Aspect-Oriented Development of a P2P File Sharing Application

Eren Algan, Ridvan Tekdogan, Dogan Kaya Berktas

Department of Computer Engineering,
Bilkent University, Ankara, TURKEY

{erenalgan, ridvantekdogan, dkberktas} @gmail.com

Abstract*—P2P applications become popular since the number of the internet users have increased. There are many concerns in these applications like concurrency, monitoring etc. Most of the concerns can modularized with OO development. However, there are some concerns that crosscut each other and Object Oriented design cannot cope with these crosscutting concerns. In this paper, we propose a new P2P application developed with Aspect Oriented design that copes with these crosscutting concerns.*

***Index terms*—Aspect-Oriented Programming, P2P, Napster**

1. Introduction

File sharing is the public or private sharing of computer data or space in a network with various levels of access privilege. P2P model is one of the models being followed by file sharing applications and yet P2P file sharing applications have been one of the most widely used applications among Internet users [1].

P2P applications run in the background of user's computers and enable individual users to act as downloaders, uploaders, file servers, etc. Designing and implementing P2P applications raise particular requirements. On the one hand, there are aspects of programming, data handling, and intensive computing applications; on the other hand, there are problems of special protocol features and networking, fault tolerance, quality of service, and application adaptability. While handling with these problems, it is important to take crosscutting concerns into account. Object Oriented designs cannot cope with crosscutting concerns. Hence, Aspect Oriented design is the solution of crosscutting concerns and yet we have designed and implemented a P2P File Sharing Application with an Aspect Oriented development.

The main idea of this study is to apply Aspect Oriented design to a P2P file sharing application, which uses Napster protocol, to solve crosscutting concerns.

The paper is organized as follows. Basic information to understand P2P file sharing applications and design patterns will be given in section 2. Example cases will be focused in detail in section 3. The approach of our system will be discussed in section 4. Problem statement will be detailed in section 5. Since there are many examples of our study, related work will be mentioned in section 6. Difficulties encountered during study will be mentioned in section 7. And finally, we will conclude our work in section 8.

2. Case Study: P2P Application

2.1 Example P2P Application

In this section we describe an example P2P application. There are various protocols designed for sharing files over Internet. The protocols can be categorized into three groups according to their network topology. 1) Napster style protocols: Clients are connected to a server for informing about their shares and for searching downloads. 2) Gnutella style protocols: Clients are connected to each other and queries send to peers and peers forward them to their neighbors until the result found. 3) Hybrid approach: Clients are clustered and select a cluster head which has responsible as server in Napster. Since in this paper we propose solution for Napster style P2P sharing application our example will be also about that.

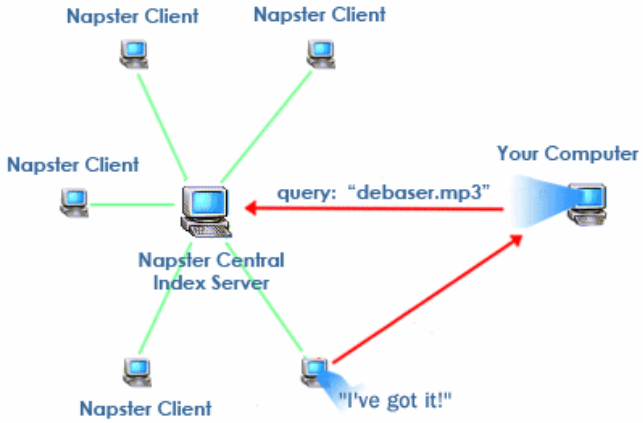

Fig 1 Example Scenario for Napster protocol

As shown in the Fig 1 there are two kinds of participant in Napster style P2P file sharing application. One of them is client and the other server. Server does not have capability of searching and sharing. It only gathers information and performs search operation. Clients connect to a server by entering its address at the

beginning and then shared folder information is sent to server. Server keeps user and his/her shared file information as long as she/he stays connected to server. When client disconnected or a connection problem occurs between client and server; server cleans the information about that client. Clients search shared files through server by sending a command. Server returns search result with owner peers' information. Client can either download from single peer or from multiple peers by directly establishing connection.

A sample OO design of P2P file sharing application is given in Fig 2 for server application and in Fig 3 for client application.

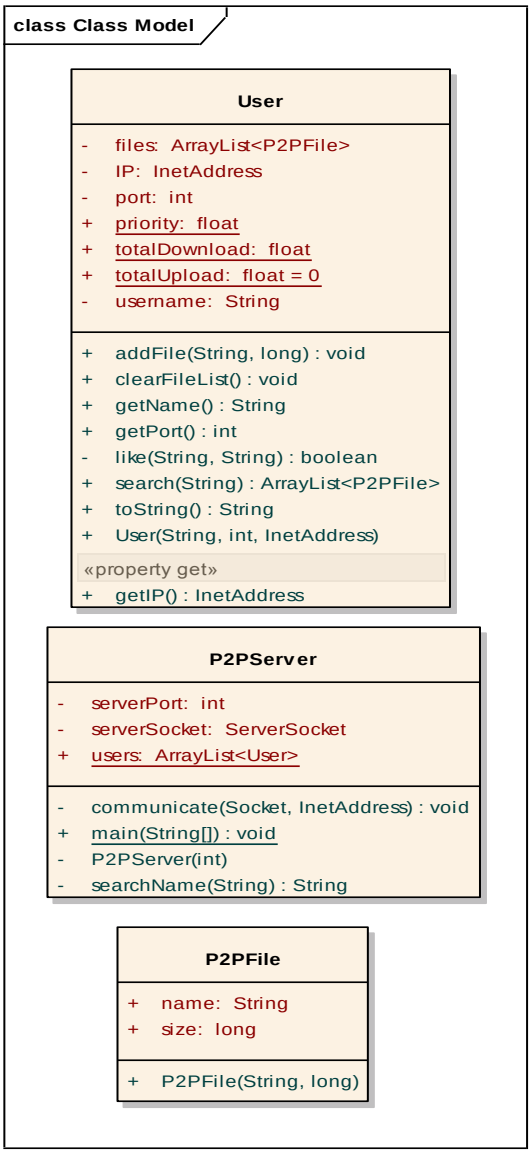

Fig 2 Class Diagram of Client Application

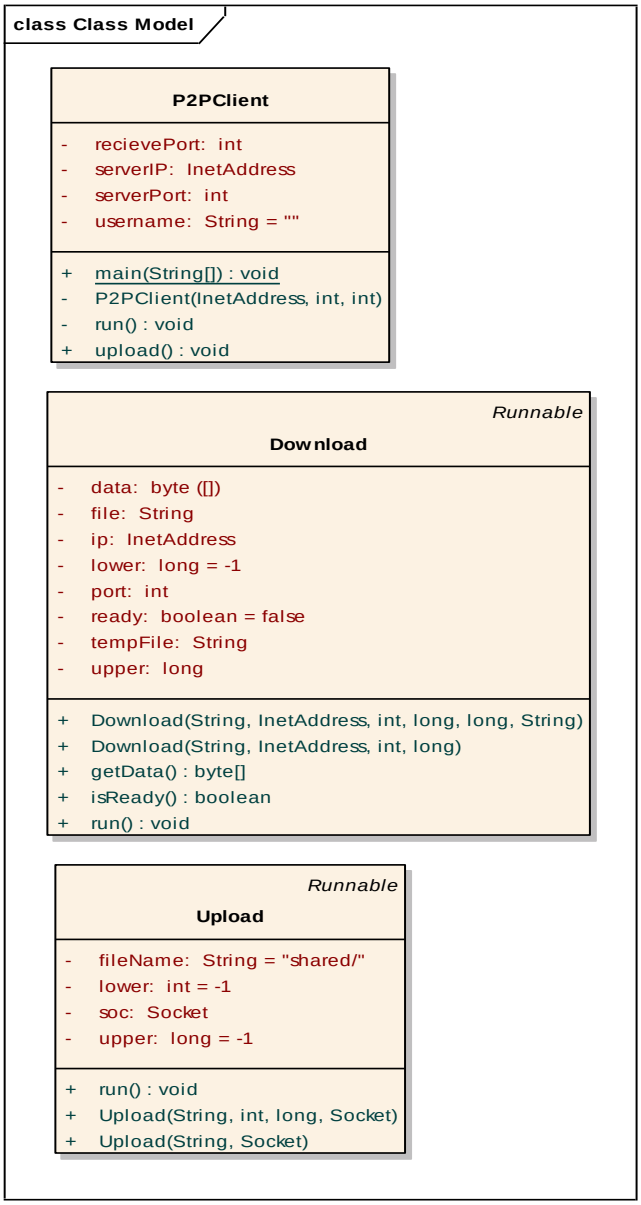

Fig 3 Class Diagram of Server Application

The server application is a batch process and when it runs it opens a server socket and listens for connections. When a client connects to server, server communicate with that client in a separate thread in communicate method.

The client application has a command line menu and when client begin to run, it opens a server socket for answering other peers file request. We have Upload and Download classes for running these operations in thread. For example when an upload request arrives, client created a new Upload instance and transfers a file concurrently with other operations.

These two applications contain the only core logic of the P2P file sharing and at the same starting point of our project. The main concern this system is concurrency: downloading uploading and doing other jobs at the same time. There are many other concerns in today's P2P applications that are not handled here. In this paper we extend these applications to cover more concerns like limiting concurrent transfers, priority based download, user friendly user interface.

2.2 Applying Design Patterns

While providing OO solution to our system, some problems have encountered are the common problems that have solutions. In such cases we make use of applying patterns. For updating UI components and informing server about change in shared folder, we provide a mechanism that monitors specified folders and inform observer classes about changes. This problem is suitable to solve with Observer Pattern. Also for providing concurrent transfer limit, we apply Active Object pattern. To make some operations to run in separate thread we have already used Worker Object pattern. These patterns and how applied will be discussed in this section.

Observer Pattern

The observer pattern (publish/subscribe) is a software design pattern in which an object maintains a list of its dependent objects and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems [5].

The participants of observer pattern are subject and observer.

Subject

This is basically an interface that enables observers to attach and detach themselves to class that implements this interface. This interface holds a list of observers that are subscribed to it. The main functions for desired functionality are as follows:

- Attach - Adds a new observer to the list of observers observing the subject.
- Detach - Removes an existing observer from the list of observers observing the subject
- Notify - Notifies each observer by calling the update function in the observer, when a change occurs.

ConcreteSubject

This class is the actual implementation for the subject. Concrete observers are subscribed to that subject. When an event occurs, the necessary observers are called.

Observer

This class defines an updating interface for all observers, to receive update notification from the subject. The Observer class is used as an abstract class to implement concrete observers.

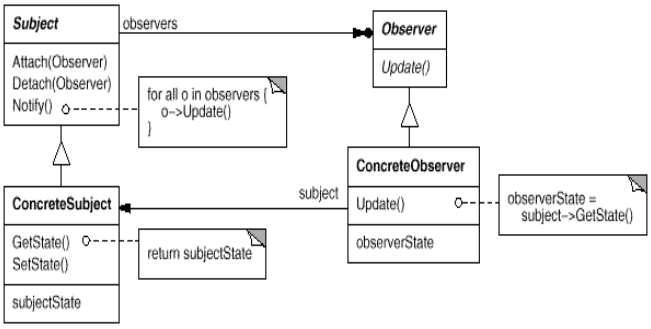

Fig 4 Structure of Observer Pattern

ConcreteObserver

This class is the actual implementation for the observers which are subscribed to ConcreteSubject. When an event occurs, the corresponding method of observers is called.

With the help of Observer pattern, subjects and observers can change independently. Any of them can be reused without having dependent to other parts. For instance, without modifying the subject, the business logic in the observers can be changed. In short, with Observer Pattern, Abstract coupling is achieved. That is subject is not aware of the concrete implementation of concrete observer and vice versa.

The downloading and uploading processes are two major parts of a P2P application since they should be in communication almost every other part of the application such as the download and upload manager, user interface, etc. The files which are downloaded and ready to be uploaded can be moved, deleted, and edited without checking the actions performed on them by the P2P system. That is why; some kind of monitoring system should be responsible for the file structure of interest and P2P system.

Active Object Pattern

Active Object pattern is object behavioral pattern, which decouples method execution from method invocation in order to simplify synchronized access to and object that resides in its own thread of control. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. A broad class of producer/consumer and reader/writer applications is well suited to this model of concurrency. This pattern is commonly used in distributed systems requiring multi-threaded servers. In addition, client applications, such as windowing systems and network browsers, employ active objects to simplify concurrent, asynchronous network operations [1].

The Structure of pattern is shown in Fig 3. Clients, who aim to call the methods of Servant class, call them through Proxy class. Only the Proxy class is visible to clients. When client calls a method on Proxy, Proxy creates corresponding MethodRequest object and enqueues it on Scheduler's queue. Scheduler contains an ActivatinQueue, which contains the MethodRequest. Scheduler also contains Dispatcher that dequeues MethodRequests that are ready to run. MethodRequest call method is responsible for calling the Servant's original method. When MessageRequest dequeued its call method is invoked.

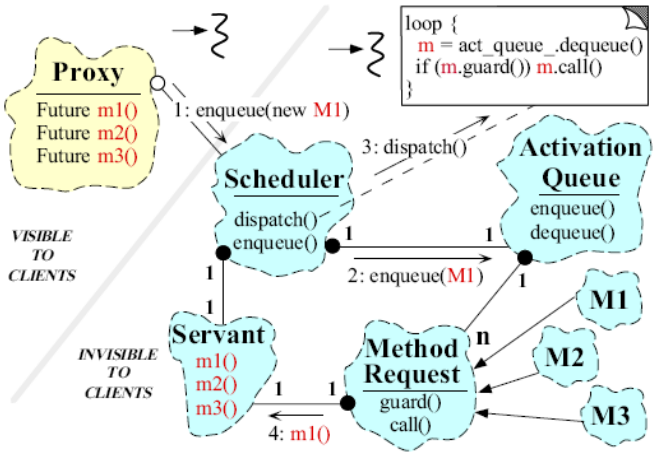

Fig 5 Structure of Active Object Pattern [2]

For decoupling the method invocation and execution to enhance the concurrency, we apply Active Object [1] pattern. Class diagram of the pattern is given in Fig 7. When we map our classes to Active Object pattern, our Servant and Client classes are same which is Client in our solution. Our MethodRequest are uploading and download because these methods have execution precedence according to priority and there is a limit for concurrent download and upload operations. So we take extra parameter in Proxy for upload and download which is the priority and also we add limit as a parameter in Scheduler's constructor. There are three phases in the pattern:

1. **Method Request construction and scheduling:** In this phase, the client invokes a method on the ServantProxy where this might upload and download in our solution. This triggers the creation of corresponding IMethodRequest which maintains the argument bindings to the method, as well as any other bindings required to execute the method. The ServantProxy then passes the IMethodRequest to the Scheduler, which enqueues it on the ActivationQueue. Here the ActivationQueue implemented as PriorityQueue based on the priority of the download and upload operation.

2. **Method execution:** In this phase, the Scheduler runs continuously in a different thread than its clients. Within this thread, the Scheduler monitors the ActivationQueue and determines which IMethodRequest(s) have become runnable, e.g., when their synchronization constraints are met. When a MethodRequest becomes runnable, the Scheduler dequeues it, binds it to the Client, and dispatches the appropriate method on the Client. When this method is called, it can access/update the state of its Client and create its result(s).
3. **Completion:** In the final phase, Scheduler continues to monitor the Activation Queue for runnable IMethodRequests. Since our methods don't return any result this phase is skipped in our solution

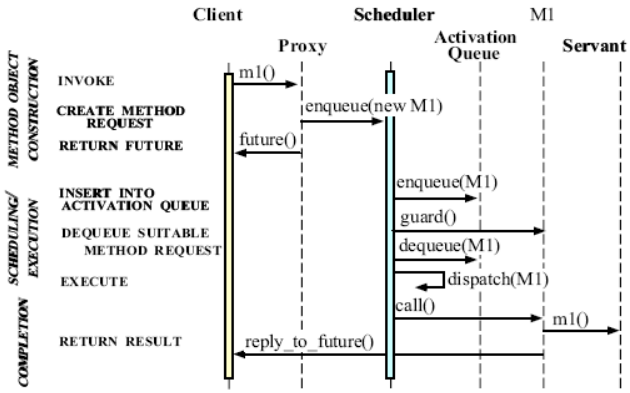

Fig 6 Dynamics of Active Object Pattern [2]

In Fig 6 the dynamics of original pattern is given, different from the original our Proxy does not return Future and in Completion phase our MethodRequest does not reply to Future.

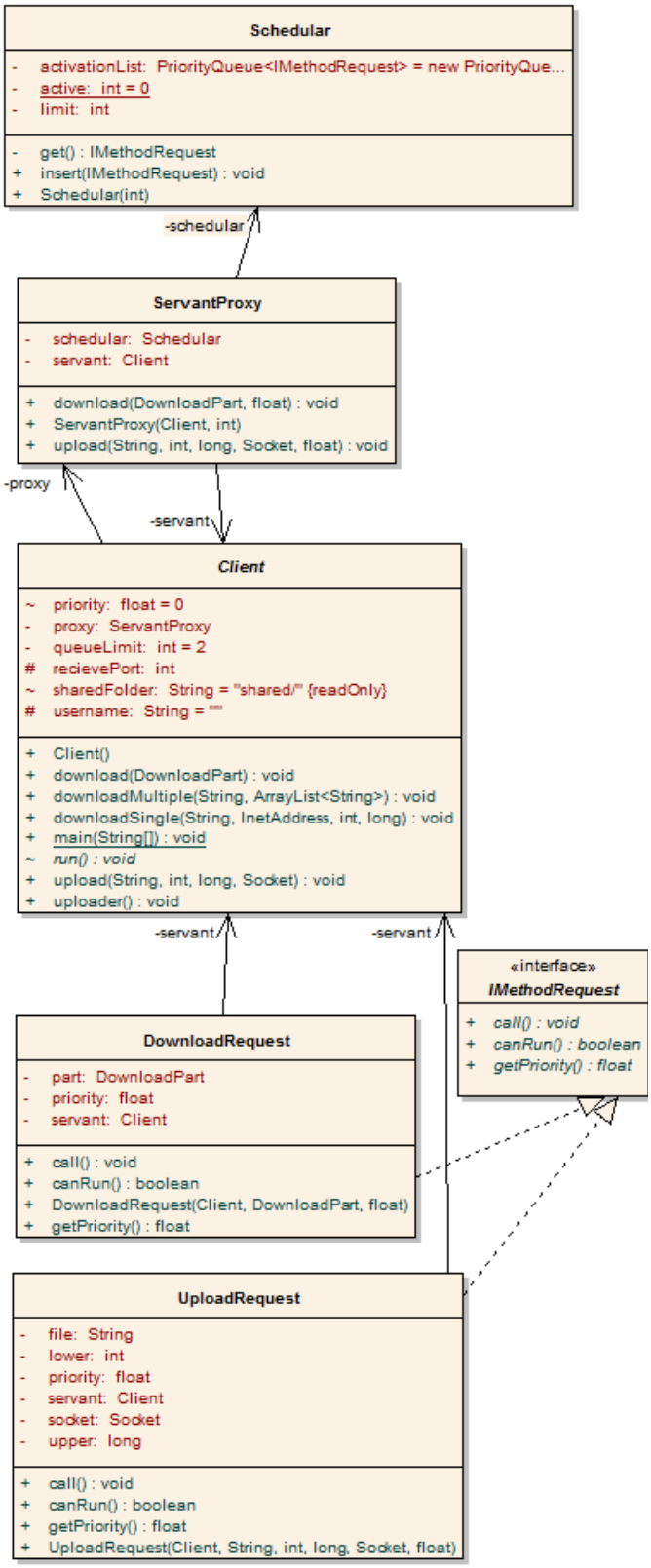

Fig 7 Class diagram of applied Active Object pattern

3. Identifying crosscutting concerns

OOSD is good way separating concerns but some concerns which have crosscutting behavior, cannot handled with OO. These concerns are creating worker objects, observer, monitoring and fault management. In this section we give the details of these concerns and why there are crosscutting.

Observing Object State Changes

The interactions between file system and P2P system are scattered among all parts (modules) of the application. Observer Pattern is the obvious choice for the problem of interaction between modules. It enable the file system watcher to broadcast event happened on the file structure. For instance, when a file is deleted, the folder structure watcher broadcast the modules that are interested with the deletion of a file such as user interface updates the table tree structure to show that the file is deleted and download/upload manager restructure its download/upload schema according to deleted source.

Another issue is to reverse communication as when the user interface wants to change the structure of the file system, and then the same procedure can be called.

Creation of Worker Objects

Since one of the main concern in P2P applications is concurrency, we need a lot of Worker Objects for operations. Some of these operations are upload, download, upload request handler, file merger which are long running operations. Choosing either defining a Worker class for each operation or creating a anonymous threads inside methods, are crosscutting.

```
public class Download extends Runnable
{
  public Download(..)
  {
        ..
  }
  public void run()
  {
        // Core aspect
  }
}
  If we define a class for Worker Object, the concerns concurrency and download, the core aspect, is tangled.
public void download(..)
{
  Thread t = new Thread()
  {
        public void run()
        {
              // Core aspect
        }
  };
  t.start();
}
```

Also creating anonymous threads inside all asynchronous methods tangled with core aspect and scattered among all classes that have such operations.

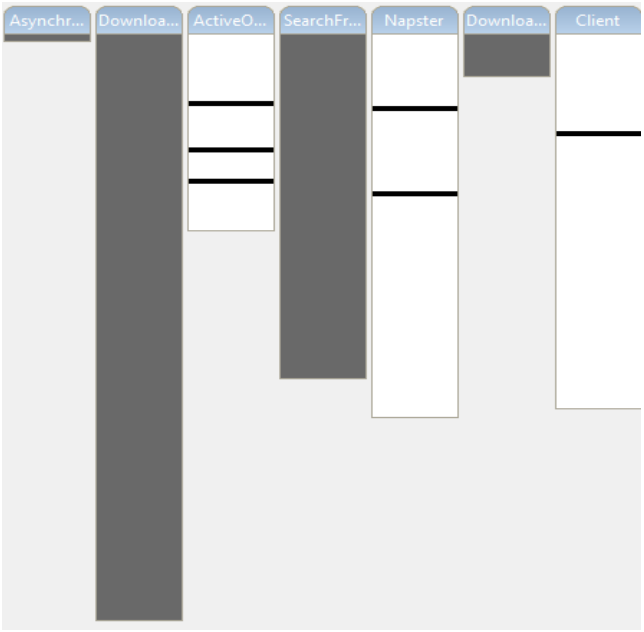

Fig 8 Scattered Creation of Worker Object Concern

As shown in Fig 8 the concurrency aspect is scattered among three classes.

Monitoring

In our concurrent system, some actions trigger some state changes objects. For example when file download completed, GUI must change status of corresponding Download object to complete. Such cases require method calls for informing about the status which is crosscuts with core aspect.

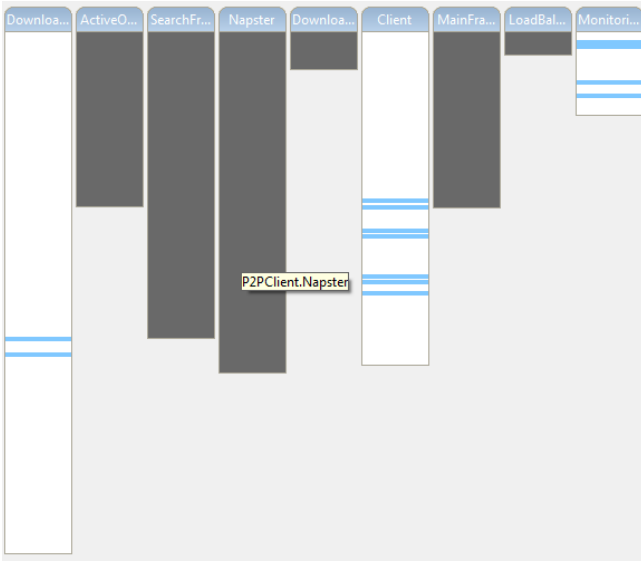

Fig 9 Scattered Monitoring Concern

As shown in the Fig 9 the monitoring concern is scattered into three classes.

4. aspect oriented implementation

Since the one the critical point software development is separating the concerns, we should separate crosscutting concerns stated in previous section. We provide our solution with using AspectJ.

Observing Object State Changes

The main purpose of the Observer Pattern is to construct a one-to-many dependency between objects so that when one object (which is the subject of the pattern) changes state, all its dependents (subscribers) are notified and updated automatically. The important questions related to Observer pattern are as following:

- Which objects are Subjects and which are Observers?
- When should the Subject send a notification to its Observers?
- What should the Observer do when notified?
- When should the observing relationship begin and end?

[4]

Before inspection our solution in terms of these questions, the motivation for using aspect oriented observer pattern is discussed below. The general object oriented implementation of observer pattern requires changes to the domain classes which is not trivial for most of the cases. For the subject class, many methods should be added (addObserver, removeObserver, notifyObservers). Also, notifyObservers method call should be added to all necessary methods to trigger the events. The observer classes should also implement update method which will be called by the subject. To start the relationship between the subject and the observer, some glue code should be implemented. As a result, it can be said that Observer pattern is not a lightweight pattern.

The peer-to-peer (p2p) system had already contains some basic functionality when we started the project. That is why; implementing observer pattern requires writing some parts of the system from scratch and for some other parts requires code modifications. As this is not a desired implementation methodology, we searched through literature and decide to apply an aspect oriented version of observer pattern. As a starting point, we use the abstract aspects of Jan Hanneman and Gregor Kiczales who prepare the 23 GoF patterns as aspect oriented. Their implementation provides an abstract aspect. Instead of the interface mechanism (subject and observer interface which contains the methods required), this approach requires these interfaces without any method in them. The aims of these interfaces are to mark the classes according to their roles. These interfaces are empty and don't require any method.

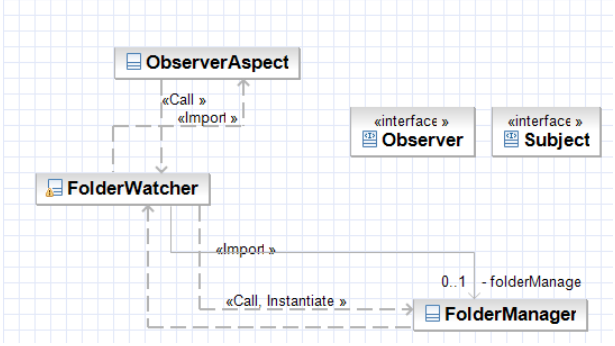

Fig 10 Observer Aspect and FolderWatcher Class Diagram

Another major difference is the tracking of observers. Normally, the participants keeps track of the observers

that listening to a particular subject. However, instead of this distributed approach, in aspect oriented approach, this job is centralized via ObserverProtocol aspect. This aspect will keep track of subjects with their observers (listeners). This is done via a HashMap structure. The key values are the subject objects and the values in the HashSet are LinkedLists that holds the observers that listen that particular subject.

```
  / * Stores the mapping between Subjects and
   * Observers. For each Subject, a LinkedList
   * is of its Observers is stored.
   */
  private WeakHashMap perSubjectObservers;

  /**
   * Returns a Collection of the Observers of
   * a particular subject. Used internally.
   */
 protected List getObservers(Subject subject)
{
    if (perSubjectObservers == null)
   {
        perSubjectObservers = new WeakHashMap();
    }
    List                observers                =
(List)perSubjectObservers.get(subject);
    if ( observers == null )
   {
       observers = new LinkedList();
       perSubjectObservers.put(subject,
observers);
    }
     return observers;
  }

/**
* Adds an Observer to a Subject.
*/
public    void    addObserver(Subject    subject,
Observer observer)
{
    getObservers(subject).add(observer);
}

/**
* Removes an observer from a Subject.
*/
public    void    removeObserver(Subject    subject,
Observer observer)
{
    getObservers(subject).remove(observer);
}
 //aspect continues...
```

Another main difference is the notifying the observers. The OO way of doing is via subject. Subject calls every observers update method. On the other side, aspect oriented version is also using a loop to call observers update methods, but this time from the aspect.

```
protected abstract pointcut subjectChange(Subject s);
after(Subject          subject)          returning          :
subjectChange(subject)
{
          for (Observer observer : getObservers(subject))
          {
            updateObserver(subject, observer);
          }
}
```

```
protected    abstract    void    updateObserver(Subject
subject, Observer observer);
```

The updateObserver() method is called when each Observer which changed state. This is implemented in FolderWatchler class which extends ObserverProtocol.

```
public  void  updateObserver(Subject  s,  Observer
o)
{
        MainFrame service = (MainFrame) o;
        service.incomingFileChange((FolderWatcher
        )s);
}
```

The role assignments are also handled via aspect. The important point here is that, all these role assignments are handled without any interference wiht modifying classes directly.

```
declare parents : FolderWatcher extends Subject;
declare parents : MainFrame implements Observer;
```

Starting the observation relationship

The last step of the pattern is to make the relationship between subjects and observers. This is also done via aspect.

```
// used for injection
private      MainFrame      defaultObserver      =
MainFrame.newContentPane;

pointcut folderWatcherCreation(Subject s) :
        execution(public FolderWatcher+.new(..))
        && this(s);

after(Subject        s)        returning        :
folderWatcherCreation(s)
{
        addObserver(s, defaultObserver);
}
```

Finally, concerning the aspect oriented observer pattern the last point is the answer to the questions mentioned above. The answer to the "Which objects are Subjects and which are Observers" is that subject is mainly the FolderWatcher class that we implemented that implements the empty Observer interface. FolderWatcher has basically maintains a thread which in charge of monitoring a given path whether there are any change in the folder structure.

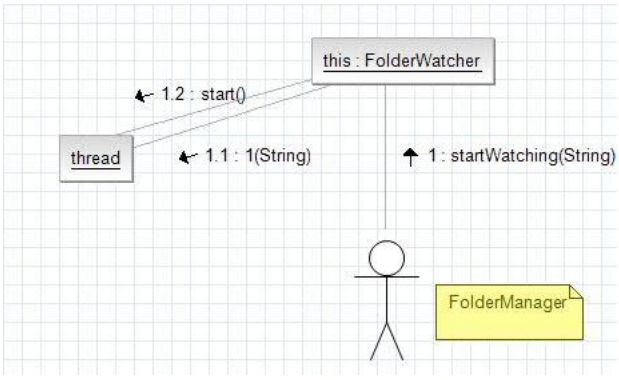

Fig 11 FolderManager thread structure

The observers are so far the user interface and to some

extent, indirectly download and upload manager. Second question is “When should the Subject send a notification to its Observers? “. This is also related to FolderWatcher implementation. Any file adding, deletion or change in the size of file produce notifications so that, for instance, when a file is being downloaded, whenever a new package is arrived, a new notification is broadcast and user interface update its view accordingly. Third question is “What should observers do when notified”. This is a part of the logic which observer has. For instance, for the case of a view in the user interface that lists the downloaded files, when user delete a file outside the system, FolderWatcher broadcasts an event and user interface should update its view via catching that event. And final question is “When should the observing relationship begin and end”. This is also a very visible through a user interface example. Say a view is disposed and no longer needed, and then the corresponding observer in the subject should be eliminated.

Creation of Worker Objects

To handle the creation of Worker Objects we define an annotation to determine which methods should be run asynchronously. Annotation does not have any attributes since only the aim is tagging the methods.

```
public @interface Asynchronous {}
```

The signatures of the asychronous methods and their usage are given below:

```
@Asynchronous public void download(DownloadPart
part): Download part of a file.

@Asynchronous public void upload(String
fileName, int lower, long upper, Socket soc):
Uploads part of a file.

@Asynchronous  private void dispatch(): Get the
next avaliable transfer request from activation
queue and calls in.

@Asynchronous public void uploader(): Handles
the upload request coming from peers.

@Asynchronous  private void merge(int numConn,
String file,ArrayList<DownloadPart> downloaders,
long fileSize): Merge the download parts.
```

A generic solution for creating Worker Object is provided with an abstract AsynchronousExecutionAspect. Aspect has a abstract pointcut which captures the asychronous methods calls. Around advice executes the method inside an anaymous Worker Object. Since In our system there is no WorkerObject returning a value, the advice returns null. It is also possible to handle the methods that return a value by defining a Worker class with a return_value attribute and getter method for that. By using the custom Worker class, the methods that return value can also be executed asynchonously.

```
public abstract aspect
AsynchronousExecutionAspect {
        abstract pointcut asyncOperations();

        Object around():asyncOperations(){
                Runnable worker = new Runnable()
{
                        public void run() {
                                proceed();
                        }
                };

                new Thread(worker).start();
                return null;
        }
}

public aspect Concurrency extends
AsynchronousExecutionAspect{
        pointcut asyncOperations():
        call(@Asynchronous * *(..));
}
```

To use the abstract AsychronousExecutionAspect we define a concrete aspect Concurrency inherits from AsychronousExecutionAspect. asyncOperations pointcut captures all the method calls with any number of parameters and return type.

Monitoring

Monitoring aspect captures all changes in objects states and takes an action according to the case. In this aspect for two cases which are monitoring the how much data download for a specific download to update the download status panel.

```
aspect Monitoring {

        long Client.totalUpload = -1;
        long Client.totalDownload = 1;


        after(Client c,byte[] data, int off, int
len): this(c)&&call(void
java.io.DataOutputStream.write(byte[],int,int))
&& args(data,off,len){
                c.totalUpload += len;

        }


        after(Client p) returning(int retval) :
this(p) && call(int
java.io.DataInputStream.read(..)) {

                p.totalDownload += retval;
        }


        after(Client p):target(p) &&(set(long
Client.totalUpload) || set(long
Client.totalDownload)){
                if (p.totalDownload> 0) {
                        p.priority= p.totalUpload/
p.totalDownload;
                }
        }

        after(Download d) : target(d) && set(long
Download.downloaded)
        {
                d.downloadFlagChanged = true;
        }

}
```

We introduce to attributes into Client class since how much data downloaded and uploaded is not the concern of client. totalUpload incremented with amount of data written on socket, where we can gather this information from the arguments of DataOutSream.write method. After advice is used since if a problem occurs while writing data, totalUpload should not change. A similar advice is

used for updating totalDownload but in this advice the return value gives us how much data is read from the socket.

For determining the priority we monitor the set operations of totalUpload and totalDownload which captures the pointcuts inside the aspect since the totalUpload and totalDownload is introduced in the aspect. Priority is the proportion of totalUpload to totalDownload.

For determining the download completed we monitor set operation of downloaded attribute and we set the downloadedFlag of Download object.

5. Discussion

In this paper we mention utilizing design patterns and aspect oriented development principles for P2P file sharing application. Instead of beginning form the scratch, we modify an existing system. Most of the problems stated in Section III were not covered in the previous system. By using the AOD principles we easily integrate our aspects to system with the minimum modification to the original system. Many of the problems that we handled are similar to existing problems in the literature so we look for the patterns applied for the similar cases. Applying patterns, especially for concurrent systems is not easy, because solutions are generally technology dependent. Implantation provided for Active Object pattern in [2] is designed for C++. At first we tried modify C++ code to get Java but we weren't able to do it. Then we provide another solution with using Proxy class and reflection in Java SDK but when calling our asynchronous methods with reflection we encountered some problems. At last we provide our proxy class implementation and handle the situation.

Since the usages of an already existing project as a starting point, applying certain patterns was hard to implement. Observer pattern is a heavyweight pattern which requires many code recompilations in many point of the code. The subject, the observers and glue code should be implemented into existing code. Instead of this approach, aspect oriented way of implementation of observer pattern is far more easy then the orthodox way. By using aspect to keep track of subject, observers' relation which is a one-to-many relation is no longer have to hard code into existing code. Moreover, aspect is also role assignments such as MainFrame class as observer is done via aspect. Normally, the code inheritance should be changed, but aspect eliminates this need. Another point is aspect catches the update in the subjects and run the desired code again inside aspect which is a perfect solution for an already existing code structure.

The problems stated in Section III are the small portion of the cake. Also access to shared resources, fault management, security and logging can be implemented by using aspect oriented development techniques.

6. Related Work

Peer-to-Peer applications have been popular lately, and therefore many studies have been performed. There are two significant works that we want to talk about in this paper. One is a framework for testing distributed systems designed by Hughes [6], and the other one is about managing concurrent file updates in a P2P system designed by Borowsky [7].

In the first system, paper discusses a Java based testing environment that facilitates the testing of distributed applications with easy publication, monitoring and control of prototype systems on a peer-to-peer test-bed. They state that, manual creation and maintenance of such tests is an extremely time consuming activity, especially where nodes are required to change their behavior dynamically. Such tests may require the creation of specialist control and monitoring tools. Furthermore, monitoring code must often be inserted throughout such prototype systems. The addition and removal of such code is time-consuming and error-prone. Their framework uses a combination of reflection and Aspect Oriented Programming to automatically insert and remove code that is required for applications to interface with the testing framework's central monitoring and control interface [6].

In the second system, they state that they solved the problem of concurrent data updates in a peer-to-peer system by layering a robust, well- tested, client-server application for managing concurrent file access on top of a peer-to-peer distributed hash table. This coupling provides a distributed, scalable, fault-tolerant, service for managing concurrent updates to replicated data. By using well-tested components, they are able to leverage the extensive development efforts put into both CVS and Bamboo to extend the functionality of both services. Moreover, this work gives proof of concept that the client-server and peer-to-peer paradigms, when appropriately layered, can provide functionality beyond the current capabilities of either paradigm alone [7].

7. Conclusion

The internet usage is widespread which leads to Peer-to-Peer (P2P) file sharing systems becoming popular way for sharing files among users on the internet. P2P enables its clients to utilize the infrastructure by downloading and uploading resources whenever available sources are found. Napster is a widely accepted protocol for P2P systems. It is basically composed of a server that holds the users info such as IP addresses and file lists that clients have and clients. Clients connect to server and give the server a list of files it has and in return client ask for other clients which contains a particular file. After the client have the list of other clients, it connects to other clients to get the file. The server doesn't interfere with the file exchanges between clients. The server only responsible for making searches among all users and return a list of clients who has the desired file.

Developing P2P file sharing system includes many complex concerns like concurrency, observing the state change of objects and monitoring status of objects. The architecture of a P2P system, due to its nature, composed of many concerns. Some of these concerns are crosscutting and they are either scattered around many modules of the system or tangling which is one module contains many unrelated concerns.

Object oriented software approach solves many problems easily but it cannot cope with crosscutting

concerns elegantly. Aspect oriented software development (AOSD) aims the separates crosscutting concerns and via that enables modularization and increase in the overall software quality. AOSD in our project act a quality enhancement.

To provide an elegant solution we apply Active Object pattern for decoupling the invocation and execution of methods for better concurrency. Downloading and uploading mechanisms are controlled by an aspect powered mechanism. The number of uploads and download and the threading mechanism is controlled in an aspect oriented fashion. For the methods that should work in separate thread we use Worker Object pattern with aspect oriented implementation. Moreover, the synchronization between file structure and P2P system is done via an observer pattern solution which is enhanced with aspect oriented approach.

Applying patterns and using aspect oriented programming make the system more reliable, easy to understand and maintainable. Also development stage of the application is shorter than traditional ways because of handling crosscutting concerns in separate modules. Another final point is that aspect oriented patterns eliminate the need to change the already existing code.

References

[1] K. G. Morrison, K. Whitehouse, “Top 10 downloads of the past 10 years”, <http://www.cnet.com/1990-11136_1-6257577-1.html>

[2] R. G. Lavender, D. C. Schmidt “Active Object : An Object Behavioral Pattern for Concurrent Programming,” http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf

[3] R. Laddad, *AspectJ in Action*, Manning:Greenwich, 2003, pp.247-256

[4] N. Lesiechi, “Enhance Desing Patterns”, <http://www.ibm.com/developerworks/java/library/j-aopwork6/index.html#sidebar3>

[5] Wikipedia contributors, 'Observer pattern', Wikipedia, The Free Encyclopedia, 10 December 2008, 15:06 UTC, <http://en.wikipedia.org/w/index.php?title=Observer_pattern&oldid=257061737>

[6] D. Hughes, P. Greenwood, G. Coulson, *Framefork for Testing Distributed Systems*

[7] E. Borowsky, A. Logan, R. Signorile, *Leveraging the Client-Server Model in P2P: Managing Concurrent File Updates in a P2P System*, Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)