with vector sets. *SIGOPS Oper. Syst. Rev.*, 41(2):68–74, 2007.

[12] C. Mastroianni, G. Pirrò, and D. Talia. A hybrid architecture for content consistency and peer synchronization in cooperative p2p environments. In *InfoScale '08: Proceedings of the 3rd international conference on Scalable information systems*, pages 1–9, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[13] G. S. Mullane and E. P. Corporation. Multi-master asynchronous postgresql replication system. http://bucardo.org/.

[14] C. M. University and I. P. Labs. Open source implementation of the andrew distributed file system. http://www.openafs.org/.