FlexiFrag: A Design Pattern for Flexible File Sharing in Distributed Collaborative Applications

Anjaneyulu Pasala D. Janaki Ram

Department of Computer Science and Engineering

Indian Institute of Technology, Madras, India.

Email:{anji,djram}@lotus.iitm.ernet.in

Abstract

The need for flexible file sharing in distributed systems is increasing in applications such as calendar management, collaborative editing of documents, collaborative software developments etc. The file sharing policies required in these applications are often very different from the traditional read/write policies. Hence, a flexible way of specifying and implementing sharing policies on individual files in file systems is required. We propose a distributed object-based system model of constructing file systems. The Object-based system model is based on a pattern called `FlexiFrag`. We show how a distributed object-based system and in particular distributed file system can be constructed using the pattern in a flexible way.

Keywords: Distributed file systems, CSCW, sharing policies, Design patterns, calendar management.

1 Introduction

Traditional file system sharing policies are globally applied to all files in the system and are based on simple read, write, and execute permissions to owner, group and others [1, 4]. However, many advanced applications such as collaborative editing require more sophisticated file sharing policies. An example is the calendar management application. In a typical calendar [2, 7], file data can be a schedule, a proposal, a note, or personal data. The different sections of the calendar file often require different sharing policies (refer to

Figure 1). In a collaborative software development environment, the programmer would like to have different sharing policies on his program files. He might allow modifications to his file with some constraints attached. Similarly, in case of collaborative document preparation, an author, while editing a section of the document may require to read other sections of the document being edited by others.

Such advanced collaborative applications require flexible file sharing policies on different segments of individual files and on different files. The system must provide adequate support for specifying access policies based on the semantics of the applications. It should also be possible to dynamically change the access policies. Further, it may be required to form groups or subgroups among users with separate access permissions.

To satisfy the growing needs of such advanced applications, the file systems should be flexible, general, and extensible. Conventional file system designs are neither general nor extensible. Object-oriented frameworks [11, 14] were developed to construct file systems to make them easily extensible. The object-oriented frameworks capitalize on the object-oriented techniques of data abstraction, data encapsulation, inheritance and polymorphism. They allow new features of file systems to be incorporated dynamically as and when required without much effort.

Since each individual file is governed with its own sharing policies, the abstraction of the file as an object and abstraction of sharing policies as a *meta-object* provides a flexible way of linking a *meta-object* with an object. The abstraction of a file as an object also helps in handling issues such as file migration and file replication.

In this paper, we have identified a design pattern named `FlexiFrag` in constructing the distributed file systems for providing the functionality of flexible file sharing, file replication and migration. The `FlexiFrag` pattern is based on three generic objects called *filter object*, *fragment object*, and *reflector object*. Since replication and migration are general problems in distributed systems, the `FlexiFrag` pattern can find widespread use in constructing distributed systems.

The rest of the paper is organized as follows. Section 2 describes the functional requirements of distributed file systems in detail. In section 3, the `FlexiFrag` design pattern is described. In section 4, we briefly describe the generic components required in building distributed file systems using `FlexiFrag` pattern to provide the required functionality. Detailed implementation of distributed file system using `FlexiFrag` pattern is described in section 5. Finally we conclude with a discussion on future extensions.

2 Functional Requirements of Distributed File Systems

Currently file system designs provide uniform global file sharing policies based on read, write and execute permissions for the owner of the file, user group and general public. However, there are a number of advanced applications such as collaborative editing of documents, collaborative software development and calendar management which require a rich set of file sharing policies among groups of members. For example, the application may require that a specific user be allowed to append data to a file. Similarly, it may be required that a user who has write permission on a file is prevented from deleting the file. In other words, the system must be able to provide file sharing policies as required by the semantics of the application.

Calendar management application can be looked in detail to understand the nature of such applications. There are several kinds of calendars representing the schedules of a person, the announcements of technical conferences, the open events such as seminars etc. In this paper, we refer, in particular, to a calendar used by professionals in an information office, wherein each person is provided with a workstation connected to a network. This type of electronic calendars provide different types of information shared among several users across the network. The kind of access to a particular type of information depends on the role assumed by the user [7] at the time of access. Figure 1 shows the information provided by the calendar along with the types of users and the sharing policies assigned to them.

The electronic calendar used by an executive in an information office is required to provide the following shared information.

- **Schedules/Appointments**: This part of the calendar is used to store scheduled meetings or appointments of an executive in an information office. The information pertaining to the time, duration and agenda of the meeting is specified. In addition to these appointments the owner may keep some of his schedules with his family members and friends as personal.

- **Proposals**: A proposal is a request by a person to seek an appointment with an executive. The caller sends a proposal using a special proposal program to the callee. The proposal to appointment consists of time required, purpose of discussion and caller signature. All proposals are appended to the proposals part

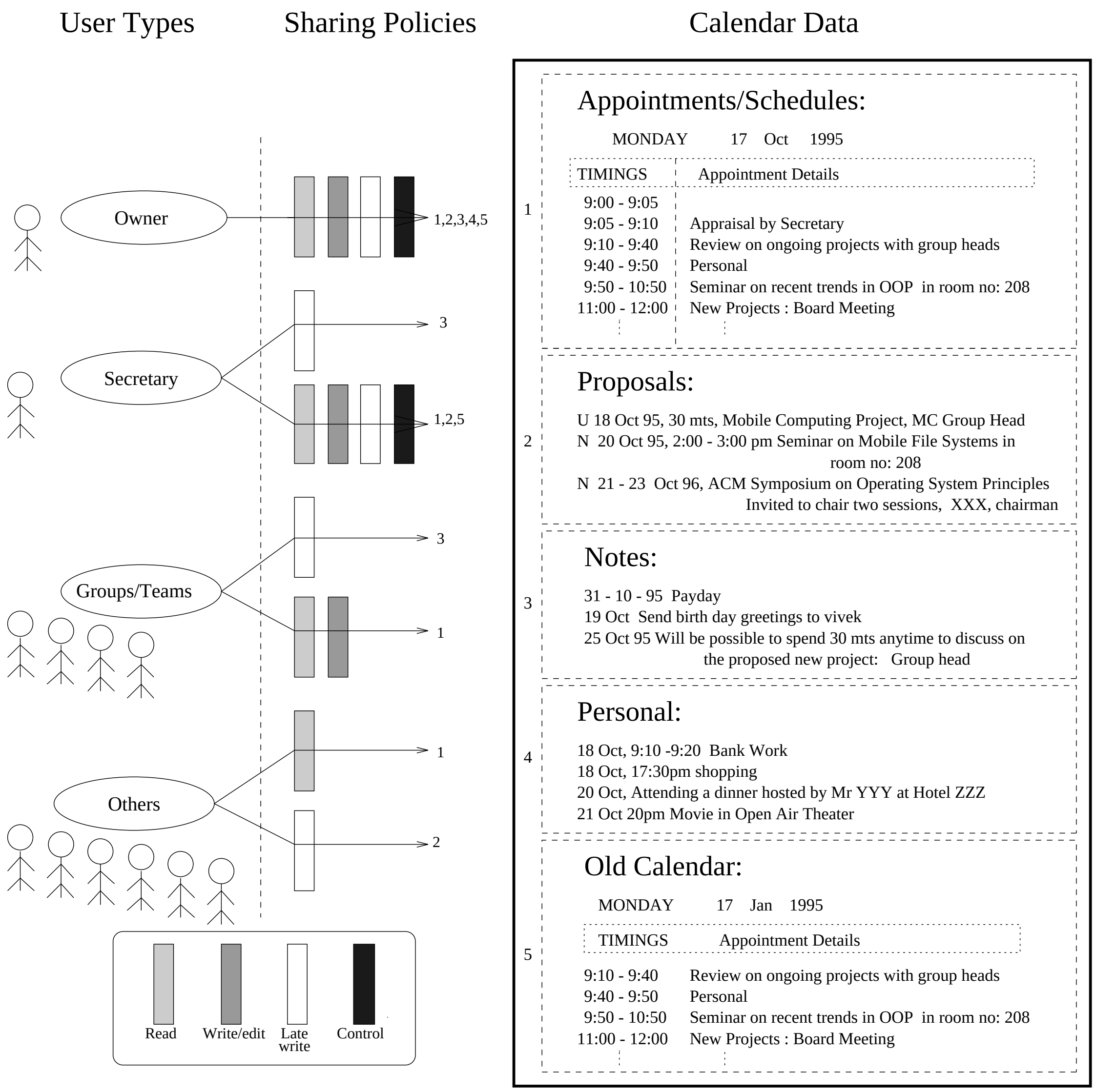

Figure 1: **Required Sharing policies on Calendar data**

of the calendar.

- **Notes**: The notes are unscheduled reminders to the owner of the calendar. These include booking gas, sending greetings, payday intimation etc. Others can also send notes calling for unscheduled discussions.

- **Personal data**: It is purely personal to the owner and strictly non-sharable with all others. The related time of these events is marked simply `busy` or `personal` in the schedules part of the calendar.

- **Old Schedules/Appointments**: It is maintained for the purpose of knowing the details of some events like whom the owner met and their addresses and other details, what was discussed etc. It also helps in recording birthdays, dates of seminars etc. It is the past copy of the schedules of an executive.

The above information can be stored as a single calendar data or can be stored as different segments with different access policies for each, as discussed below.

2.1 Sharing Policies

Different operations are performed on different segments of the calendar. Users assume different roles and access to different segments of the calendar is based on the specific role of the user. The following roles are specified for the user in the context of calendar management.

- **Owner**: The person who owns the calendar. The owner is allowed to perform all possible operations on the calendar data.
- **Secretary**: Owners may have a secretaries. The authority is delegated to the secretary by the owner to confirm or reject appointments on their behalf. Hence, the secretary should have the authority to read and modify any segment of the calendar except personal data.
- **Group/Team**: The members at the same hierarchy level form a team or group. Group permissions can be specified on shared segments of the calendar.
- **Public**: All others called public are allowed to read schedules and send proposals for appointments. The proposals must be confirmed or rejected by either the owners or the secretaries. The public is not allowed to read the proposals data.

The above four roles require different sharing policies on different segments of the calendar as shown in Figure 1. The owner of the calendar specifies the sharing policies on segments of the calendar applicable to different users. The owner can also change these sharing policies dynamically.

It is possible to construct such flexible distributed file systems by abstracting the sharing policies as meta-objects and files as objects and dynamically creating a link between the file object and the policy meta-object. We propose a design pattern based on file-fragment objects, policy meta-objects, and reflector objects to achieve the goals of flexibility and migration in the context of distributed file systems. Since flexibility and migration are generic problems in the context of distributed systems, the proposed design pattern could be considered generic to distributed systems.

3 Design Pattern for Flexible Distributed File System

Design patterns achieve reuse of software architectures at a level higher than source code [5, 15]. The `FlexiFrag` pattern has been identified to solve the generic problem of flexibility and migration in the context of distributed systems. We describe the `FlexiFrag` pattern using the format suggested by Gamma et al [5].

3.1 Intent

The pattern simplifies the validation and delivery of messages to a particular object in a group of objects which are logically related in a network of workstations environment.

3.2 Motivation

In distributed collaborative applications, users collaborate through the use of shared data. These shared data consists of multiple fragments. Each fragment is required to have its own sharing policies and the user should be able to change dynamically these shared policies. These fragments can also keep moving from one location to another location. From the view-point of the user, it is very difficult to keep track of the location of a fragment. Hence, flexible and elegant handling of shared policies and migration is required in distributed systems.

Similar problems exist with several other services present in distributed systems. Each individual service would like to have its own policies. The agents providing the service keep migrating from one node to another.

The flexible specification of policies on services and the migration of services can be considered as generic problems encountered in distributed systems. We propose a design pattern to handle these common problems in distributed systems. Each fragment is treated as an object and its sharing policies are abstracted into a meta-object. The fragment object is dynamically linked or unlinked to its meta-object. Whenever a fragment object wants to migrate, it informs the reflector object about its destination. Thereafter, the reflector object will redirect the messages to the new location.

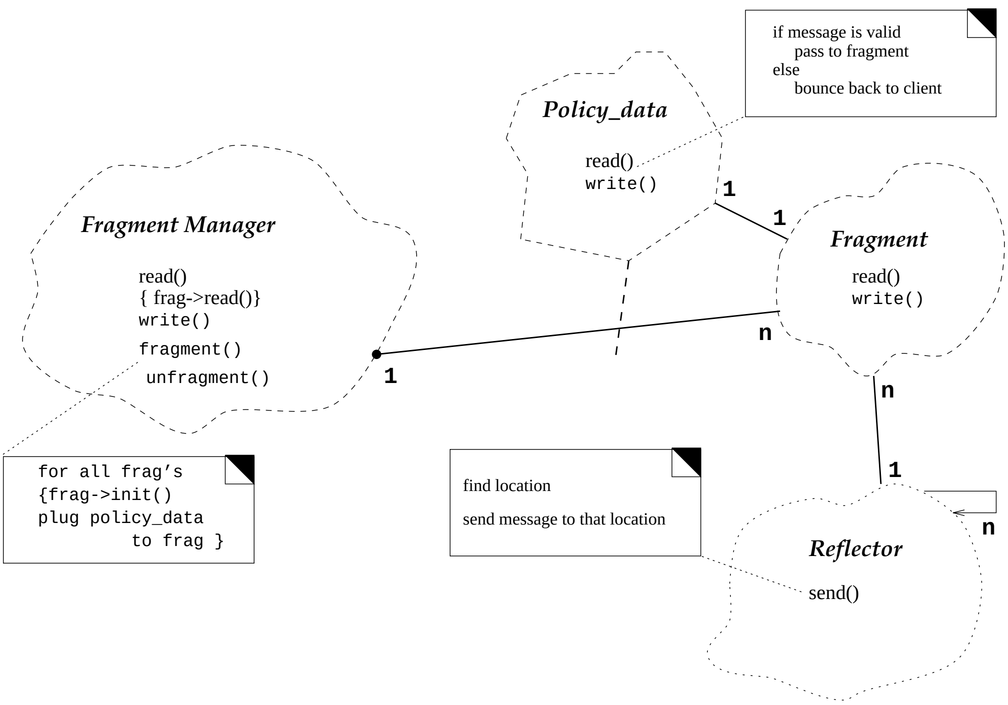

Figure 2: **Structure of the `FlexiFrag` pattern**

3.3 Applicability

`FlexiFrag` pattern can be used where:

- Individual segments of an object require different sharing policies
- It is required to logically group similar services to provide flexible maintenance
- Individual segments of an object require concurrent access
- An object is required to be distributed across the network of nodes

3.4 Structure

The structure of the `FlexiFrag` pattern is illustrated in Figure 2 using the Booch[6] class diagram. In the figure, dashed clouds indicate classes, an undirected edge with a small bullet at one end indicates a composition ('has' a relationship or aggregation), a self-loop indicates that more than one instances of the class may collaborate with one another, a dashed edge indicates the interception of messages by an instance of the class, and an undirected edge indicates an association.

3.5 Participants

The following are the participants in the `FlexiFrag` pattern.

Fragment Manager: It is the control part of the fragmented object and identifies its fragment objects and their corresponding locations. Its interface is equal to all the fragments interface put together and visible to the user.

Fragment: It is the data part of the fragmented object and provides interface to access the data. Its presence is known only to its fragment manager. The interface is invisible to the user. The fragments keep moving between the nodes.

Policy-data: It holds the access policies, replication policies etc. of an object. These are dynamically linked and de-linked to the fragment objects. The policies can also be changed dynamically.

Reflector: It holds the names of the objects and their respective location addresses to which they have migrated from the server. The object at the time of migration leaves its name and location address of the destination with the reflector. Then onwards the messages of migrated objects are reflected to their respective locations.

3.6 Collaborations among the participants

Collaborations among the participants are given in Figure 3.

- The read and write messages are routed to their respective locations by the fragment manager. These messages are further intercepted by the respective policy-data objects for validation of messages. The valid messages are passed to the fragment and the invalid messages are bounced to the client. The result is directly delivered to the client by the fragment. It is shown in Figure 3(a).

- In case of migrated fragments, the messages are routed to the present location of the fragment by involving one or more reflectors. As shown in Figure 3(b), the valid messages are delivered to the fragment which sends back the results.

- In case of valid migrate_message, the fragment sends the location information to the reflector of the present location as shown in Figure 3(c).

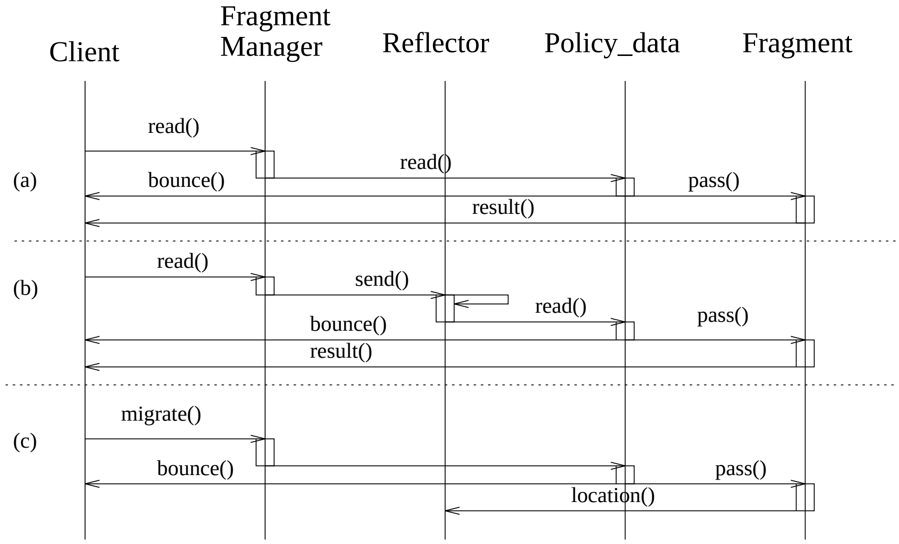

Figure 3: Collaborations among the participants

3.7 Consequences

- The pattern frees the user from objects location management.
- The pattern allows fast access to migrated objects.
- The pattern provides concurrent access on different parts of an object.
- Access policies can be specified independent of the object data.

3.8 Known Uses

- The Constraint Meta-Object Model [8] was proposed to capture collaborative design activity, where each designer works on his own individual workstation and collaborates with other designers' workstations connected through a local area network. The collaboration in this domain is captured as constraint satisfaction on design spaces and constraint propagation among related design spaces. Each design space consists of design objects and the associated constraint space, a set of constraint meta-objects. A dynamic link is created between the design object and constraint meta-object.
- Distributed shared memory, Distributed storage, Cooperative file caching, Distributed directory

4 Use of FlexiFrag Pattern in Structuring Distributed File Systems

As explained in the previous section, the `FlexiFrag` pattern can be used in structuring the cooperative applications such as calendar management. In this section, we explain how the `FlexiFrag` pattern can be used in the calendar management.

The Calendar can be considered as a fragmented object consisting of schedules, proposals, notes, old calendars, and personal data segments as shown in Figure 1. These segments can be treated as fragments of an object. Hence, a calendar object can be considered as a *fragmented object* which consists of multiple fragment objects.

As explained in section 2 each individual segment requires to specify its own sharing policies for different types of users. The owner of the calendar would like to change these policies dynamically independent of its data. Hence, the abstraction of sharing policies into a meta-object will provide a flexible way of specifying and changing these policies by dynamically creating a link between fragment object and meta-object.

The fragment object can migrate from one node to another. In order to deliver a message it is required to know the present location of the object. We use the concept of *reflector object* to keep track of the location of the migrated objects. When a request comes to the object to migrate, it leaves the destination address with the *reflector object*. Henceforth the *reflector object* will redirect the messages to that address.

The policy meta-object intercepts the messages sent to a fragment object and performs the required action. It is for this reason, we model meta-object as a filter object. The filter object model provides language constructs to link or de-link an object to its client object.

In the rest of the section, we present the concepts of *message filtering*, *fragmented object* and *reflector object* models to provide insight into the `FlexiFrag` pattern.

4.1 Filter Object Model

We briefly describe the *filter object* model developed in [12]. In the conventional object model, a message generated by an object (called *source object*) is directly delivered at the *target object* which invokes the corresponding method. This method consists of both message control code and message processing code. Message control code performs

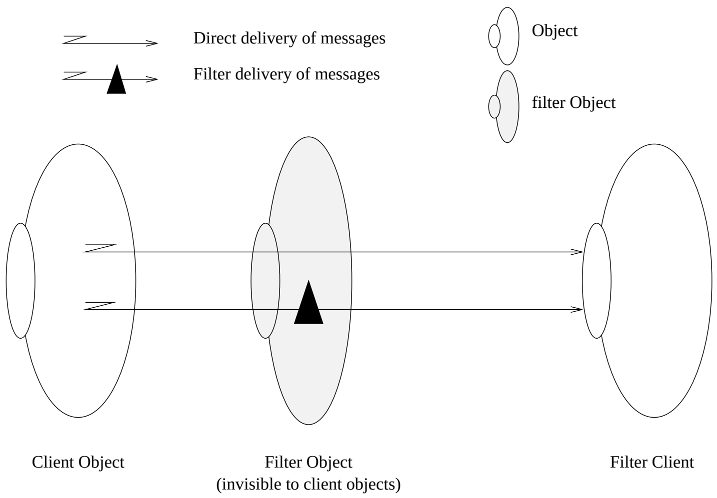

Figure 4: **Filter Object Model representation**

intermediate message manipulations such as validating the message etc. To abstract out separately the message control part of the code, a filter delivery model was proposed. In the model an intermediate object known as *filter object* is inserted between a source object and a target object as shown in Figure 4. The function of the *filter object* is to intercept and manipulate messages sent to the target object (called filter client) through its methods. The *filter object* supports both `upward` and `downward` filtering to facilitate interception of both `upward` and `downward messages`.

The following are the basic principles of the *filter object* model.

1. To achieve message filtering, a filter relationship is established at `class` level between a client `class` C and a filter class F using *filter* operator. The filter relationship enables an instance of filter class F to intercept messages sent to an instance of its client class C.
```
class C { ... };
class F : filter C { ... };
```

2. A *plug* operator is used to bind the *filter object* and its client object at run-time.
```
plug co fo; // co is client object to filter object fo.
```
An *unplug* operator is used to break the filter relationship between objects.
```
unplug co;
```

3. Filter methods are defined as part of the filter class. These methods are automatically invoked when their corresponding methods in the client object are intercepted by the *filter object*. `Upfilter` methods are defined to intercept and manipulate the

upward messages going towards the filter client. Where as `downfilter` methods are defined to intercept and manipulate the return values from the filter client.

4. Two standard beta messages `enable` and `disable` are provided to mark the commencement and the termination of a filtering action respectively.

5. On successful filter action, the message is passed onto the client object by an operator *pass*. Otherwise a return value is bounced to source object on behalf of its client using *bounce* operator.

6. Filters can also be nested i.e., a filter can be designed to act as a filter to another filter.

4.2 Fragmented Object Model

The *Fragmented Object* (FO) [3] is composed of multiple fragments. A fragment is part of an object which is similar in structure to its constituting object. The fragment also has state, behavior and identity. A fragment's state is made up of the values of its attributes. A fragment's behavior is an interface that the fragment provides to its FO. Fragments are private to its FO and invisible to the outside world. As far as the public interface is concerned, the FO is indistinguishable from an unfragmented object.

The fragments of FO are distributed across the network of nodes. Multiple fragments can exist on a single node and the control structure is used to manage them. The fragments provide network transparency. Figure 5 shows the internal structure of an FO.

The FO presents two interfaces depending on the abstraction required by a user. For a user, the object presents a user interface(abstract view). In the abstract view, the FO is indistinguishable from an unfragmented object. The objects present a designer interface (concrete view) for a designer of the object. The FO is viewed as being composed of loosely coupled fragments in the designer's view.

Fragment interface consists of two interfaces: (1) `Private Interface`: It is made up of data members and methods which can be accessed by the members of the fragment. (2) `Public Interface`: It is made up of methods that are available to the designer of the object but not available to the user object.

There are two kinds of fragmentation. (1) `State Fragmentation`: The data members associated with an object are fragmented. (2) `Control Fragmentation`: The control

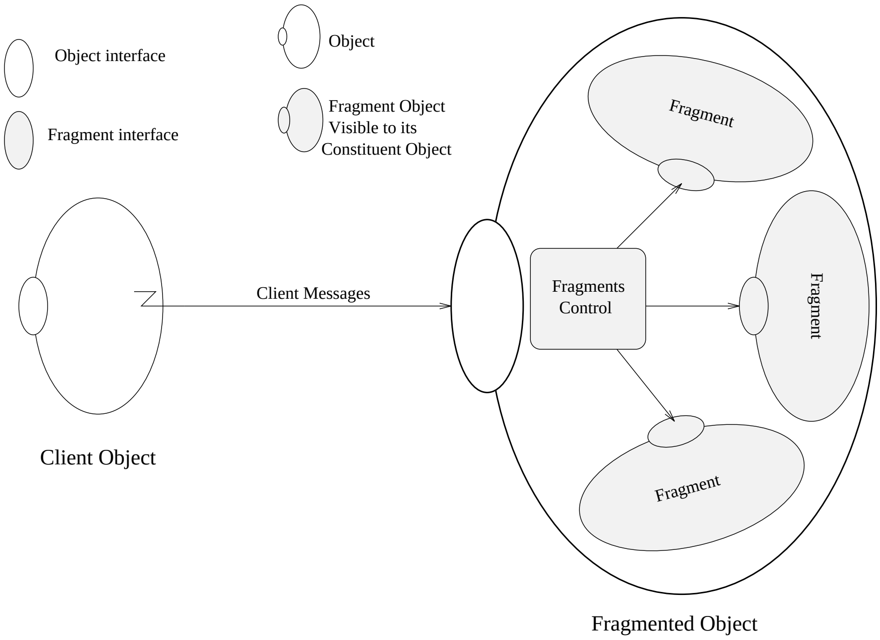

Figure 5: **Fragmented Object Model representation**

members that can be invoked on an object, are fragmented. That is, some methods are located on one node, while the rest of the methods are located on other nodes.

4.2.1 State Fragmentation

The state fragmentation can be provided in two ways.

Horizontal State Fragmentation(HS): Here the data members of the object are grouped and each group goes to a fragment. No data member is spliced. For example, consider a class FOO defined as follows

```
class FOO {
      private: int bar[1024]; float foobar[2048];
      public: void readbar(); void readfoobar(); }
```

The above class can be horizontally fragmented into two fragments frag1 and frag2.

```
fragment {
  private: int bar[1024];
  public:  void readbar();
           void readfoobar();
    } frag1;
```

```
fragment {
  private: float foobar[2048];
  public:  void readbar();
           void readfoobar();
    } frag2;
```

Vertical State Fragmentation(VS): Fragmenting of aggregate types is allowed, but not of scalar data types. For example, when an object of FOO class is vertically state fragmented, the two resulting fragments frag1 and frag2 are:

```
fragment {
  private: int bar[1..512];
      float foobar[1..1024];
  public: //same as foo
  } frag1;
```

```
fragment {
  private: int bar[513..1024];
      float foobar[1025..2048];
  public: //same as foo
  } frag2;
```

`Hybrid state fragmentation` is also possible where fragments consist of some grouped data members and also some aggregate type data members split vertically.

4.2.2 Control Fragmentation

The Control fragmentation is of two types. (1) `Horizontal control(HC)`: Here the methods of objects are grouped into fragments. (2) `Replicated control(RC)`: All the fragments contain all the methods of FO.

4.3 Reflector Object

In this subsection, we propose a new object called *reflector object* to handle the messages of migrated objects in object-based distributed systems.

The *reflector object* maintains the list of migrated objects and their respective location addresses to which they have migrated. On receiving a *migrate_ message*, the object leaves its name and the location address to which it is migrating with the *reflector object* at the present location. Thereafter, the messages of migrated objects are redirected to the respective locations by the *reflector objects*. On the other hand, the return messages are directly delivered by the respective objects at the client objects.

For example, consider the situation depicted in Figure 6. Assume an object O is with the server A. On receiving a *migrate_ message* to server B, the object O leaves the server name B with the reflector object ROA. Further, on receiving another *migrate_ message* to server C, it leaves the destination C with the reflector object ROB. Now, if a client object CO wants to send a message to object O, its server D knows that the object is with the server A and sends the message to server A. On receiving the message by the

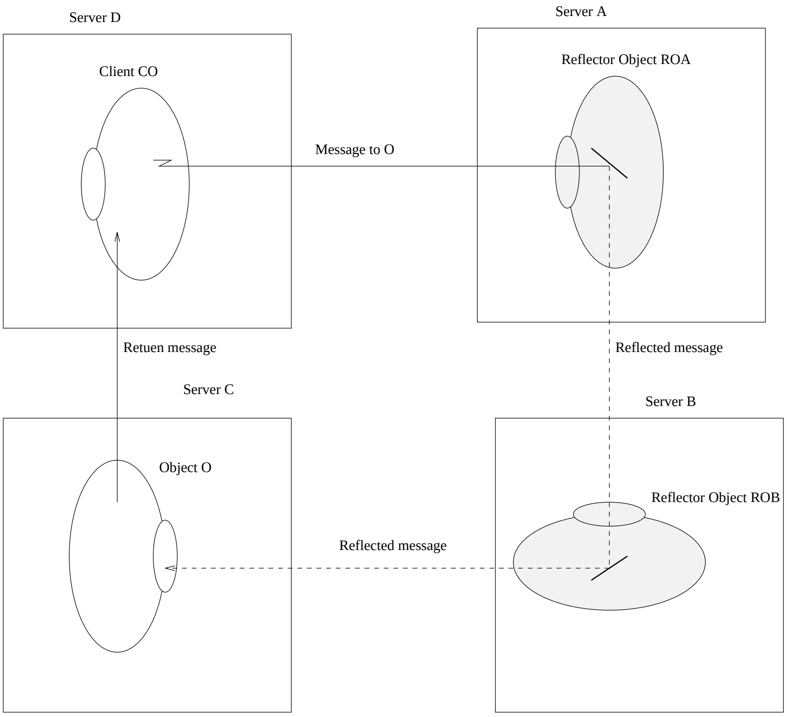

Figure 6: **Reflector Object model**

server A, it is redirected by the reflector ROA to the server B. Further, at the server B the reflector ROB redirects the message to server C. The return message, if any, is directly sent to the client CO at the server D. It is possible to reduce the redirection chain by updating the latest server with the reflector objects.

5 Implementation of Distributed File System

In our implementation of distributed file system using `FlexiFrag` pattern, the co-operation between the objects is based on delegation of authority to named objects called *filter objects* to intercept and manipulate the messages sent to an object. The delegation of authority is selective and dynamic.

In our approach to object-based distributed file system, a file has been abstracted as an object and file access policies are specified as a *meta-object*. The object delegates the authority to the *meta-object* to intercept and manipulate its messages. The access policies are separated from the object data. Hence the `FlexiFrag` pattern helps in changing the access policies of an object without effecting the object data. That is the *meta-object* can evolve itself. The *meta-object* can implement its own replication policies etc. and can change them dynamically. The access policies can also be changed while the data object is being accessed. The file system broadly consists of three types of objects namely Data Objects, Access Policy Objects and Other Policy Objects. We describe in detail the implementation of these three types of objects. The implementation details are shown in Figure 7.

5.1 Data Objects

In our implementation of calendar, we have the calendar object and its fragment objects namely proposals object, schedules object, notes object, personal object and oldcalendar object. The fragments are created by grouping the respective data items. As an example all proposal data items are grouped into a proposals fragment. Further, each fragment consists of its own methods to access the data. Hence, horizontal state and horizontal control fragmentation is followed while fragmenting the calendar object. The fragment objects are created when the calendar object is fragmented by sending the *fragment_message* to the calendar object. The fragmentation is achieved by attaching the *fragment manager* (described below). Here the authority is delegated to the *fragment manager* to direct the messages to fragment objects rather than to the calendar object, as shown in Figure 7. Once the calendar object is unfragmented, the messages are directly delivered to the calendar object rather than to its fragments. The *fragment_message* and *unfragment_message* are sent either by the owner or by the secretary. Fragmentation of calendar object will facilitate different types of sharing, caching and replicate policies on different segments of the calendar data. All these operations are transparent to users.

5.2 Access Policy Objects

These objects are instantiated from the classes which are filters to other classes. The access policy object encapsulates the sharing policies for all user types. The access policy object captures the messages sent to the data object and based on the senders identity access to the data is permitted. If the user is allowed to perform the requested operation, the message is passed onto its data object. Otherwise the message bounces back to the sender.

The sharing policies are provided in the form of an access list. The access list is supplied to the access policy object at the time of instantiation of the access policy object. The list mainly consists of type of users and whether the object is available or hidden to

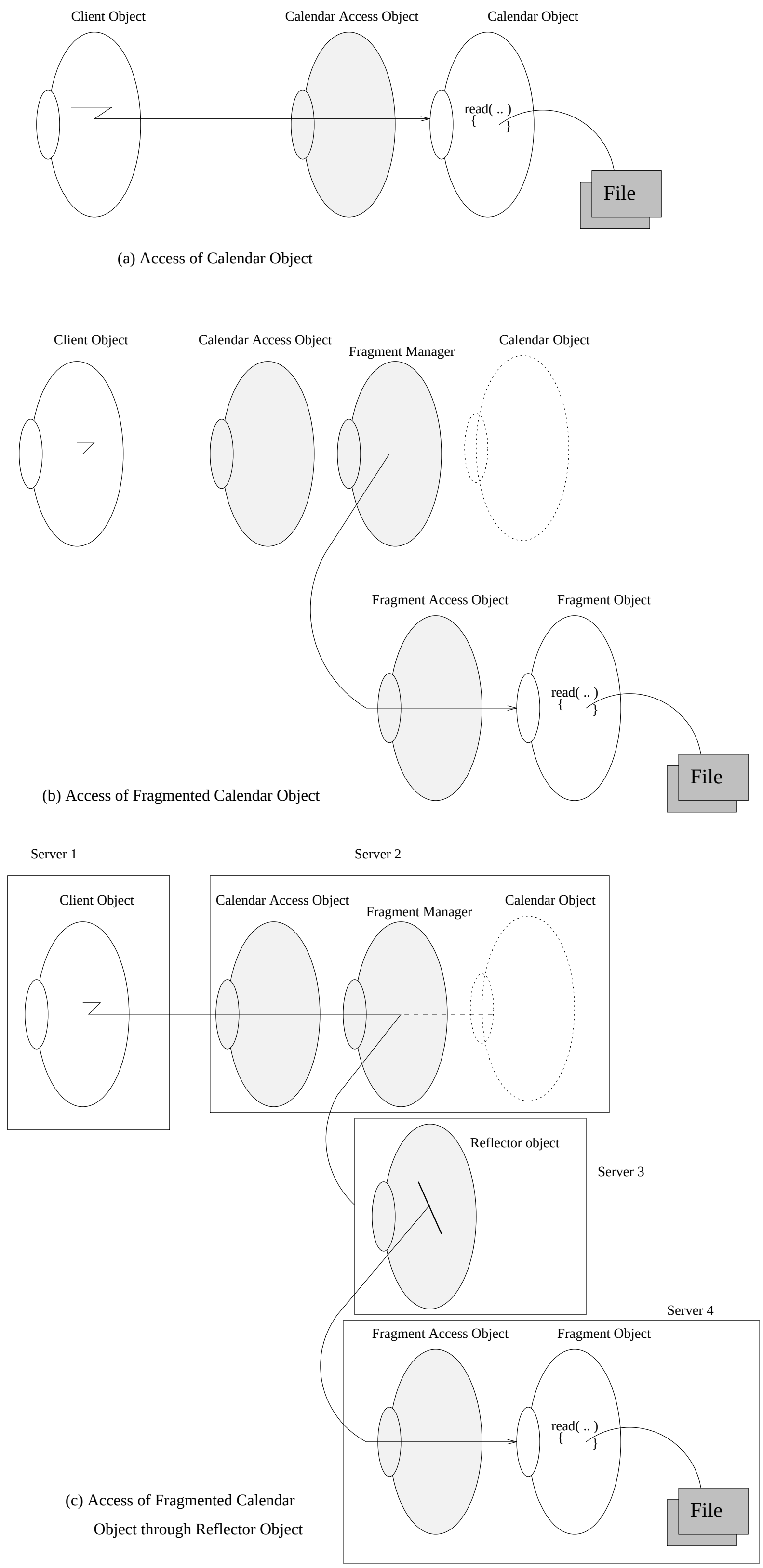

Figure 7: **Implementation of file access mechanisms**

those users. Further the owner can modify the sharing policies dynamically by sending appropriate messages namely add-group(..), delete-group(..), change-access(group,..) etc.

Our implementation has seven access policy objects: One for the unfragmented calendar object, one for each of the five fragments of calendar object and finally one for the fragment manager. The access policy object used for unfragmented calendar object cannot be plugged when the calendar object is fragmented. This is because the calendar is fragmented by plugging the fragment manager to the calendar object. It is not possible to have the same filter to act as a filter to the data object at one place and act as a filter to the filter of the same data object at another place. Hence it is required to have a separate access policy object for the fragment manager which provides the same functionality as the access policy object of the unfragmented calendar object. The advantage of having access policy object to calendar object is that the owner can permit or deny access to the calendar as a whole to a particular role.

The fragments sharing policies are encapsulated in the respective access policy objects. The fragment access policy object captures the messages sent to its fragment object and validates them based on the access permissions of the sender. Several operations are possible on the fragment data. Each of these operations have their own sharing policies. Different types of users are allowed to perform different operations on fragment data based on their roles. For example, in the case of proposals data in calendar management, owners and secretaries are allowed to perform read and write operations where as public and team members are allowed to append data to proposal fragment.

5.3 Other Policy Objects

This group of objects consists of a fragment manager, different caching scheme objects and different replicate objects. The implementation of fragment manager and replicate object are explained here.

Fragment Manager: As discussed in the previous section, fragmented object model mainly consists of control and data fragments. The control fragment implements object interface and keeps the addresses of the nodes where the fragments reside. The control part of the fragmented object is abstracted as a filter to the calendar object. Hence, the main function of the fragment manager is to capture the messages to be delivered to the calendar object and find which fragment object is to be selected and direct the messages to that particular fragment object.

Fragment manager uses the fragment address list. The fragment address list consists of node addresses on which the fragments reside. The fragment list is passed to the fragment manager at the time of instantiation of the object. If the list is not passed, the list can be dynamically created at the time of instantiation of the fragment manager by the system itself.

Replicate object: In *fragmented object model* every fragment consists of both control interface and data. The fragment as a whole (both control and data) is replicated. It allows parallel accesses to fragments by several users which reduces access time and improves the throughput of the system. A ShadowObject [13] also called a replicate object is used to manage the replicas of the fragment. The hide and export constructs are used to export the individual replicas of the fragments or to hide them from external clients. During read replication, the replicas are exposed and to achieve writes on the replicas, they are hidden. Thus replication is handled transparently.

5.4 Performance Study

In our implementation of object-based file system using `FlexiFrag` pattern we have abstracted a file as an object and file access policies as a *meta-object* and *meta-object* is implemented as a *filter object*. These objects introduce run-time overheads for file operations. The run-time overheads consists of access permission checks, function invocation in objects and passing of parameters between objects. The overheads measured on SUN4 architecture with NFS file system are shown in Table 1. The overheads are independent of the file size.

6 Advantages

The following are the advantages provided by our `FlexiFrag` pattern as applied to structuring of file systems.

1. Sharing of data based on application semantics is possible.

2. The system also allows multiple shared policies on the segments of an object. Multiple shared policies are required in applications like calendar management, collaborative document preparation and collaborative software development wherein

Table 1:

Performance Overheads

File size (KB)	NFS File (μ s)	Calendar Object + Access Object (μ s)	Calendar Object + Access Object +Fragment Manager +Fragment Object +Fragment Access Object (μ s)
1	76.26	101.05	147.68
2	77.84	101.98	148.99
5	76.93	101.62	147.2
10	77.25	101.16	147.19

different categories of users/authors share different segments of a file and also different files.

3. Users belonging to different system groups can form subgroups and decide the access policies for the subgroups. This subgrouping is essential in any organization where a user is involved in more than one project.

4. In our design, each fragment can have its own access permissions in addition to the protection provided to the object as a whole. In case of multiple groups of authors cooperatively writing a text book, all members of a particular group require write permission to their part of the text whereas other groups require read permission on the same part the text.

5. The flexible approach to sharing of files can be provided either at system level or user level.

6. Each component provided in our system evolves independently and can be flexibly plugged into the system. All these components are reusable.

7. As the object is fragmented, multiple clients can simultaneously access different fragments of the object. As different users access different fragments, the conflicts are less because the users read or write only to their fragments rather than the object as a whole. Hence, this improves the throughput.

8. As systems are still evolving with new applications such as mobile computing, multimedia etc., new requirements will arise. These new requirements can be

easily incorporated into our approach.

7 Related Work

Several systems have been designed to enable file sharing among a number of users of network of workstations [4, 10]. These systems are based on sharing policies of read, write and execute permissions, applied globally to all files.

LOCUS, NFS and SPRITE systems use the same file sharing semantics used in UNIX operating system [4] to provide upward compatibility with UNIX as these systems designers regard UNIX as a model operating system.

Coda file system and its descendant Andrew file systems [9] provide two levels of sharing one based on what can be done and the other based on who can do it. Venus provides UNIX mode of sharing semantics based on read, write and execute permissions on individual files. Where as vice uses access lists mechanism to provide entry into individual directories.

Ni and Goscinski[10] suggest traders (extension of RHODOS trader) cooperation to enable object sharing among a number of users in homogeneous distributed systems. The owner of object exports it to make it visible to others and sharable by others. While exporting the owner specifies certain conditions. Whoever satisfies these conditions can import the object. It is the task of the trader to provide all these services. The trader cooperation is too simple to model applications such as calendar management. Our system is more comprehensive in providing flexible sharing policies required in applications such as calendar management.

8 Conclusions

In this paper we have proposed a design pattern to implement flexible sharing policies on different segments of a file as demanded by advanced distributed cooperative applications such as calendar management, collaborative document preparation etc. The motivation for the present work is to make data sharing as flexible as possible and allow rich set of multiple sharing policies to exist simultaneously on the same data object.The pattern is based on the concepts of *filter object*, *fragmented object* and *reflector object* models. As a part of the pattern, we have developed reusable components for object access, object fragmentation, object replication, and object migration. Each of these components can be dynamically plugged individually as per their requirements. The flexibility in using design pattern for data sharing in distributed file system was discussed in the context of calendar management application. The approach can be extended to object migration in the context of mobile environment. Our scheme results in construction of flexible and scalable distributed systems at the cost of slight decrease in performance.

References

[1] Andrew S. Tanenbaum *Distributed Operating Systems* Prentice Hall Inc., 1995.

[2] Christine M. Kincaid, Pierre B. Dupont and A. Roger Kaye *"Electronic Calendars in the Office: An Assessment of User Needs and Current Technology"*, ACM Transactions on Office Information Systems, Vol. 3, No. 1, January 1985, Pages 89 - 102.

[3] Dinesh T. *Fragmented Objects*, Under Graduate's theses, Dept. of Computer Science and Engineering, Indian Institute of Technology, Madras, India, June 1994.

[4] Eliezer Levy and A. Silberschatz *"Distributed File Systems: Concepts and Examples"*, ACM Computing Surveys, Vol. 22, No. 4, December 1990, Pages 321 - 374.

[5] Erich Gamma, Richard Helm, Raph Johnson and John vlissides *Design Patterns: Elements of reusable Object-Oriented Software*, Addison-wesley, 1995.

[6] Grady Booch *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin-Cummings publishers company inc., 1994.

[7] Irene Greif and Sunil Sarin *"Data Sharing in Group Work"* ACM Transactions on Office Information Systems, Vol. 5, No. 2, April 1987, Pages 187 - 211.

[8] Janaki Ram D., Vivekananda N., Srinivas Rao CH., and Krishna Mohan, N. *"Constraint Meta-Object: A New Model for Distributed Collaborative Designing"*, IEEE Transactions on Systems, Man and Cybernetics, Vol. 27, Part A, Issue 2, March 1997.

[9] Mahadev Satyanarayanan *"Scalable, Secure and Highly Available Distributed File Access"*, IEEE Computer, May 1990, Pages 9 - 21.

[10] Ni Y and Goscinski A *"Trader Cooperation to enable object sharing among users of homogeneous distributed systems"*, Computer Communications, Vol. 17, No. 3, March 1994, Pages 218 - 229.

[11] Peter William Madany *An Object-Oriented Frameworks for File Systems*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

[12] Rushikesh K. Joshi, N. Vivekananda and D. Janaki Ram *"Message Filters for Object-Oriented Systems"*, Technical Report No. IITM-CSE-DOS-95-08, IIT, Madras, India. (To appear in Software-Practice & Experience)

[13] Rushikesh K. Joshi, D. Janaki Ram *"ShadowObjects: A Programming Model for control Replication in Distributed Systems"*, Technical Report No. IITM-CSE-DOS-95-03, IIT, Madras, India.

[14] Ware Myers *"Taligent's CommonPoint: The Promise of Objects"*, IEEE Computer, March 1995, Pages 78 - 83.

[15] Wolfgang Pree *Design Patterns for Object-Oriented Software Development*, Addison-Wesley Publishing company, 1995.