Object-oriented Framework for Adaptation in a DFS

Arvind Kumar and Mitchell D. Theys
Computer Science Department
University of Illinois at Chicago
{akumar, mtheys}@cs.uic.edu

ABSTRACT

Today's large diversity of network resources, computing resources and heterogeneous devices can lead to a mismatch between network-capacity and the volume of data transferred, or between devices and data (consider transferring a large video clip to a handheld over a congested wireless network). Traditional distributed file systems including NFS [1], AFS [2] and Coda [3], fail to address this issue, as they cannot adapt their services according to environmental constraints. We address the problem of adaptation in a DFS by modeling the file system as an object-oriented framework using persistent distributed objects. Files (persistent objects) in the DFS participate when the system reacts to environmental constraints (e.g. network congestion) and/or user-imposed constraints (e.g. security). For instance, when the user wants to play a video clip stored on a remote server with an untrusted network in between, the system chooses a version of the clip appropriate to the current network traffic, encrypts it over the wire, and plays it on the client. Further, the behavior of our general-purpose framework can also be *specialized* according to specific application requirements, to implement domain-specific optimizations (e.g. modifying concurrent-write semantics).

Keywords: Adaptation, distributed file system, object-oriented, framework, extensible

1. INTRODUCTION

This millennium is witnessing an explosive growth of computing resources, in terms of network resources and heterogeneity of devices. The increasing popularity of handhelds and Web-enabled cell-phones is allowing the mobile individual to access data on the Internet, on the corporate network and at home. The rapid development in mobile digital devices is generating more multimedia-rich content and inflating data sizes. In addition, the massive growth of the Internet is pushing computing to a truly network-centric paradigm.

However, the above technological changes also create potential problems if the network cannot transfer the volume of data within the application-imposed deadlines, or if the device cannot handle the particular type of data transferred. For instance, transmitting a 170-megabyte movie directly to a wireless Web-enabled cell-phone is infeasible, both in terms of the latency/bandwidth issues involved, and in terms of the resource limitations of the cell-phone. Traditional distributed systems cannot modify their services in the presence of dynamic environmental constraints.

Much like a biological system that has to adapt to constantly fluctuating external conditions in order to function smoothly, the operating system (and the file system in particular) also has to respond to the rapidly changing environment and adapt to the application-imposed constraints. A general-purpose mechanism is required to deal with system adaptation that is extensible and customizable according to the application domain. Ideally, the mechanism has to be abstract enough to address the needs of every conceivable situation, including currently unforeseeable applications.

We propose an object-oriented framework for achieving the twin goals of adaptation and extensibility. A proof-of-concept implementation of such a system has been developed in Java, called **ADIOS** (**A**daptive **DI**stributed **O**bject **S**ystem). Its architecture and implementation is described in the rest of this paper.

1.1 Goals and Approach

The goal of our research is to build a general-purpose distributed file system (DFS) that can support adaptation in the presence of system- or user-imposed constraints. Application developers should also be able to specialize the system behavior according to their specific requirements.

Application-aware Adaptation

There are many levels of adaptation, with different degrees of participation from the system and the application [4] [5]. The "application-aware" approach, with a collaborative partnership between the system and the applications in solving the adaptation problem, has been shown to be the most general and effective approach [6]. As a consequence, the application can query the system's current resource-situation, and the system will have access to the application's interfaces.

However, if the file system has to participate in the adaptation process – a process that is highly specific to the type of data involved – it needs to take decisions that are *relevant* to the data, to be effective. For instance, over a low-bandwidth network, a picture can be transferred more quickly if its resolution is lowered; a text file can be compressed using Huffman encoding; and a sound file can be modified by (band-pass) filters. If we used the traditional representation for data – raw byte-streams, or *files* – the approach breaks down, as the type of the data cannot always be inferred by inspection. The fundamental problem is that conventionally data is just a stream of bytes carrying no information by itself, useful only in the presence of associated applications.

Object-oriented Paradigm

To solve the problem, data needs to be bound with its associated behavior. The classic solution to this problem is to use the object-oriented (OO) approach. The OO paradigm encapsulates data with operations applicable on the data. Traditional file systems are comprised of two types of files – *data files* and *applications*. Data files contain information (*state*), and applications know how to use or manipulate that state (*behavior*). An object encapsulating state and behavior can combine the two different but related types of files into a unified self-contained entity.

Framework

To deal with extensibility and specialization, ADIOS uses the concept of a *framework*. A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [9]. Frameworks extend the power of object-oriented programming by providing abstract components that can be extended and/or used for composition to provide richer functionality while retaining a common structure. It is a set of cooperating classes, exploiting the features of inheritance and dynamic binding in object-oriented programming, to make up a reusable design [9] [10]. Frameworks are fundamentally different from class libraries. In an application, "a class library contains abstractions that the application's abstractions instantiate/invoke; a framework contains abstractions that instantiate/invoke the application's abstractions" [20].

1.2 System Overview

ADIOS consists of two parts: a framework specifying an inheritance and collaboration model for persistent objects, and a distributed model for storage. The framework is presented in Section 1.2.1, and Section 1.2.2 summarizes the distributed model.

1.2.1 Framework for inheritance and collaboration of persistent objects

This framework specifies the structural, functional and semantic relationships for objects that reside in the system (see Figure 1). Every entity in the system is implemented as a `PersistentObject - ObjectHandle` pair. The `PersistentObject` is accessible only to the system and is visible to the application through its `ObjectHandle`. The `PersistentObject` interface forms the root of the object hierarchy, and the user can access an instance of a `PersistentObject` (indirectly) only through an instance of the `ObjectHandle`.

1.2.2 Distributed model for storage

The model for data storage abstracts mechanisms for locating and retrieving references to distributed objects. It assumes that the logical space of the system is divided into regions called *domains* for administrative convenience (refer to Figure 2 and 3). The system consists of three components: *nameservers*, *domain administrators*, and *object servers* (illustrated in Figure 4). Distributed nameservers provide a mapping between the logical and physical namespace. Domain administrators manage domain specific activities. Object servers store objects physically on disk (or other storage) and retrieve them.

1.3 The rest of this paper

The rest of the paper documents the framework's class design and its prototype implementation in detail. Section 2 discusses work related to adaptation and extensible systems, done earlier. Section 3 describes the system's architecture. Section 4 discusses our prototype implementation of the model described in Sections 3, and our experiences with the tuning the system. In Section 5, we present our conclusions and the plans for our future work.

2. RELATED WORK

There has been research on providing adaptation in distributed applications, and many solutions have been proposed to achieve this objective, including server-side support, middleware support and client-side action [13]. However, there hasn't been a system-based approach to solve the problem. Very few distributed file systems have tackled the problem of adapting to network variation. Coda [3], which tackled the network variation problem, dealt with a very specific case (disconnected and weakly-connected operation). Odyssey [4] [6] dealt with adaptation in depth, but it was not designed to be an extensible file system that could be customized based on application profiles.

Our approach of enabling the system to take decisions based on the type of data has been investigated before in the context of semantic file systems [15]. A semantic file system "understands" the content of files in the system using special transducers. However, this information is only used for faster means of locating files that are possibly semantically related. It is not used to customize the file system's behavior for those related files, and is not used for adaptation.

Efforts with the SPIN operating system [17] and the Exokernel architecture [14] have produced robust and efficient models for developing flexible and completely extensible operating systems. SPIN allows safe operating system extensions that can change the system's behavior. However, SPIN is a general-purpose operating system that was not meant to deal with the file system-specific problem of adapting to network traffic or device constraints.

Our use of a framework as a guiding concept in system design has been used before to develop other systems. Choices [7] is an example of a framework used to develop a complete operating system. The Choices operating system provides the advantage that the system can be extended and specialized by application developers to improve performance and provide application-specific optimizations. However, Choices does not address adaptation of the system to external factors.

Perhaps the closest research in this area of providing an extensible system that also addresses adaptation is the 2K operating system developed at UIUC [16]. 2K is an integrated operating system architecture that addresses the problems of resource management in heterogeneous networks and dynamic adaptability. However, the degree of extensibility of this system is not evident, and we believe that the minimal set of interfaces that our model defines gives a much higher flexibility to application designers.

3. SYSTEM ARCHITECTURE

We elucidate the architecture of the system in two sections. Section 3.1 details the framework for object relationships between all the classes in the persistence hierarchy. In Section 3.2, the distributed model of storage is presented. The framework is illustrated in Figure 1 using UML notation.

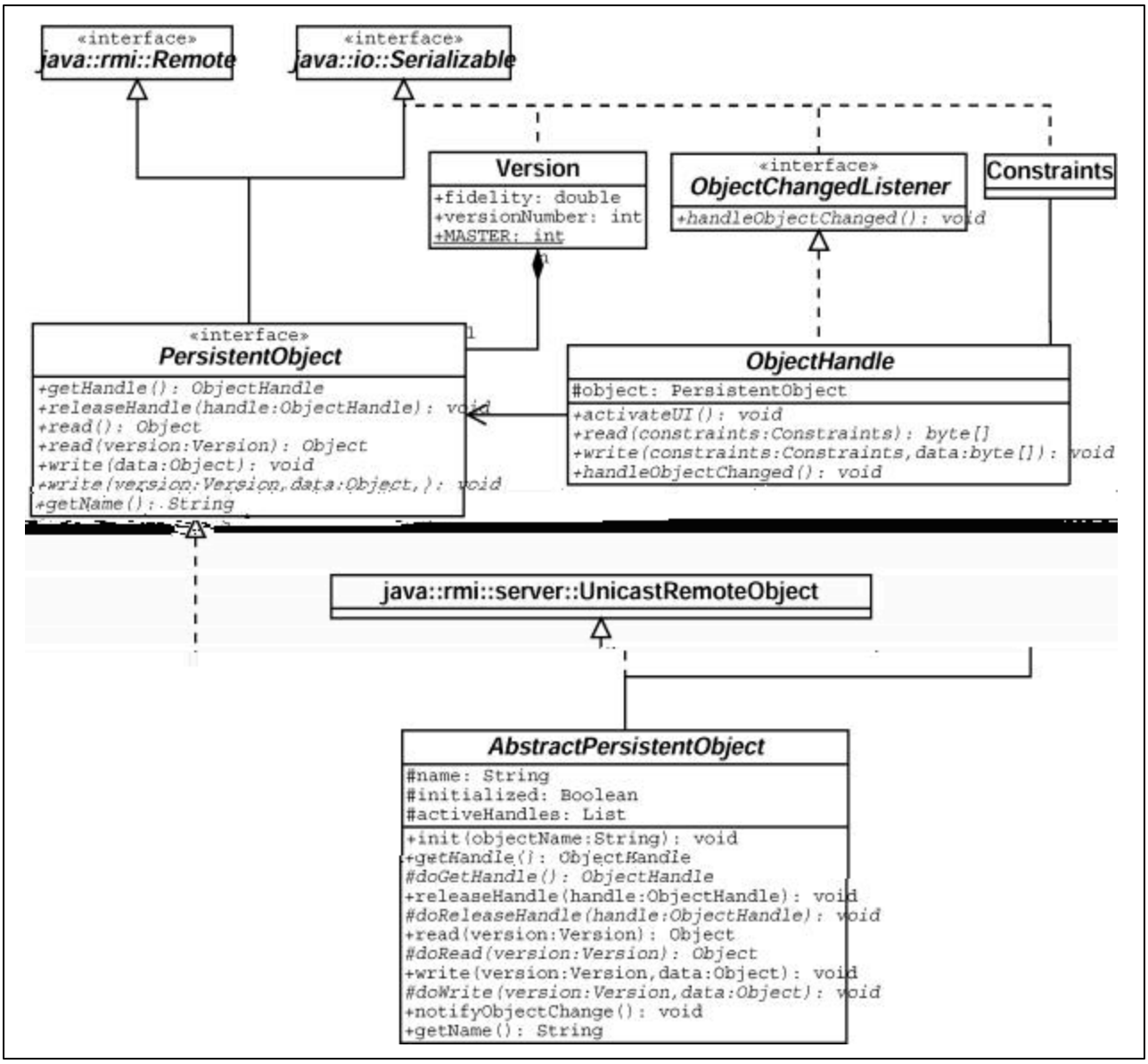

Figure 1:Framework for object collaboration in UML [20] (only the more important methods/classes are shown)

3.1 Framework for Object Inheritance and Collaboration

ADIOS tries to model the conventional file system as an object-oriented system. Therefore, all the data on the file system, originally present as raw byte-streams, has to be represented as objects encapsulating state and behavior, with data files turning into state and applications transformed into behavior. In the object-based system, the following classes are used to architect the design:

PersistentObject

Every semantically distinct collection of data in the system is a `PersistentObject`. The object is *persistent* since its lifetime is independent of the process that created it. The persistent object is typically stored on a remote server (the exact distributed model is described in Section 3.2). Throughout the object's lifetime, there is exactly one physical copy of the object, irrespective of the number of clients accessing it at any given time. Access to the `PersistentObject` is controlled through a handle (which is an instance of the class `ObjectHandle`). When a client wants to use a `PersistentObject`, the system spawns an `ObjectHandle` for the object, which is then returned to the client.

AbstractPersistentObject

According to the framework, `PersistentObject` is a pure interface that specifies only the contract for persistent objects in the system. To help application developers, ADIOS also defines an `AbstractPersistentObject`, a partially implemented subclass realizing the `PersistentObject` interface. This class defines the kernel of most operations and defers the rest of the implementation to subclasses, in the style of a *Template* pattern [18].

This class provides AFS-like update semantics and strong synchronization guards for concurrent access. Typically, application developers would specialize this class (or its derivatives) rather than the `PersistentObject` interface.

ObjectHandle

`ObjectHandle` acts as an access-point to a `PersistentObject`, on the client side. It is similar to a *Proxy* pattern [18] in that it acts as a surrogate/placeholder for the `PersistentObject`, but is different since it does not expose an interface identical to the `PersistentObject` (and hence cannot be freely interchanged). The difference in interfaces arises because the `PersistentObject` is meant for the system, while the handle is meant for the application.

ObjectChangedListener

This interface provides the "hook" for the system to notify that the state of the object has been modified. If multiple users concurrently access a remote object, and one of them modifies the object-state, the system can notify the other users about the state change. This interface includes the method `handleObjectChanged`, which is the equivalent of the AFS' server callback. `ObjectChangedListener` is essentially an *Observer* design pattern [18].

The system can provide a wide variety of semantics by simply controlling the invocation of the callback method. If `handleObjectChanged` is called within the body of a `write`, the `PersistentObject` enforces UNIX-like semantics (writes are immediately visible to server, and all users). A slight variation can produce NFS-like semantics. If the callback is invoked within the body of `releaseHandle`, the object enforces AFS' "write-on-close" semantics. Of course, the framework-implementer can define a completely new semantics too if that proves effective for the application.

Constraints

This class represents the environmental constraints on the system during file access. It is not possible to define `Constraints` as a fully realized concrete implementation, because of two reasons. One, it is highly specific to the application; two, the notion of constraints might change in the future. Therefore, this class does not have any attributes or responsibilities. The application is free to define and specialize this class according to its requirement.

Version

This is a utility class that helps in adaptation. A persistent object can be internally stored as a collection of different versions that are appropriate for different situations. The system can choose a particular version at runtime according to

the environmental profile at that moment. These different versions need not necessarily be stored permanently; they can be generated on demand.

Applications can use an instance of the `Version` class to denote the level of adaptation that is required in a particular situation. The `Version` class contains two attributes, `fidelity` and `versionNumber`. The former represents the closeness of the version to the "master" copy, and the latter is a label for the particular version. Applications can choose between either of these attributes (or a combination of the two) to refer to a version.

`SystemState`

This class encapsulates information about the current system resources, in terms of device resources and network resources. For instance, network parameters like bandwidth, latency and round-trip time to particular nodes can be obtained from this querying this class. Currently, ADIOS has only a very basic, inchoate implementation of this class. However, we expect this component to be critical to applications exhibiting adaptation, and future work will concentrate on developing this class to a more robust and sophisticated layer to query system resources, as described in Section 5.

3.2 Distributed Storage Model

The distributed model for storage comprises of three components: `Nameserver`, `DomainAdministrator` and `ObjectServer`. These components are described in Section 3.2.1. However, before their discussion, we briefly describe the general organization of the file system in Section 3.2.1.

3.2.1 Logical vs. Physical File Space

The file system has a UNIX-like namespace, with the exception that shared files have a special "`/shared`" prefix. Users are allowed to explicitly indicate to the file system that certain files should not be "shared" (for efficiency reasons), that is, they will be present on the local disk and not be placed on a remote server. As a result, other nodes on the network cannot access these files.

The physical organization of the file structure is a hierarchy, divided into domains for administrative convenience (see Figure 2 and Figure 3). Domains are independent from each other, and possess their own set of name servers, administrators and file servers. Faults within a domain are localized to that domain (however, in the event of failure, clients may require that at least the name servers be functional, to parse path components involving that domain). Typically, the name servers are replicated to increase fault tolerance.

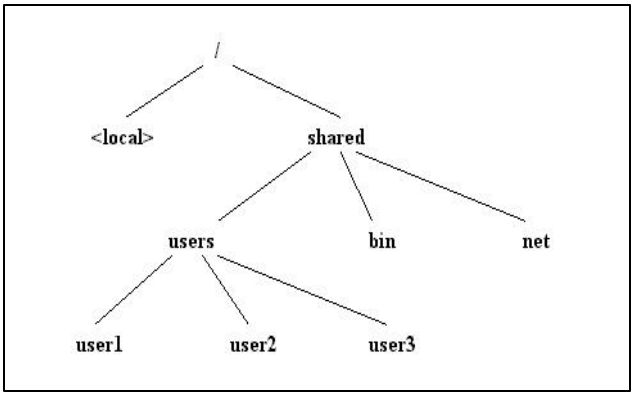

Figure 2: User's view of the logical file space

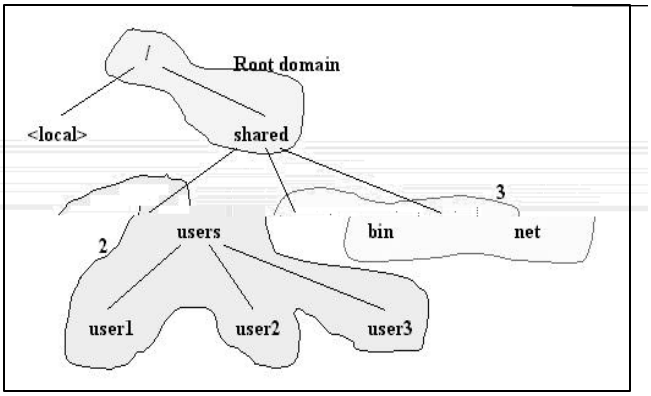

Figure 3: System's view of the logical file space, organized into domains

3.2.2 Components in the Distributed Model

The three components of the model are explained below. Figure 4 illustrates the following concepts.

`Nameserver`

This component is responsible for mapping names in the logical space to their physical locations. The logical namespace is organized like a UNIX file system except for the "`/shared`" prefix for remote files. However, the location of the file is hidden from the user for remote files.

Name servers are "fragmented" so that no single name server knows about the entire file system by itself. The `Nameserver` accepts requests to parse a particular component of a logical name. If the server possesses the information for that component, it answers the request. If it does not, it gives the location of the name server that might be able to handle the request, like the Internet DNS [11]. However, unlike the DNS, the name server does not recursively contact the other name servers to resolve the name fully. The client side of the file system has to construct the path, component by component.

`DomainAdministrator`

This component is responsible for maintaining the integrity of domains in the file structure. Newly created domains have to register with the parent domain's administrator, and should ensure that they have a name server and an object server.

`ObjectServer`

This component is responsible for physically storing persistent objects on the disk and retrieving them. It regulates access to `PersistentObject`s by spawning the `ObjectHandle` for every persistent object on demand, and shipping the handle to the client instead of directly transferring the `PersistentObject`.

`VirtualFileSystem` (VFS)

This component is present as a layer over the native file system. It maps all the file system calls to the ADIOS implementation and provides a unified interface to the clients of the ADIOS system. All the file system calls from the client are routed through this layer. The VFS internally contacts the other components of the distributed model explained above to service the client's request. As a result, clients are not aware of any system component except the VFS. It is to be noted that this VFS is not to be confused with the VFS of BSD and Linux file systems.

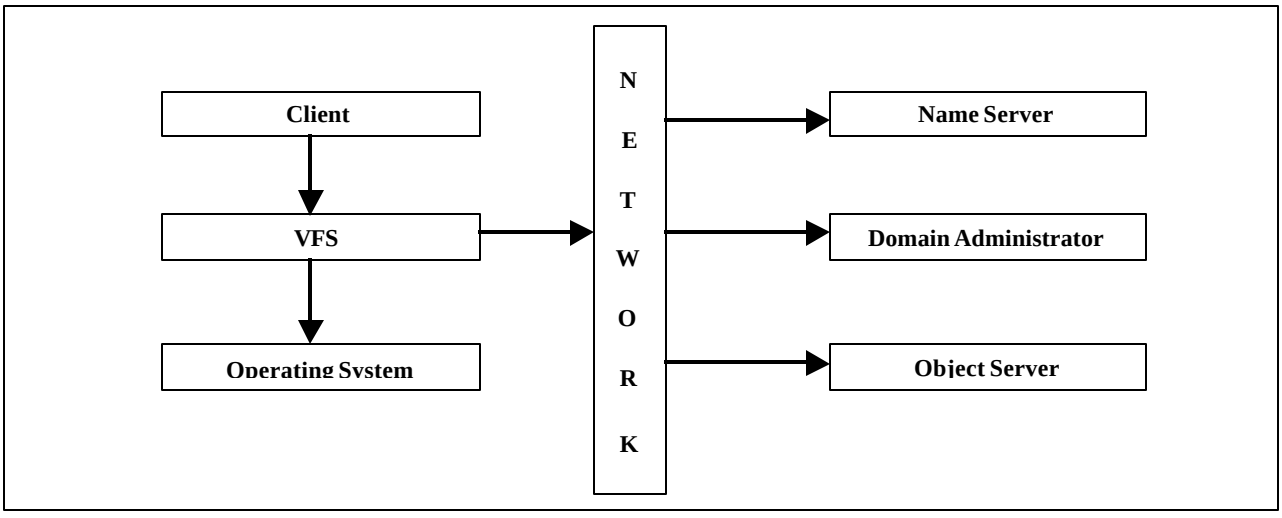

Figure 4: System architecture for distributed objects

4. PROTOTYPE IMPLEMENTATION

ADIOS is implemented completely in Java, and so assumes that all the objects in the system are accessible from and enclosed within a Java Virtual Machine. Other implementations are also possible for the framework, and we include a brief discussion about such an implementation in Section 4.2.

Java RMI (Remote Method Invocation) provides extensive support in building the framework that we have described. It provides a platform to independently implement, test and deploy server-side components and client-side components. It also possesses features critical to the framework, including migration of objects over the network and code mobility [12]. The ability to download code at runtime makes the framework more powerful, since the behavior of the system can be modified *after* deployment, without having to shutdown any part of the system.

There were a few Java-specific issues in the implementation of the framework. `PersistentObject` is modeled as a pure interface, with `AbstractPersistentObject` as an implementation (partially concrete), to increase the flexibility of the framework. Hence, application developers have a choice between a more generic and abstract `PersistentObject` (but with less support from the system); and a less generic but more feature-rich `AbstractPersistentObject` (that enforces strong concurrency constraints using thread synchronization, and stores state information for server callbacks to enforce strict AFS-like update semantics).

Since the `PersistentObject` is accessed using Java RMI, it has to extend the `java.rmi.Remote` interface. The `AbstractPersistentObject` extends `UnicastRemoteObject` (in the `java.rmi.server` package) so that its instance can be "exported" as an RMI object.

It should also be noted that any object that has to be transmitted over the network is required to implement the `java.io.Serializable` interface.

4.1 Strengths and Problems with the Java-based approach

The Java implementation has the advantage that all the system and application objects reside safely within secure Java Virtual Machines. Every operation within the system is subject to Java's strong type safety and uniform access across heterogeneous environments. However, Java is not the standard choice for system programming, and it may be difficult to directly port the system into the kernel. Hence, we investigated the feasibility of other approaches, which we briefly describe next.

4.2 CORBA/C++/Java based implementation

We implemented a partially-CORBA-based realization of the framework to test the feasibility of our thesis on non-Java platforms. The distributed storage components were fully implemented in C++, and the persistence framework was implemented in Java. The two parts of the system were bridged together using CORBA. This system was found to perform as well as the pure Java counterpart from the point of view of the extensibility and adaptation goals.

4.3 Testing

The objectives behind ADIOS were to come up with a design for an extensible distributed file system that can adapt to externally imposed demands. Hence, the prototype was tested on these two counts – extensibility and adaptability.

Extensibility is too abstract to test precisely. All software is extensible in one way or another, but it is the extension *model* that determines the ease, transparency and efficiency with which an extension can be applied [17]. A system can be shown to be extensible if it is general enough, and can be specialized for any particular domain. ADIOS imposes hardly any restriction on application-developers from the Application Programming Interface (API) perspective. The API is similar to the general-purpose UNIX file system. ADIOS was tested with applications that were specific to a particular domain, like multimedia applications, and was found to be specializable. The customization of the framework

will also be investigated for large-scale visualization and tele-immersive applications like those written for the CAVE [19].

Adaptation in a file system, as mentioned earlier, can be categorized into network-adaptation, device-adaptation and user-defined adaptation. ADIOS has not yet been tested for device adaptation, but the other forms are explored below.

Network adaptation requires the system to provide appropriate service with changing network parameters. To test this, a `Picture` class that contained multiple versions of an image with different fidelities (resolutions) was used. The system retrieved the lowest-fidelity version when the network became congested. On changing the round-trip time (RTT) to the node containing the `Picture` object, versions with varying fidelity were retrieved. The modeling of network parameters was initially simulated, and later tested on real networks with dynamically changing characteristics, with similar results.

User-defined adaptation was tested with two scenarios, one involving security and the other involving concurrent updates. The previously referred-to `Picture` object also had an encrypted version for secure transfers (that could also have been generated on demand). When the system was provided with a list of hosts marked as untrusted, it retrieved the secure version when the data was expected to pass through the untrusted hosts at runtime (currently it uses a `traceroute` implementation that will be replaced with other network monitors in the future). Concurrent-update semantics was tested using a "chat" application (a similar real-world application for such semantics is a stock-market environment). A `TextObject` was created on a remote server and was accessed simultaneously by multiple users. The situation of users exchanging messages was simply modeled as concurrent writes to a common object. It was found to be very simple to provide such functionality using the framework. The default system provides Andrew semantics for concurrent accesses.

5. CONCLUSIONS AND FUTURE WORK

The ADIOS proof-of-concept implementation demonstrates that it is possible to provide general-purpose adaptation in a file system, which can be customized and specialized for specific application needs.

Object-oriented frameworks allow us to partially define system behavior without enforcing unreasonable constraints on application developers. Frameworks help achieve adaptation and customization simultaneously using "inversion of control". That is, when events occur the framework's dispatcher reacts by invoking hook methods on pre-registered handlers. For instance, when any user modifies the state of a shared object, the `ObjectChangedListener`'s handler method performs application-specific processing, in this case, implementing application-specific concurrency semantics. Inversion of control allows the framework to uniformly apply a well-defined sequence of operations in response to external events, while still allowing an application to customize the behavior of those operations.

We still have to develop the implementation of the `SystemState` class such that it can respond better to applications querying it about available system resources. We plan to incorporate a network-monitoring tool to discover the network characteristics more accurately at runtime. We are investigating a plan where a network-scheduling algorithm can be integrated into the system for desirable characteristics like higher global throughput and QoS provisioning. Applications can negotiate with the `SystemState` class to improve performance and/or quality.

By keeping the definition of the `Constraints` class as general as possible, the framework gives application-developers complete flexibility in defining their own notions of adaptation. The goal of adapting to any system- or user-imposed constraint can be modeled by defining the `Constraints` class suitably.

Thus, the system is general enough to accommodate a wide class of applications, and can be made specific enough to enable individual applications to specialize the response of the system according to their individual requirements. We thus believe that our proof-of-concept ADIOS can be used to build a more robust and complete framework that can eventually be implemented as a standard file system on any operating system.

REFERENCES

1 Sun Microsystems Inc., "NFS: Network File System Protocol Specification," RFC 1094, http://www.faqs.org/rfcs/rfc1094.html

2 J. H. Howard, "An Overview of the Andrew File System," in *Proceedings of the USENIX Winter Technical Conference*, 1988.

3 M. Satyanarayanan, "Coda: A Highly Available File System for a Distributed Workstation Environment," *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, September 1989.

4 Brian D. Noble, M. Satyanarayanan et al, "Agile Application-Aware Adaptation for Mobility," *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

5 Brian D. Noble, "System Support for Mobile, Adaptive Applications," *IEEE Personal Communications*, Vol. 7, No. 1, February 2000.

6 Brian D. Noble, M. Satyanarayanan et al, "Application-Aware Adaptation for Mobile Computing," *Proceedings of the 6th ACM SIGOPS European Workshop*, Sep. 1994.

7 Roy Campbell, Nayeem Islam et al, "Designing and Implementing Choices: an Object-Oriented System in C++," *Communications of the ACM*, September 1993.

8 Yasuhiko Yokote, "The Apertos Reflective Operating System: The Concept and Its Implementation," OOPSLA'92 *Proceedings of the ACM*, pp.414-434, October 1992.

9 Ralph Johnson and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, SIGS, 1, 5 (June/July. 1988), 22-35.

10 L. Peter Deutsch, "Design, reuse and frameworks in the Smalltalk-80 system," in *Software Reusability, Volume II: Applications and Experience*, pp. 57-71, Addison-Wesley, Reading, MA, 1989.

11 P. Mockapetris, "Domain Names - Implementation and Specification", RFC 1035, http://www.ietf.org/rfc/rfc1035.txt

12 Sun Microsystems Inc., *Java Remote Method Invocation Specification*, 1.1 edition, November 1996.

13 A. Fox, S. Gribble, Y. Chawathe, and E. Brewer, "Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Perspectives," *IEEE Personal Communications*, August 1998.

14 D. R. Engler, M. F. Kaashoek et al, "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

15 David K. Gifford, Pierre Jouvelot et al, "Semantic File Systems," *13th ACM Symposium on Operating Systems Principles*, October 1991.

16 Fabio Kon, Roy H. Campbell et al, "2K: A Distributed Operating System for Dynamic Heterogeneous Environments," *9th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, August 1-4, 2000.

17 Brian Bershad, Stefan Savage et al, "Extensibility, Safety and Performance in the SPIN Operating System," in *Proceedings of the 15th ACM Symposium on Operating System* Principles, Copper Mountain, CO. pp. 267--284.

18 Erich Gamma, Richard Helm,Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, October 1994.

19 C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, et al, "The CAVE: Audio Visual Experience Automatic Virtual Environment," *Communications of the ACM*, Vol. 35, No. 6, June 1992, pp. 65-72.

20 Grady Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, Object Technology Series, October 1998.