THE DESIGN AND IMPLEMENTATION OF AN INTELLIGENT AGENT-BASED FILE SYSTEM

S. Andrew Hopper, B. S.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2000

APPROVED:

Armin Mikler, Major Professor
Paul Tarau, Committee Member
Steve Tate, Graduate Coordinator
Roy T. Jacob, Chair of the Department of Computer Science
C. Neal Tate, Dean of the Robert B. Toulouse School of Graduate Studies

Hopper, S. Andrew, The Design and Implementation of an Intelligent Agent-Based File System. Master of Science (Computer Science), May 2000, 79 pp., 1 table, 20 figures, references, 34 titles.

As bandwidth constraints on LAN/WAN environments decrease, the demand for distributed services will continue to increase. In particular, the proliferation of user-level applications requiring high-capacity distributed file storage systems will demand that such services be universally available. At the same time, the advent of high-speed networks have made the deployment of application and communication solutions based upon an Intelligent Mobile Agent (IMA) framework practical.

Agents have proven to present an ideal development paradigm for the creation of autonomous large-scale distributed systems, and an agent-based communication scheme would facilitate the creation of independently administered distributed file services. This thesis thus outlines an architecture for such a distributed file system based upon an IMA communication framework.

© Copyright 2000
by
S. Andrew Hopper, B.S.

ACKNOWLEDGEMENTS

I would like to thank Armin, for sharing my accomplishments and focusing my exuberance. Thank you so much Amanda, for enduring my late-nights, and for our baby, for giving me the added incentive to reach my goals. Most of all, I would like to thank God, without whom I could have done nothing.

CONTENTS

ACKNOWLEDGEMENTS iii

1 Introduction 1

1.1 Distributed File Systems 2

1.2 Intelligent Mobile Agents 3

1.3 IAFS Overview 5

2 Vision: Optimizations Through Agents 7

2.1 Transient Contribution 8

2.2 QoS Customizations 11

2.3 File System Location 12

2.4 Dynamically Reconfigurable Behavior 13

3 IAFS Architecture 16

3.1 Platform and Interface Requirements 16

3.2 Quality of Service Requirements 17

3.3 Architectural Overview 18

3.4 Architecture of a File System Contributor 21

3.5 Architecture of a File System User 24

3.6 Architecture of a File System Administrator 27

3.7 File System Architecture 29

3.8 Fragment Transferral 31

4 Interfaces 36

4.1 File Fragment Transfer Protocol 36

4.1.1 Functional Description 36

4.1.2 Protocol Definition 37

4.2 File Operation Transaction Protocol 41

4.2.1 Functional Description of FOTP 41

4.2.2 Protocol Definition 44

4.3 Distributed Agent Delivery System 48
4.3.1 ADP Protocol Specification 49
4.3.2 DADS Design 53
4.3.3 Security 60

5 Conclusion and Future Work 62
5.1 Future Work 63

A FOTP Protocol Specification 65

B ADP Perl Module 70

BIBLIOGRAPHY 77

LIST OF TABLES

4.1 FFTP Transaction Types 39

LIST OF FIGURES

2.1 NFS One-To-Many Relationship 8
2.2 Distributed File System Many-To-Many Relationship 9
2.3 IAFS Administrative Topology 14

3.1 IAFS Network Topology 22
3.2 FSC Architecture 23
3.3 FSU Architecture 26
3.4 Admin Architecture 28
3.5 Sample IAFS Inode Structure 30
3.6 Example Fragment Retrieval Pattern 33
3.7 Improved Fragment Retrieval Pattern 34

4.1 FFTP Retrieve Request Message Sequence Chart (MSC) 38
4.2 FFTP Retrieve Request Message 40
4.3 FFTP Store Request Message 40
4.4 FOTP Message Serialization 42
4.5 Agent Transfer from Node 2 to Node 1 49
4.6 Connection-oriented ADP MSC 51
4.7 Connection-oriented ADP Packet 51
4.8 Connectionless ADP MSC 53
4.9 Connectionless ADP Packet 54
4.10 ADP/Patron Protocol Stack 55

CHAPTER 1

Introduction

Among the most significant changes that have affected the domain of computer networking is the proliferation of high performance processing units, combined with ever increasing bandwidth capacity both at the Local Area Network (LAN) level, as well as at the very backbone of the world-wide infrastructure that services the Internet. As a consequence of the introduction of gigabit ATM and Ethernet-based infrastructure solutions, high-bandwidth distributed computing services[10] have begun to proliferate at an increasing rate[29].

Simultaneously, data storage devices have become capable of warehousing ever-increasing quantities of data. Although it might have once been true that storage sufficient for large data warehouses could only be found at research and industrial centers, this is no longer true. In fact, the Internet has proven that high-capacity storage technology has indeed reached the common populace. For this reason, it is a natural expectation that services will be developed that utilize the vast distributed storage system that is the Internet. Distributed file systems, in particular, can allow users to transparently access large amounts of data without having to solely contribute storage to such repositories.

However, the unpredictable nature of the Internet can make the administration of such a file system problematic. Current distributed solutions present the framework for the distribution of data, but do not attack the enormous task of administering a large number of nodes. Indeed, such administration requires a significant contribution by a human administrator. This paper describes the architecture for a distributed agent-based file system that focuses upon autonomous administration.

The Intelligent Agent-Based File System (IAFS) is unique because it uses Intelligent Mobile Agents as an abstract layer of communication between all system components. Such abstractions allow for unique optimizations to be made in the behavior of the file system, as well as in the creation of a decentralized administrative agent network. IAFS will not only solve the problem of centralized administration,

but facilitate customizations in administrative behavior through the use of Intelligent Agents.

This introduction will present the concepts that are central to the intent and design of IAFS. First, an overview of distributed file systems will be presented with specific examples of comparable implementations. Intelligent Mobile Agents as an architecture for developing distributed systems and as a vision of an entirely new network paradigm will also be introduced. Finally, specific information regarding IAFS and a general overview of the project will be provided.

1.1 Distributed File Systems

The development of high-bandwidth LANs and the need to share data across an organization encouraged the development of numerous *Network File Systems* over the course of the last two decades. Network file systems such as NFS[24] have fulfilled the need of users to utilize file storage resources over a network, and eventually presented a efficient solution for high-speed LAN environments, which are typically centrally administered. Because of this assumption upon centralized storage, and thus as a consequence of its design, NFS does not scale well to distributed environments in which storage resources need to be combined into a single shared resource.

By contrast, *Distributed File Systems* allow multiple "servers" to combine to form a single shared resource, whereas network file systems typically only map one resource to a single server. Distributed file systems typically support an inherent degree of fault tolerance: the existence of multiple storage servers allows data to be "striped" or replicated across multiple resources. Certainly, the concept of seamlessly distributing data and presenting a single, unified drive to the user implies that numerous coherency and synchronization issues need to be addressed. Distributed file systems have also typically assume some amount of physical administration to ensure the integrity of redundancy aspects of the file system. For the purposes of comparison, the most notable distinct implementations are the Andrew File System (AFS)[21], xFS[1], and Coda[26]. Although there are numerous other specific implementations, these are perhaps (at present) the most contemporary, and thus seem to integrate the most

novel approaches that have characterized prior examples.

AFS was among the first distributed file systems to present an efficient and stable implementation. Andrew allows users to mount shared *volumes* and provides a level fault tolerance and data redundancy. However, AFS assumes a relatively high level of administration. For instance, if a new server is added to the file system, the task of initializing this resource is a manual one. Coda, a descendant of AFS, improved upon the concept by providing high-availability and thus is more resilient to network and server failures.

AFS and Coda provide some degree of data migration services, or the movement of data to the area of networks in which it is needed. The central design assumption of these systems are for a two-tier hierarchy: the users of the file system, and the servers that combine to provide shared resources. xFS, which is a completely distinct implementation from AFS and Coda, addressed this by adding asynchronous data migration to service the locality in which the data is actually referenced. Although xFS is perhaps the most ambitious implementation in terms of supporting WANS as well as simply LAN environments[32], the administration of the file system still presents a burden upon networks which are truly shared.

The central goal of IAFS is to produce an autonomous file system which *completely* isolates users from aspect of system maintenance. Ideally, only the first and last contributor to an IAFS network need to concern themselves with the state of the file system as a whole. This paper describes the architecture of IAFS, and the way in which IMAs can allow the system to function truly autonomously.

1.2 Intelligent Mobile Agents

In short, Intelligent Mobile Agents *move* processes which operate on data instead of moving data to the processes which operate on them. Thus, agents are "mobile processes" which are composed of a section of code, transferable data, and a program state. These mobile processes, once initiated, have the capability of moving autonomously throughout a network. This approach, though dependent upon conventional client-server methodologies, represents an abstraction which can be useful

in the development of data-rich distributed systems[31].

Although these aspects of agents can be roughly duplicated using conventional client-server methodologies, perhaps the most significant gain in using agents is in the design abstraction that agents can represent. Agents (as a programming abstraction) allow behavior to be separated from the implementation and thus present a radically different pattern for the development of distributed applications. Hence, the use of mobile agents in creating the infrastructure for a distributed system could present a drastically different paradigm than that exhibited by conventional systems. Indeed, mobile agents can play a crucial role in developing efficient and varied synchronous and asynchronous services[27].

In order to illustrate the advantages that agents can present, it is first necessary to examine contemporary network design paradigms. Currently at least two widely used network application methodologies prevail in the field of network system design: *Client/Server*, and *Peer-to-Peer*. The Client/Server paradigm maps well to an environment in which nodes fall into the roles of *data/service owner* and *data/service user*, whereas the Peer-to-Peer methodology applies to networks composed of nodes which are both owners and users of data and services. These approaches suggest the roles that nodes play within a distributed system.

Not only do these methodologies define the roles of nodes, but also affect the design of those nodes. For instance, nodes that are "servers" (in the case of Client/Server) will be designed such that the main loop waits for connections and spins off threads of control to service requests[15]. Likewise Peer-to-Peer clients will likely implement a link resumption scheme to resume lost connections. Most importantly, however, these methodologies encourage developers to couple the format and location of data with the processing of that data.

For instance a web server would likely be tightly coupled with the fact that the data is being transferred via TCP/IP. Such a server would naturally associate the source address that originated a particular connection with the destination machine that initially requested the data. One might rightly suggest that a web server should not be concerned with the location of a host that requests data, nor with the format in which the host initiates the request. This not only serves to tightly couple the

web client with the web server, but implies that if data should be transferred over a different underlying communication channel, HTTP and TCP/IP must rest atop the new protocol stack. Even if the web server and clients are re-written to support a new environment, they would need to be re-installed on every node throughout a network.

IMAs, however, offer a solution to this coupling of systems. Networks which use IMAs to communicate treat data as a functional entity (an IMA). Instead of burdening nodes with the responsibility of knowing how to communicate data to/from other entities, the agent accepts this task. In effect, the node asks the agent to retrieve the data, in any way it can[5]. The node only needs to understand how to communicate with the agent. This not only allows developers to de-couple the communication between nodes within a network, but encourages them to de-couple the format and method that data is communicated from the processing of data. If the channels over which data is communicated should ever change, systems that utilize IMAs should only need to introduce a new agent into the network [22].

Because of these reasons, IMAs have begun to radically alter the way that we view the exchange of information across networks[3]. Many view the use of agents as an entirely new way to communicate data. One might say, though, that IMAs merely present a new paradigm which is functionally equivalent to Client/Server and Peer-to-Peer but unique that agents forces the separation between the communication of data and the way that such data is communicated.

1.3 IAFS Overview

Primarily, IAFS was initially intended as a case study in the development of agent-based distributed systems. Naturally, our criteria in the selection of a domain included the observation of the wide-spread use of conventional distributed applications: the choice of a domain which has already been at least partially researched would ultimately facilitate performance comparisons with existent solutions. In addition, because our goals included developing a large-scale distributed system which could be useful not only in scientific simulation circles, but ultimately in private and business

applications such as the Web Operating System[14], it was felt that a distributed file system would best fulfill these goals.

Distributed file systems also present design challenges which we hope to solve through the use of agents. The potential high latency and throughput and a low tolerance for delay makes this domain an excellent test-bed for agents in high performance computing.

This document will first address the optimizations that IAFS can achieve through the use of agents, such as *Distributed Administration*, *Transient Contribution*, and *Dynamically Reconfigurable Behavior* (Chapter 2). The requirements for an autonomous distributed file services system will then be outlined, followed by the specification of the architecture of IAFS (Chapter 3). Finally, the interfaces that connect the architecture, including the specification for the Distributed Agent Delivery System will be described (Chapter 4), followed by current state of IAFS development and the future work and research that is presented in Chapter 5.

CHAPTER 2

Vision: Optimizations Through Agents

In the development of local disk-based file systems, administrative tasks have typically been centralized at the level of operating system. Since these types of file systems are inherently local, this approach is an understandable design limitation: typically, such services will be administered locally, and the distribution of the administration of a file system which resides on a *centralized* machine would be naturally problematic. Of course, the same is true of network file systems such as NFS: although there is now a one-to-many relationship between the contributor of storage and the machines accessing the resources, the administration of the contributor is reduced to the same problem as in the local file system scenario (see Figure 2.1).

Unfortunately, the administration of distributed file systems has been approached in much the same way, although the distribution of the storage resources into a many-to-many topology can greatly complicate the task of administration (see Figure 2.2). Instead of an administrator having only to concentrate on the state of one storage contributor and perhaps maintaining the consistency of the data shared among the various users, the state of numerous contributors must be considered as well as the optimal distribution of data among them.

Most, if not all, conventional distributed file systems depend on some form of semi-centralized administration. Although vital data may, in fact, be distributed in such systems, administration is centralized in that administrative duties are privileged and centered around a designated [human] administrator. Although many asynchronous administrative tasks are performed (such as data caching/migration for the purposes of optimizing read accesses), the expectation that an administrator will be available to administer the storage contribution resources relegates such implementations to use only on well-maintained internal networks environments (corporate, academic, etc.).

IAFS, however, relegates almost all such administrative duties to the agents, and IAFS can in most (if not all cases) operate with no human administrator intervention.

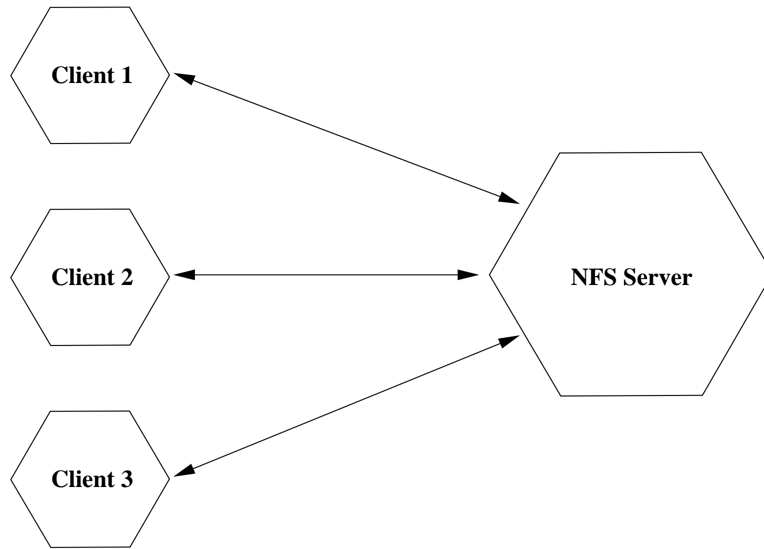

Figure 2.1: NFS One-To-Many Relationship

If for instance, a contributor ceases its contribution suddenly, IAFS should have the ability to “make the best” of the situation, and perhaps even plan for such occurances.

In the environment of the Internet (for which IAFS was not solely designed, but which IAFS seeks to accommodate from the onset), the usage patterns and dynamic contribution of resources will be unpredictable and changing. Data redundancy and migration, whether for the purposes of fault-tolerance or predictable removal of resources has been one of the design requirements from the conception of IAFS. In such an environment, for instance, the support of transient contributors and the automatic migration of critical data off of soon-to-be removed resources is vital. Such administrative duties are, in fact, relegated to administrative agents which reside throughout the file system.

2.1 Transient Contribution

One example of an administrative duty which is typically delegated to human administrators is dynamic storage contribution. Conventional distributed file systems

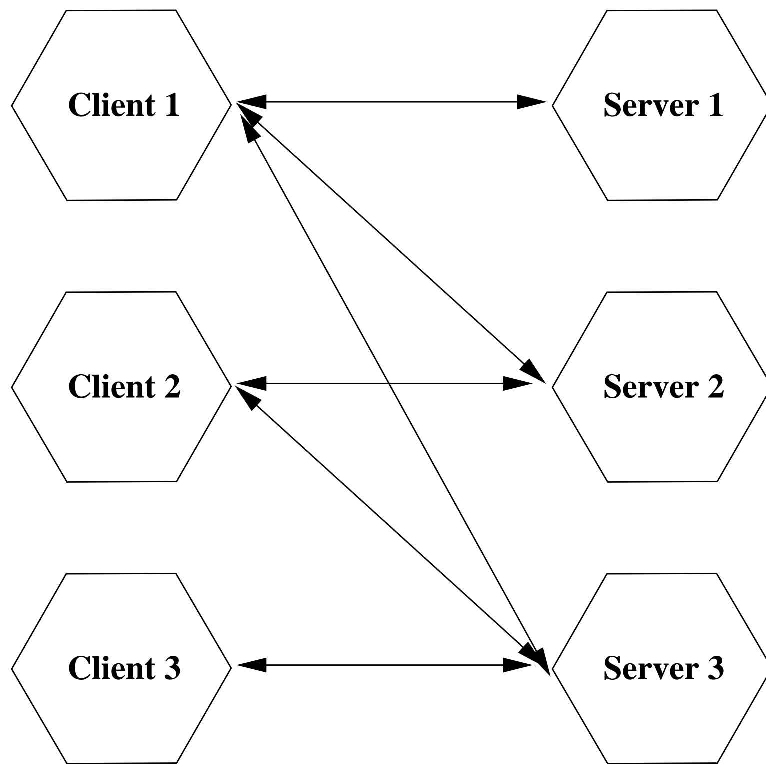

Figure 2.2: Distributed File System Many-To-Many Relationship

require users to manually move data from one resource to another in the case of server replacement, etc. IAFS, however, inherently supports transient contribution of resources: users can contribute data for a pre-specified time only, after which the contribution may be revoked. IAFS thus treats the assumption on indefinite contribution made by of conventional file systems as merely a special case of this generic solution.

Thus, the functional component of IAFS known as the *administrator*, must provide a facility for the migration of data off of DCs (or *Darwinian Contributors*), which are those contributions which are soon to be removed from the storage pool. This facility should be generic in that it should present an uniform interface to the task of data replication for fault tolerance parameters and to the migration of data for the optimization of read accesses.

At some reasonable point before the time that a particular contribution will cease to be available, administrative agents will dispatch *Migration Agents* (MA) which will begin to gradually relocate data to other storage facilities. These agents will work on behalf of the administrator, but will *own* the task of the transferral of data.

The MA will first attempt to determine the state of the contributor, by querying the administrator as to the amount of time until the end of the contribution period, the amount of the contribution, and historical data regarding this contribution (reliability, etc.). The MA will then obtain a list of contributors, noting the their reliability, storage capacity and relative position (in terms of capacity, throughput) to the DC. At this point, the MA has sufficient information to complete its task. The data is migrated first to the closest nodes, and if necessary to more distant nodes if space constraints warrant the dispersal of information across multiple machines. Following the migration of data, the MA returns to an administrator and conveys the successful completion of its task.

Interestingly, this type of administration fits the both the needs of an Internet *file warehouse* as well as those for local area networks[14]. For instance, a large file warehouse could be instantiated on the Internet with one contributor. Gradually, other contributions will be added to the warehouse, while others are removed. Over time, the original set of contributors could be completely replaced without data loss or

the need for any (human) administration. Although such an administration scheme is most useful for domains where administrative overhead *must* be minimized, centrally administrated LAN environments can also benefit from such a scheme. For instance, additional storage servers could be added with minimal setup overhead.

2.2 QoS Customizations

From the administrator's perspective the "call" to a migration agent to service a Darwinian Contributor would represent a request for an asynchronous service. In such an environment, the paradigm of a client and server has thus been radically altered. The architectural advantage in separating the interactional behavior from the implementation of the system as a whole facilitates the creation of variant behavior patterns, perhaps depending on dynamically changing user quality of service requirements.

This behavioral aspect of agents within the system represents the very heart of the design advantage that agents can represent. Agents truly represent a departure from conventional client-server architecture in that no participating network element truly plays the role of client or server. Agents represent *mobile threads* which can be synchronously or asynchronously dispatched on behalf of the user[30].

Because the agents' behavior is encapsulated, agents with variant behavioral patterns can co-exist within the same file system. For instance, suppose that an administrative agent is able to ascertain that certain contributors, whether because of their previously transient nature or even because of their location (that is, the network separating the administrator from the contributor) are inherently unreliable. The administrator can thus dispatch different, perhaps more aggressive agents to recover data from such contributors. In contrast, contributors which are historically stable can be relocated much more conservatively.

The administration of an IAFS file system is not solely facilitated by the Administrator agent. The alteration of all behavioral aspects of component interactions can be facilitated through the modularization of these behavioral characteristics through the use of agents.

The fault tolerance requirements would necessitate the duplication of file fragments, and the intelligent "striping" of those fragments across the file system. This service, in fact, can be consolidated within an *Dispersal Agent* (DA) specifically designed for such a task. A DA would work on behalf of a the user, replicating fragments according to the users configuration parameters. Again, since this functionality is transparently encapsulated within the DA, its behavior can be effectively altered without affecting the system as a whole.

2.3 File System Location

The implementation of a distributed file system requires that a catalog be maintained containing at least the following information:

- List of users
- File system statistics (size, space left, etc.)
- Catalog of contributors *reliability* index, including historical data, etc.

Because this information will be frequently referenced by all network elements (and agents), such data is naturally important to the normal functioning of the file system as well is the maintenance of the its consistency. For this reason, it is vital that the information be distributed throughout the file system and available to all agents, whether fulfilling a read/write request or an asynchronous worker-agent working on behalf of the user.

Ideally, the administrator would be viewed as working *above* the file system as a whole, fulfilling an omnipotent rule in the course of the system. In practice, however, the agents must exist somewhere and although in theory the administrator *could* exist anywhere, in practice its location should exist on one of the physical participants of the file system. Because the user is by definition relatively fleeting in nature (its presence is, in some sense passive and not essential to the normal operation of the file system), the Administrative component resides on every contributor, constantly reconciling its information with other administrators.

The role of the administrator is very loosely coupled with the relative location of the contributors. In fact, the residence of the administrator on the contributor sites comes out of implementation convenience, rather than through investigation. Future experimentation may yield a somewhat high overhead in the reconciliation of data between the administrators, and an optimized solution may include the reduction in the number of administrators depending on collected network statistics or topological information. For this reason, the administrator role seems to most closely resemble that of an independent agent. Initially, of course the administrator will be stationary, but communication with the administrator will take place through mobile agents, allowing future design optimizations to be facilitated.

Regardless of the implementation of the administrator, however, it is, in many respects, the reference or “handle” to the entire file system. Because the contributor list changes, it is necessary that the file system as a whole continue to be accessible without maintaining a central registry that must be constantly updated.

For instance, a new contributor who wishes to contribute to a file system needs to find at least one other contributor of the file system. Although the new member knew at one time the IP of a fellow contributor, that IP might no longer refer to an existent node.

For this reason, each administrator (at its conception) could register itself in a *multicast group*, and de-registers itself at the time of contributor de-registration. This multicast group would have been initialized by the very first administrator to have been created at the time of file system conception. New contributors or “lost users” can regain contact with the file system by multicasting a message to every administrator within the file system (see Figure 2.3). Thus, even as the file system “migrates” a central reference to the file system can be maintained.

2.4 Dynamically Reconfigurable Behavior

Since the entirety the communication of the file system is accomplished through the use of agents, and because the relationship between the agents and the network elements has been formally (and generically) defined, the behavior of the agents has

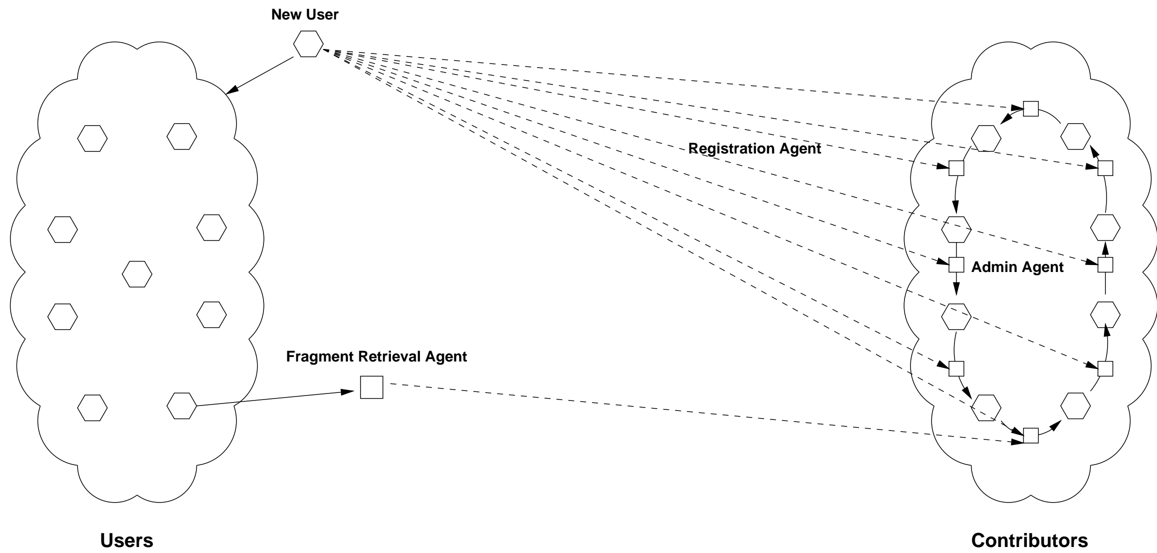

Figure 2.3: IAFS Administrative Topology

been effectively encapsulated: neither the contributor nor the users will have any idea how or through what means an agent accomplishes its task.

This behavior can be likened in principle to behavioral encapsulation techniques prevalent in Object Oriented Design methodologies. In this case, however, the "agent" can be compared to a mobile object[12]. This object has a predefined and standardized interface and thus presents an uniform face to the rest of the system. However, not only are the methods and data on which the object depends private to that object, but so is the *location* and the *implementation* of the object.

Because an agent is inherently mobile (a stationary agent being a special case of this general behavior), the *client* of the agent is isolated from not only to the agent's internal behavior but also to the locale at which its behavior can be observed. In this sense, the agent can actually be viewed as both the client and the server from the perspective of the system as a whole: to the requester, the agent represents a server, and to the service which ultimately services all or part of the request, the agent can be seen as a client. Thus, the agent truly behaves as a intelligent proxy, acting on behalf of those requesting its services.

The way that such a proxy meets the needs of its client, however, can vary from its internal behavior or even with regards to its language. Assuming all agents have a generic migration protocol, agents written in intepreted languages such as Java, Perl, or Python could inter-operate and *appear* to be the same type of agent. Such a choice could, indeed, depend on the availability of interpreters or runtime compilers on a particular network. If, for instance, a laboratory was composed entirely of Intel 80486 processors, perhaps the most efficient method would be to transfer executable, architecture dependent agents.

Most interestingly, however, is the concept that agents allow the behavior to be *shared* throughout the system. The same agent fulfilling the request of a particular component can, indeed, maintain a "laundry list" of various other components who also require the agents services. One *instance* of a mobile agent could theoretically serialize many common tasks which must be accomplished throughout the system into one such list and indeed fulfill each of them in turn. Although this approach might be inefficient, the key idea is that with the advent of intelligent mobile agents the communication between nodes can be abstracted to the point that no assumption can be made regarding the implementation of its behavior and that thus an agent can simultaneously meet multiple goals throughout a system.

This feature allows human administrators to manage the way the file system operates. For instance, a "proxy agent" could be developed which could relay data through a gateway or firewall. "Secure agents" could be developed which could force the authentication of all agents or even the encrypt/decrypt all transferred data (see Section 4.3.3 for a discussion of IAFS agent-based security).

Replacing agents represents only one way in redefining the behavior of the system. Suppose, for instance, that a user wishes to maintain only one contributor and in this case practicality would not necessarily warrant the use of agents (in terms of the overhead agents can introduce). In this scenario, mobile agents could be entirely removed from the system and replaced by a single, stationary agent which communicates directly with the contributor.

CHAPTER 3

IAFS Architecture

In order to create an agent-based distributed file system, the architecture of the system must first be created. In the case of IAFS, this includes the identification the network components and specification of interfaces between those components. In addition, the derived requirements for each network components should be itemized so that the overall system requirements will "trickle" down to the design phase.

The architectural design phase in the development of IAFS (or any large system) is perhaps the most crucial step towards an extensible design. Indeed, the development of a generic system architecture for IAFS is critical for the successful integration of IAFS into distributed systems of varied design.

In developing an agent-based file system that accommodates the system requirements that will be outlined in this chapter, certain constraints are implicitly imposed upon the architecture of the system. For instance, the need for platform independence mandates that as much of the behavior of the system should remain outside the operating system. Likewise, the requirement for distributed administration implies that components should be self managing. Most importantly, though, the usage of the file system dictates the placement and type of interfaces that should be used within the overall architecture.

3.1 Platform and Interface Requirements

As previously stated, the intention of the development of IAFS is to serve both as a case study in the use of agents, and to explore the optimizations that can be realized in developing an open distributed file system. The intended audience for IAFS is the Internet community, as it is foreseen that numerous distributed services can be built upon the IAFS architecture (not merely file manipulation services). Because the Internet is (and will continue to be) an extremely heterogeneous environment in

terms of computational platforms and network topologies, IAFS should be designed with this in mind.

IAFS should be designed and implemented as a portable architecture with open interfaces between network components. A platform independent design should allow IAFS to be easily "ported" to a myriad of computing architectures and open, well-documented interfaces should serve to allow multiple implementations and extensions to be integrated in to the IAFS architecture.

Toward this goal, IAFS should not primarily reside within the domain of the host Operating System, but should be designed as an external service that communicates with small operating system *drivers*. This not only simplifies IAFS, as less cumbersome *kernel code* need be developed, but forces the architecture toward an open interface design: if a generic protocol can be developed to communicate with the kernel, services/applications that do not require the abstraction that a file system offers can utilize an IAFS network directly. Most importantly, this implies that if IAFS honors conventionally supported interfaces such as NFS, OS-specific IAFS support could be as simple as "mounting" IAFS as a shared NFS drive.

Finally, the design IAFS interfaces should not depend upon specific services provided by TCP/IP. Ideally, IAFS should assume the availability generic services (such as reliable connection-oriented transport) and design the communication interface so that the implementation dealing with communication services can be easily replaced without affecting the entire system.

3.2 Quality of Service Requirements

IAFS should be designed such that it can be optimized both for use on LANs and WANS. This requirement implies a number of architectural constraints, all of which can be optimized through the use of agents. For instance, users within a LAN environment (such as within a corporate or university setting) are generally likely to value throughput over security, whereas WAN (Internet) users are likely to value security over latency. In order to accommodate these constraints, each variation could merely implement a different agent communication system: agents could be responsible for

ensuring the security of the data transfer (via other agents), and other agents could be designed to implement a lighter-weight security implementation, but support an extremely robust file transfer mechanism.

Aside from the overhead introduced by agent mobility, as an architecture IAFS should be robust and efficient. In order to support this requirement, the implementation of a Java-based architecture would not be an ideal solution at this time, as the cost of interpretation could actually present measurable performance degradation. IAFS should also be based upon an Object-Oriented component-based architecture, since this approach has repeatedly been proven to produce (in most cases) more reliable systems. Event-handling should be configurable to facilitate the data-capture regarding system faults, but provide still lower verbosity events for a normally-functioning system.

Finally, IAFS should be scalable in supporting literally *huge* file systems. Currently, locally administered file systems such as the BSD Unix File System (UFS)[18] or NFS are limited to file system sizes in the giga-byte range. IAFS should provide an addressable space to allow file systems that are literally global in scale to be created (in the terabytes). In implementing a scalable architecture, the design/implementation should not assume a maximum limit upon the number of agents that can be concurrently present within the system, and in fact should provide flexibility in supporting multiple agent architectures.

3.3 Architectural Overview

As mentioned, the usage of IAFS largely dictates the type and number of the network components that will compose the IAFS architecture. In order to distill this, the *usage* of IAFS must first be understood. Indeed, there are at least two ways to *use* IAFS.

First, individuals can use the system to contribute local storage space to the file system. This implies that a client must exist that can register this space and provide external, transparent access to the local file system. Because users can exist on multiple platforms, the communication to this client must be independent of the platform on which the client is running. Also, because the local file system is of an

unknown type and thus makes different assumptions on data access, sizes, etc., the facility for the physical storage of data must be re-configurable.

Second, individuals can use the IAFS as a storage medium for their data, either through as file system or by using an application that is *IAFS-aware*. These types of usages implies that a user client should consist of a small kernel module that allows IAFS to appear as a native file system. This module should utilize a well-defined interface to communicate with a user-space daemon that manages the retrieval and storage of files through the agent network.

These two use cases deal with the explicit relationship between the user and an IAFS network. An implicit use case for each are the administrative services that *must* exist. Though not directly related to the business of transferring files from point to point, there are a number of administrative duties that must be accomplished for IAFS to function properly. For instance, in order for a system to contribute storage space, it must be able to *locate* the file system. For a stationary distributed system, this would conceivably be a trivial task: maintain a one-to-one mapping between the identifier of the system and the network addresses of any one of the hosts that make up the system. However, an IAFS file system could conceivably migrate over time such that none of the original contributors remain valid. This and other tasks (such as dynamic configuration) constitute the duties of the *IAFS Administrator*.

The communication between the contributor and user should be abstract and network-independent, and thus mobile agents will be the application-level transport mechanism for all requests. There are at least two different types of traffic that must be transferred: file requests/data, and administrative data. For this reason, there will exist at least two different types of agents to accommodate these two different types of requests.

Finally, in order for the File System Contributor (FSC) and File System User (FSU) to communicate and receive requests to agents, there must exist generic, extensible interfaces that are common to both the agents and the FSC/FSU. Among the interfaces that can be identified at this stage are the file request/receive interface that will allows FSUs to request the retrieval or storage of a file (or part of a file) and which will allow agents to transfer file data to/from an FSC. Secondly, there must exist an

administrative protocol that will allow FSUs and FSCs to register their presence to the rest of the IAFS network and to communicate configuration or administrative data.

At this point, the components that will form the IAFS network have been determined (see Figure 3.1). Although further analysis due to changing requirements might create a need for additional infrastructure, the accommodation of these new requirements should be met by the the creation of new types agents. For instance, if and when IAFS supports file replication, or the storage of multiple copies of files for fault tolerance or performance reasons, the addition of a file synchronization agent to IAFS would allow the consistency management of the replications.

In the following sections, the high-level architecture of the FSU, FSC, and the agent/administrative network will be outlined. Because it would be beyond the scope of this document to proceed through the lengthy process of decomposing and discovering the high-level architecture of each individual network component, these details will not be included. Instead, the result of this analysis will be presented and explained. Although the data presented will begin to touch on design issues, the determination of architectural constraints will only serve as a guideline for the design and implementation of the system: only high-level components and functionality will be addressed in this architectural analysis.

3.4 Architecture of a File System Contributor

The role of the FSC can be simply understood as a daemon receiving transaction requests to store or retrieve file *fragments* ("pieces" of files) via IMAs, and servicing those requests by reading or writing data on the local file system. Figure 3.2 illustrates the overall architecture for the FSC. Note that this diagram does not attempt to follow any standard notation: lines with arrows indicate data flow in the implied direction, thick squares represent communication ports (channels), and three-dimensional objects are replicated (representing multiple instances or streams of execution). Also note that any task division (processes, threads, etc.) is merely implied as a matter of convenience and is left entirely as an implementation detail.

To facilitate the generic transfer of agents from one IAFS component to another, a protocol and implementation for the generic transferral of agents of multiple types (languages). In the Figure 3.2, this component is referred to as *DADS* (Distributed Agent Delivery System). Section 4.3 details this protocol and the framework that is reused in both the FSC and FSU[13]. Although there is a great deal to be said regarding the actual transferral and maintenance of agents within IAFS, the pertinent issue from the perspective of the IAFS network components is that agents can be delivered from host to host and can communicate to a FSC or FSU in an abstract manner. In addition, it can be assumed that there exists a generic method for requesting that a "fresh" agent be created, as illustrated by the *Agent Factory* in Figure 3.2.

In order to communicate with agents that arrive requesting the storage or retrieval of file fragments, there must exist a communication channel over which a well-known protocol is spoken between the agent and the FSC. In Figure 3.2, this is referred to as the File Fragment Transfer Protocol (FFTP) Port. FFTP is a Domain-Specific Protocol as is specified in Section 4.3.2.

FFTP requests are forwarded to components within the FSC, which in turn interfaces directly with the local file system (marked as "OS FS" in Figure 3.2). File fragments are either stored or retrieved from the local file system and requests are processed in parallel (as indicated in Figure 3.2 by the replicated FFTP port). Because users may wish to contribute space to multiple IAFS networks, the FSC must

logically separate storage space on the local file system.

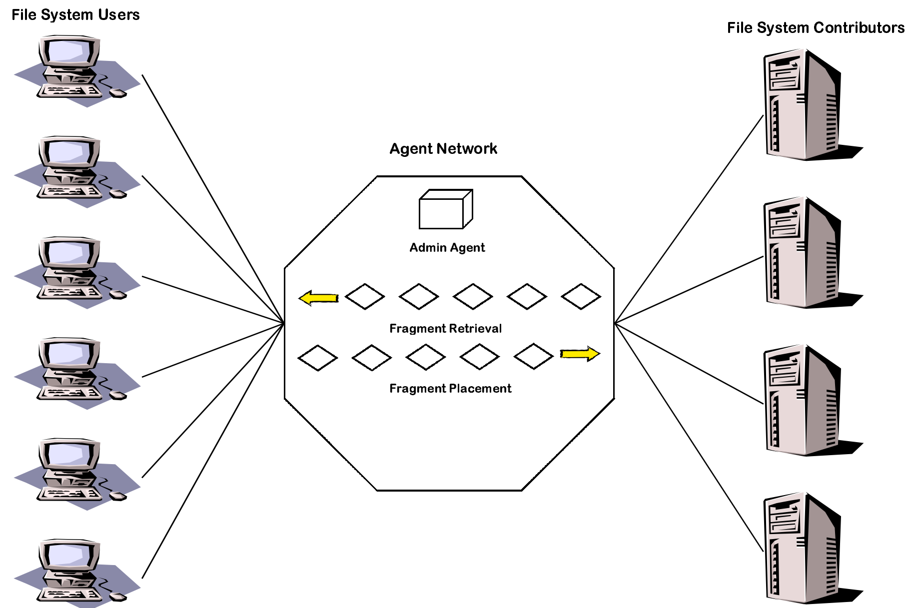

Figure 3.1: IAFS Network Topology

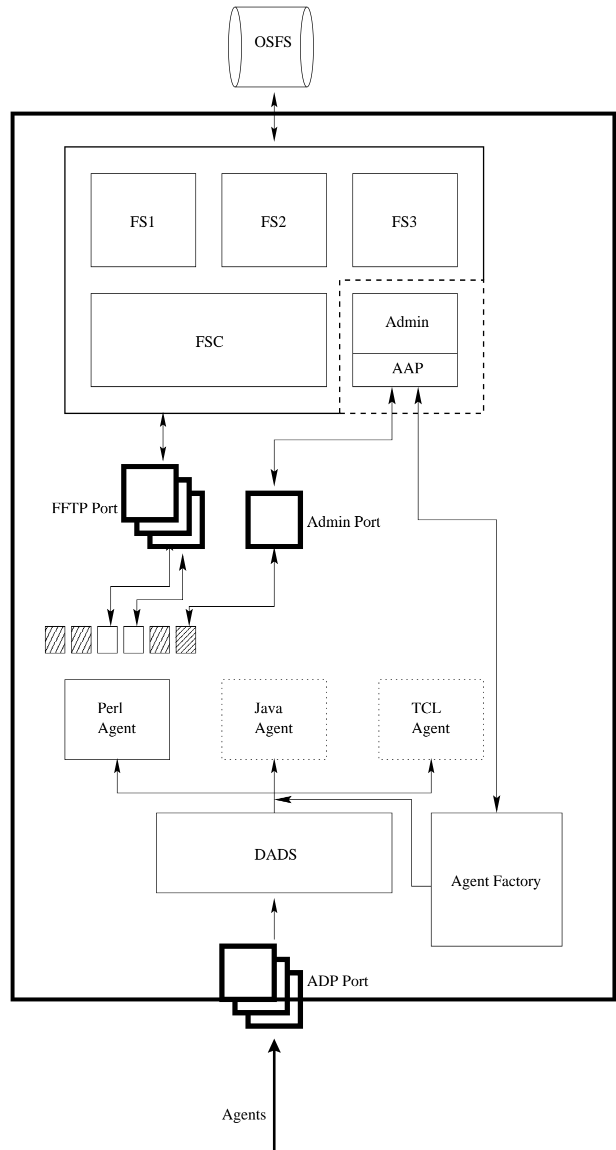

Figure 3.2: FSC Architecture

3.5 Architecture of a File System User

At first glance, the architecture of the FSU (see Figure 3.3) does not appear significantly more complex than that of the FSC, the complexity of the task that the FSU fulfills is much higher. Instead of being driven solely upon incoming agent transaction requests, the FSU is driven by both requests from the user and incoming agent transactions.

Beginning from the perspective of an incoming agent, the FSU architecture suggests the reuse of the DADS that is used in the FSC (see Section 4.3). Additionally, the FFTP protocol implementation is also a likely candidate to achieve some degree of reuse between the two tasks, although the FSU and FSC will only need to implement approximately half of the entire FFTP specification (the agent, however, will of course, need to communicate the entire protocol suite).

The *FFTP Router*, however, is more than merely a communication port implementing the FFTP protocol. Instead, its task is to *associate* incoming agent transactions with pending transaction registration requests. For instance, if the *File Cache* requested the retrieval of a particular file fragment, it would first register it transaction with the *FFTP Router* and then signal the *Agent Factory* to create a "fresh" agent. After agent initialization, the agent will connect to the *FFTP Router*, registering its transaction identifier so that it might be associated with the pending transaction that is awaiting the arrival of the new agent.

The *File Operation Translator* queues user high-level file transaction requests (read, write, delete, etc.), and processes these requests by dispatching agents to retrieve or store file fragments. In the case that agents become "lost" from the perspective of the FSU, this component will set a timeout on agent transactions, dispatching new agents as necessary. In addition, the *File Operation Translator* maintains caches of *Inodes* (see Section 3.7) and file fragments for each "mounted" file system.

Incoming user transactions are "serially multiplexed" and sent through the *FOTP Port* and delivered via the *File Operation Transaction Protocol* (FOTP) (see Section 4.2). The reason for implementing a serial delivery scheme, as opposed to supporting multiple concurrent FOTP "connections," is that the *FOTP Port* could conceivably

be a single character device (communicating directly with the kernel) instead of a TCP connection. The *FOTP Router* decodes and directs FOTP transactions to the appropriate file system context.

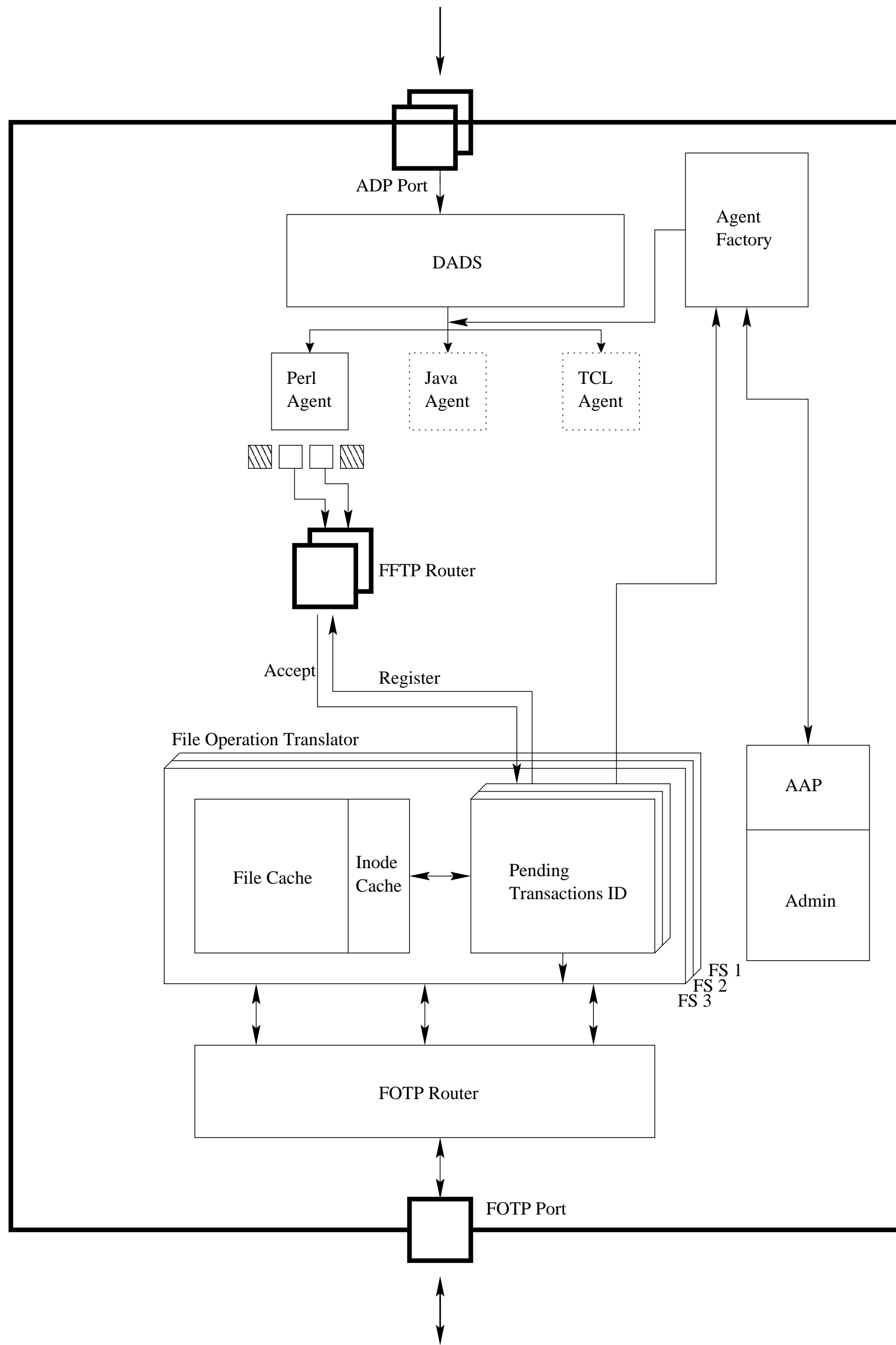

Figure 3.3: FSU Architecture

3.6 Architecture of a File System Administrator

The *Administrator* is unique with regards to the FSC and FSU in that its behavior is entirely encapsulated by IMAs. Indeed, the *Administrator* is an IMA, although it may in fact be stationary most of the time. That is to suggest that the agents that communicate with *Administrators* honor an administrative Domain-Specific Protocol (DSP) (for information regarding DSPs see Section 4.3.2), and their implementation effectively hides the existence of such a protocol. For this reason, the specification of the *Agent-Administrative Protocol* (AAP), and the detailed specification of the functionality of an *Administrator* will be left to the design phase, as this can be seen as an implementation detail.

Although the details of the *administrator's* duties have not been completely determined, the *Administrator's* overall architecture would regardless appear as that in Figure 3.4. It must be noted that this suggested architecture does not imply a single-agent design: the *Administrator* would likely be distributed and network management operations would likely be delegated to other types of agents that might be needed.

Agents transactions would be communicated via AAP to a *Request Handler* that would accept requests and query the resident data as needed and perhaps reconcile new data with that of other *Administrators*. Since IAFS is a distributed, fault-tolerant system, a number of *Administrators* will be present throughout the file system at any given time.

Because there are multiple *Administrators*, and these could migrate to another set of nodes entirely, the addressability of the *Administrators* becomes an interesting problem. One suggested solution is to register an IP Multicast Address to the entire set of *Administrators* at the time of creation of an IAFS file system[19]. As the network begins to migrate, the *Administrators* will re-register and thus dynamically alter the multicast group of which they are a part. In this way, the addressability of the entire IAFS network could be reduced to a single multicast IP address.

This section has outlined the high-level view of the role of the *Administrator* within the context of the architecture of IAFS. In actuality, the *Administrator* would be responsible numerous other duties relating to the administration of an IAFS file system

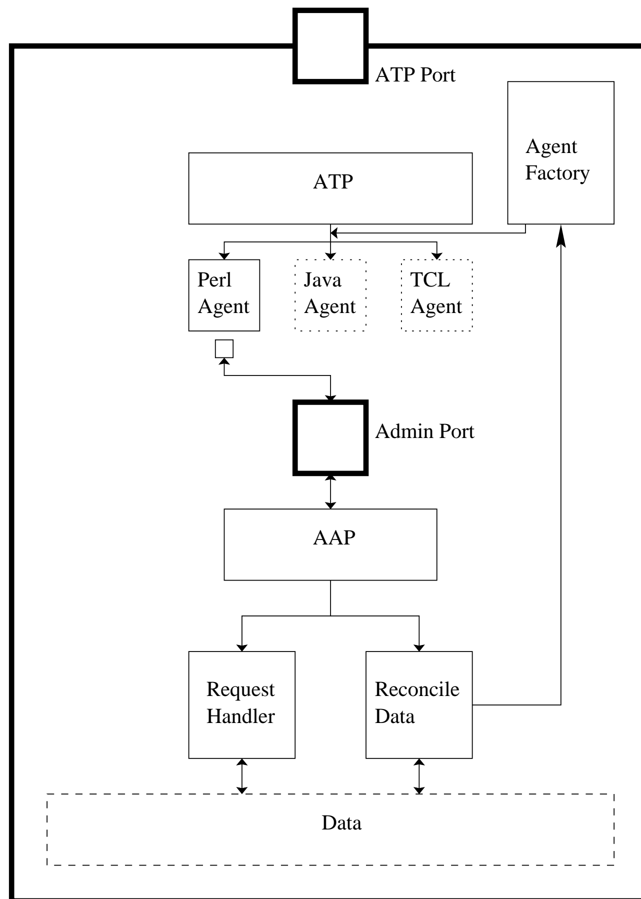

Figure 3.4: Admin Architecture

as a whole. Among these tasks would be *Configuration analysis/management*, *Fault management*, *Data redundancy* (fragment replication), *System creation/destruction* and *User/Contributor registration*.

3.7 File System Architecture

In order to develop the protocols (FFTP, FOTP, ADP) that will provide the public interfaces for IAFS, the overall architectural design of the physical file system must be available so that the data that these protocols will convey can be known. Because complete specification of the intricacies of the file system structure could be quite lengthy, and since this data would not fall under the domain of system architecture, a cursory view of the file structure will be presented. Further detail is left to the discretion of the implementor.

The storage structure of IAFS is roughly modeled after the BSD UFS File System [18]. Like the UFS, IAFS divides files into sections of uniform size which these pieces are referred to as *blocks* by UFS. IAFS, however, denotes these sections as *fragments*, as blocks tends to refer to physical storage (i.e., disk *blocks*). Also like UFS, IAFS tabulates fragments in structures called *Inodes* that provide direct fragment accessibility for small files, single indirection for files of moderate size, double and triple indirection for large files, and quadruple indirection for huge files (see Figure 3.5). This type of indirection is achieved by directly addressing a fragment that contains fragment references to other fragments, etc. IAFS does not globally specify a fixed fragment size, but it does place the constraint that within a particular file system, all fragments must be uniform in size, which greatly simplifies the problem of physical storage on FSCs.

Unlike UFS blocks, IAFS fragments can not be directly accessed. Because a single fragment may simultaneously exist in multiple locations throughout a single file system, Inode "fragment pointers" contain a list of FSC addresses. The size of this list is not fixed, hence representing an Inode as a simple word-aligned structure would be difficult. For this reason, and because it is necessary that the format of inode storage be as platform in-specific as possible, the Inode is ASN.1 encoded[7] (the exact implementation of the structure is viewed as implementation-specific and is thus not detailed in this document). An IAFS Inode consists of multiple fragments: the "root Inode fragment" contains all the persistent data for the file (such as modification time, size, number of fragments, etc.) plus the single, double, triple, and quadruple

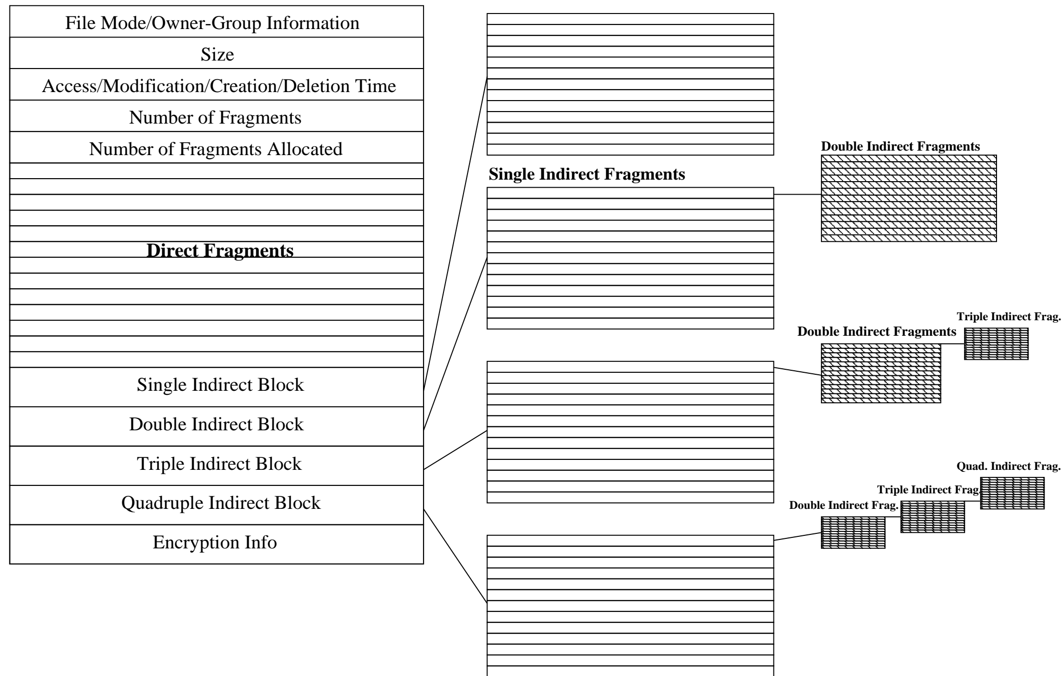

Figure 3.5: Sample IAFS Inode Structure

indirect fragment pointers. Each root fragment points to fragments that contain a list of fragments pointers.

Inodes, themselves, are identified by a 96-byte network-byte-order identifier. In theory the only requirement in forming this specifier is to ensure that every Inode has a unique identifier, although in practice this is difficult because identifiers are assigned in a distributed fashion. For this reason, this identifier should be formed from the unique network address (IP) of the machine that instantiated it, the time when this identifier was allocated, and a random integer. In all but the most coincidental circumstances, this will ensure the uniqueness of every Inode identifier, but in the case that two random integers are assigned within one time unit, the creator should keep a cache of all random numbers used within that time unit, and recalculate the identifier if necessary.

With this framework, the structure of the file system has been fully defined. Applications such as databases that need only to store data in a distributed format and associate that data with a key (Inode identifier) could utilize the file services adequately at this level. However convenience dictates that IAFS should also provide a mechanism for the organization of files into a hierarchical tree or graph structure (as UFS has). IAFS thus provides such a hierarchy by creating "directories" that merely map Inode identifiers to a list of file names. These identifiers may map to files or to other directories. Directories are simply files and so these mappings are contained in file data. Note that although IAFS, like UFS, does not distinguish between binary and text files, directories can be distinguished from other types of files by the examination of the *file flags* field in the Inode. IAFS also provides hooks for the support of hard links (*Links Count* in the Inode) and symbolic links can be achieved by mapping file names to other names within the directory file data.

3.8 Fragment Transferral

Up to this point, much has been said regarding components that compose IAFS and to the physical structure of the file system, but the method in which fragments are transferred across a network has not been outlined. Certainly, the details of the transferral can be "put off" to the low-level design of the agents, and this fact truly represents the sheer elegance in utilizing agents as a method of abstracting the communications behavior of a system. Nevertheless, the IAFS architecture would not be complete without not at least showing one possible pattern in which agents could collect and move fragments across the file system.

In the case of the fragment retrieval pattern depicted in Figure 3.6, a FSU would dispatch a single agent to serially retrieve nine fragments from four FSCs. In defense of simplicity, this approach is perhaps the most straightforward way in which agents could be used to recover fragments that are distributed across a network. This pattern does not attempt to take advantage of the parallelization of data retrieval that is possible with distributed file systems. Indeed, the FSU could actually dispatch four separate agents (one to each FSC).

The cost of this approach, though, should be weighed against the benefits. For instance, if too many agents are dispatched, the network overhead of transferring and executing the agents could outweigh the performance increase that could realized by parallel data transfer. Thus, a set of heuristics should eventually be developed (perhaps experimentally) in determining the proper number of agents that should be assigned to the task of retrieving a number of fragments. The work of Mikler and Abbas[20] has greatly improved our understanding of the problem of determining the optimal number of agents for the purposes of information retrieval. Indeed, ideally the agents can themselves dynamically determine the average cost that their presence incurs upon the network and weigh this cost against constraints that the network imparts to the task of information retrieval. In this case, the factors in determining these heuristics would be:

- Distance to FSCs
- Varying latencies to set of FSCs
- Cost of agent transfer vs. number of fragments to be retrieved

Finally, the approach outlined in Figure 3.6 assumes that fragments will *follow* the agents throughout the path of fragment retrieval: as the agent collects fragments from contributors, the size of the agent will grow accordingly. The rational for utilizing this would be simplicity, but in the interests of performance, this method would not be optimal. Indeed, because these fragments are requested by only one user (at least from the perspective of the retrieving agent), there would be no benefit in moving a possibly large number of fragments more than one hop. En lieu of this, then, one possible solution is to assign a stationary agent to accept fragment requests from the FSU, distribute new agents to handle the requests, and to wait for fragments to be transferred directly from the remote agents (see Figure 3.7).

The major concern in fragment retrieval is performance, given a predetermined set of nodes that contain a set of fragments. IAFS will eventually support *fragment replication* (see Section 2.2), and the issue in the case that multiple references of

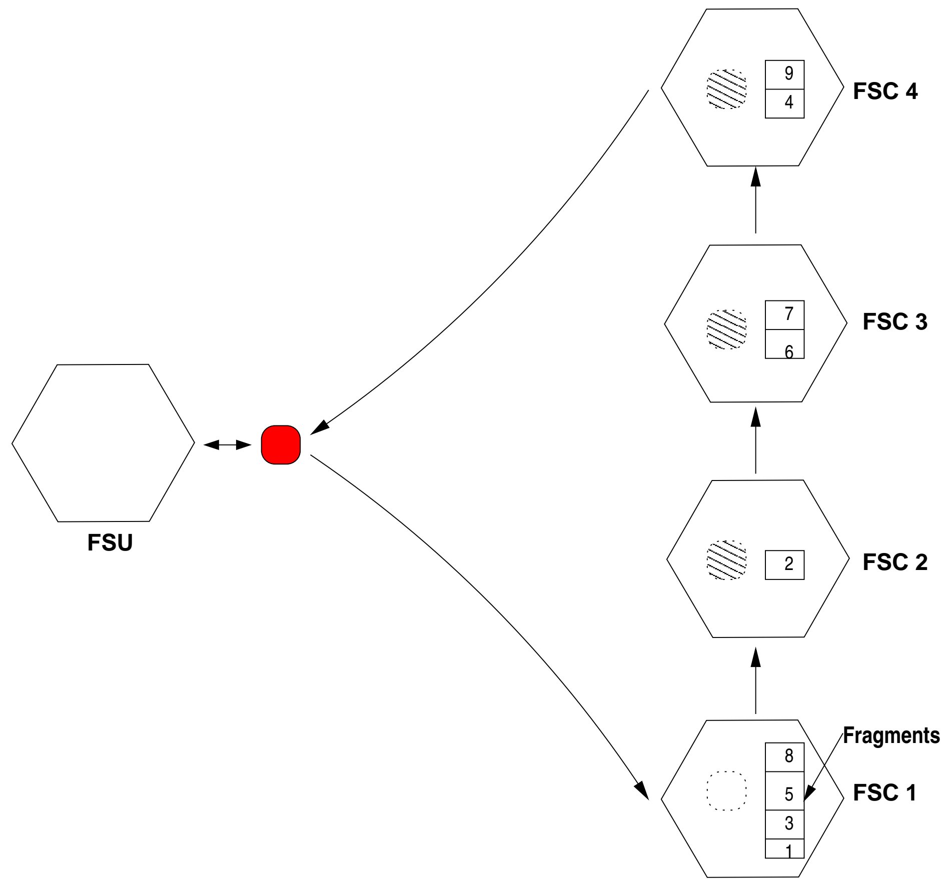

Figure 3.6: Example Fragment Retrieval Pattern

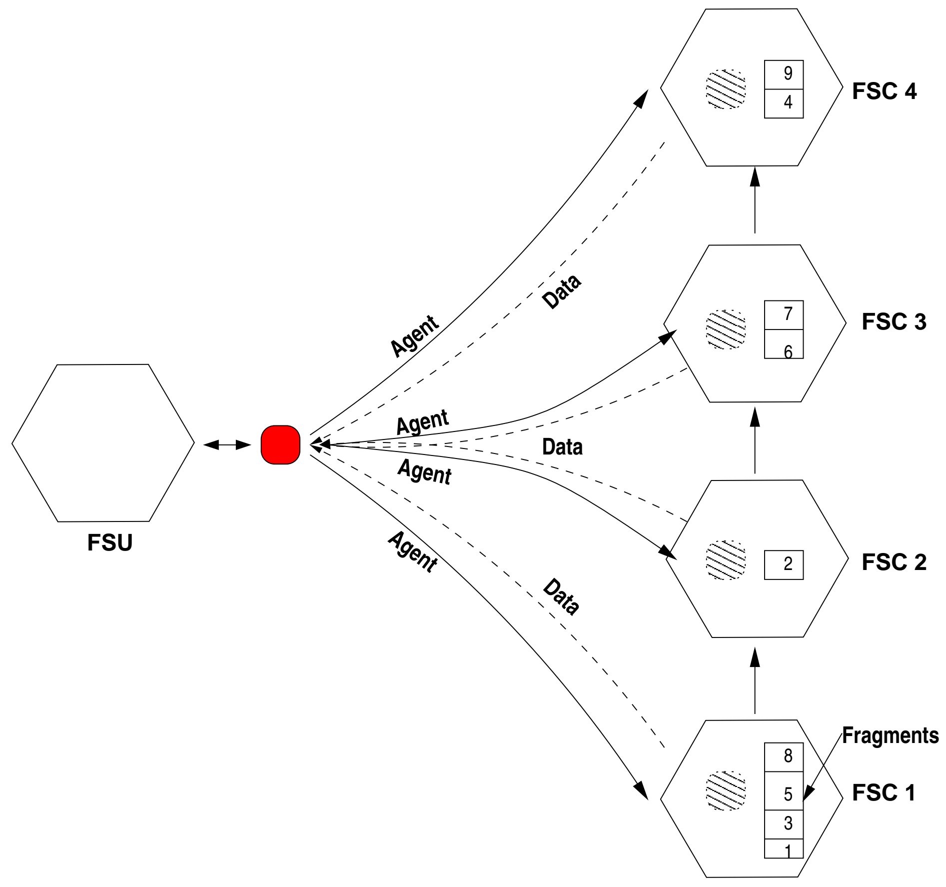

Figure 3.7: Improved Fragment Retrieval Pattern

fragments exist in various locations throughout the network is selecting the closest in terms of latency.

Fragment placement upon multiple contributors can be distilled into a similar problem as that of the retrieval: given that there are a set of fragments to transfer and a set of nodes that can store the fragments, the problem of transferring the fragments is merely the reverse of the retrieval. However, the critical issue in selecting nodes on which to place fragments is the selection of nodes that will be optimal for future retrieval. In addition, the selection of these optimal nodes could be a cumbersome and time-consuming task. Thus, in managing these constraints, it is assumed that an approach analogous to that of the *Sprite Log-Structured File System*[25] be taken.

With *Sprite* (a conventional local-disk file system), *writes* are optimized by positioning all data at the end of the log. Multiple writes of a single block would thus result in multiple copies of the block, each of which are versioned. Asynchronously, the file system collapses the log and deletes old versions of blocks. In the same way then, IAFS should version fragments, and write fragments to the closest nodes which contain free contribution space. These *collapsing agents* will be assigned the task of redistributing/replicating these blocks to optimize future retrievals. In projecting future retrieval patterns, these agents should also maintain statistics as to the usage patterns of stored data.

CHAPTER 4

Interfaces

4.1 File Fragment Transfer Protocol

One of the most unique facets of IAFS is that the responsibility of transferring file fragments is placed on the agent network, rather than the nodes that participate in the file system. Although this provides a useful abstraction in separating contributors and users from the details of data transfer across a network, a well-understood protocol must exist such that users and contributions can communicate to the agents. As outlined in section 4.3.2, such a protocol is referred to as a *Domain-Specific Protocol* (DSP) and serves to provide an abstract and convenient way to interact with agents.

The intent of the File Fragment Transfer Protocol (FFTP) is to define a DSP for the retrieval and storage of file fragments. IAFS Fragment Transfer Agents (FTAs) act as an intermediary between the FSU and an FSC, and thus the FFTP protocol will be used both by the FSU in issuing fragment transaction requests or by the agent in communicating requests to an FSC.

4.1.1 Functional Description

In developing an generic interface to agents that will transparently manage the transfer of fragments, the type of protocol (structure, encoding scheme, etc.) is driven by the architectural constraints. One constraint is the type of agents that will be used. For instance, if Java agents are used, a Java Serialization or Java Native Interface (JNI) communications framework would perhaps be the most intuitive (these features are built into the Java language). If, however, Perl or Python agents are used as well, (either in combination with or instead of Java agents), a more generic interface that would most elegantly support multiple types of languages would be required. Indeed, the main issue in the determination of the structure and encoding of FFTP messages is the ensurance that the chosen scheme does preclude the effective use of the protocol with any language or environment.

Another constraint that will determine the type of encoding scheme that will accommodate FFTP is the performance that the method will support. As with any engineering domain, when identifying areas for performance improvement, the most logical step to begin with is the identification of tasks that occur most often, and working to improve those tasks first. The transferral of fragments to and from agents will be the single most repetitive task that will occur on the FSC and FSU, and thus this protocol must be extremely efficient.

Although not an architectural constraint, the accommodation of direct communication between an FSC and an FSU could be very useful for performance benchmarking purposes. In order to facilitate this design goal, the FFTP protocol should be flexible enough to allow nodes to communicate directly across a network.

In order to generically support multiple agent types and to not incur the overhead of messaging packages such as Remote Procedure Call (RPC) or Abstract Syntax Notation 1, Basic Encoding Rules (ASN.1/BER), FFTP has been defined in terms of packets composed network-byte-order fields. Because FFTP only needs to support the straightforward task of requesting the retrieval and storage of fragments, and the parameters of these messages will be very simple, the complexity of implementing the FFTP protocol in this manner will likely be very low.

4.1.2 Protocol Definition

FFTP is a transaction-based protocol, defining transactions as request/reply pairs (see Figure 4.1). FFTP requests can be understood as a "laundry list" of fragments to retrieve or store. The transfer of these fragments over FFTP generally takes place locally (not across a network), thus multiplexing the throughput would not significantly decrease the transfer latency. For this reason, FFTP messages are serially encoded over a FFTP "connection." Every FFTP request message is preceded by a header that indicates the transaction type (see Table 4.1). Currently the FFTP specification defines the following types of request transactions:

- *Register*

  FSCs can contribute storage space to more than one IAFS file system, and

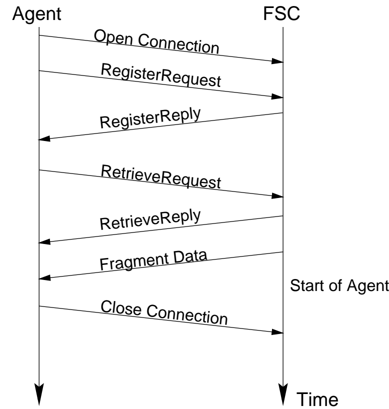

Figure 4.1: FFTP Retrieve Request Message Sequence Chart (MSC)

thus a series of FFTP transactions must be addressed to the appropriate file system instance. From the perspective of the agent, this message requests that the context for fragment storage/retrieval be switched to a particular file system. Alternatively, this message allows a FSU to signal *to* an agent that certain fragment should be retrieved from a particular file system. The *RegisterRequest* message consists only of an 4-byte IP address which denotes the multicast address that refers to a particular file system (see Section 3.6).

- *Retrieve*

  This allows the request for the retrieval of a particular fragment to be communicated to an agent from a FSU or to an FSC from an agent. *RetrieveRequest* contains the identifier of the fragment to be retrieved (see Figure 4.2). Because an IAFS file fragment is merely the value of a logical position of the fragment within an inode, the fragment ID consists of the combination of the inode identifier (*InodeID*) and the fragment number (*FragID*). The InodeID consists of 3 4-byte fields: the IP of the machine that created the file, the time of creation, and a random integer. It should be noted that the only requirement in the creation of the InodeID is that it be unique; the logical division is not a requirement. The fragment ID is a 8-byte integer that refers to the offset of the

fragment with the inode, and the *NumFragLocations* field denotes the number of existing replicated fragments that refer to this identifier. The list of actual network locations of these fragments conclude the request message.

- *Store* This allows the request for the storage of a particular fragment to be communicated *to* an agent or to an FSC *from* an agent. The *Store* message is composed of a fragment ID denoting the fragment to be stored, followed by the data that is to be transferred (see Figure 4.3). The size of the fragment data is that of the fragment size as configured by the file system. Note that the fragment locations that are specified in this message refer to the locations of the parent Inode and not to the locations of the current fragment, as the actual position(s) of this fragment may change as a result of this request and in this case the inode will need to be modified as well.

Table 4.1: FFTP Transaction Types

Transaction Type	**Parameter**
Register	0x1
RegisterReply (success)	0x2
Retrieve	0x3
RetrieveReply (success)	0x4
Store	0x5
StoreReply (success)	0x6
GenericReply (does not exist)	0x7
GenericReply (not registered)	0x8
GenericReply (unknown error)	0x9

For each request, FFTP also defines associated reply messages. *RegisterReply* contains only the transaction type which may indicate a successful registration, or a generic error response type. The *StoreReply* message is merely the *Retrieve* message containing the corresponding transaction type. In the same manner, the *RetrieveReply* message is exactly the same as the *Store* message, but is preceded by the *RetrieveReply* transaction type.

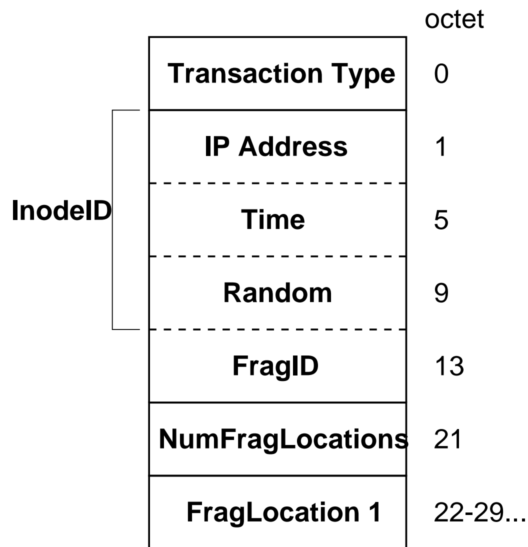

Figure 4.2: FFTP Retrieve Request Message

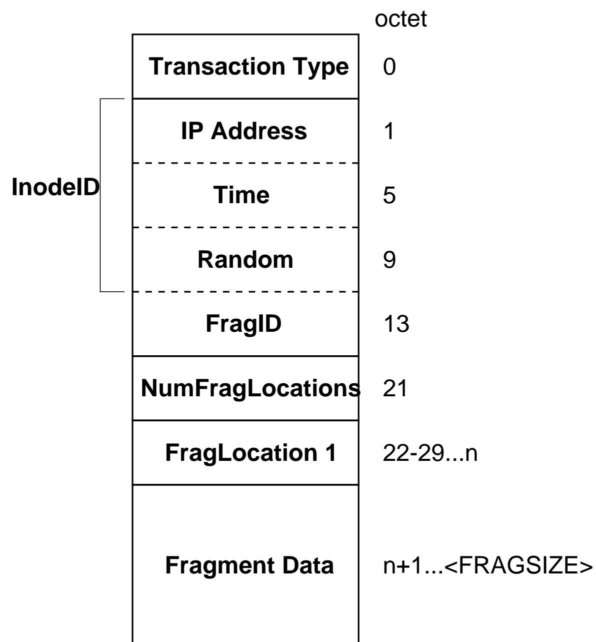

Figure 4.3: FFTP Store Request Message

4.2 File Operation Transaction Protocol

The intent of the File Operation Transaction Protocol is two-fold. First, the protocol will presumably be used to communicate file operations over a character device, eventually allowing kernel-level file transactions to be communicated to the FSU. Therefore, FOTP assumes a single path, bi-directional mode of communication, and all transactions are *multiplexed* upon this single communications channel.

Second, FOTP represents a useful abstraction. Because the protocol is generic as to the *type* of file system used beneath, this single point of communication that rests between file system operations and inode operations allows the reuse of these two levels of the IAFS file system. Additionally, the break between the *kernel-level* operations and *network-level* transactions is strategically placed, facilitating the integration of other file system-level front ends, such as NFS. Indeed, the migration path of IAFS toward a multi-platform file system will undoubtedly include adding an NFS front end to the file system, and mapping NFS operations into FOTP transactions.

4.2.1 Functional Description of FOTP

FOTP as an application-layer protocol is connectionless and assumes a reliable transport protocol. In practice, however, it is expected that a connection-oriented transport mechanism, or even a block device will be used. Every FOTP message is either a FOTP request or FOTP response, each associated with a unique transaction ID. For this reason, FOTP traffic can be "serially multiplexed" over a network connection (see Figure 4.4).

Currently, the FOTP specification includes the following operations:

- *Register File System*

  This is an explicit request to the FSU to initialize a new file system. In actuality, this could be replaced by an implicit request such that any other operation would notify the FSU that a file system should be registered if it hasn't been already. However, file system registration may be deemed a privileged action to be performed only by authorized users, and thus implicitly deriving this action

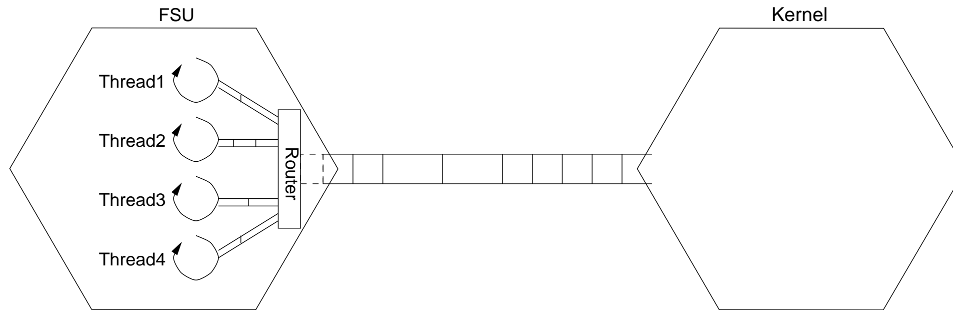

Figure 4.4: FOTP Message Serialization

based upon other, non-privileged requests could undermine security policies that may be implemented by the user.

- *Open*

  This operation signals the FSU that it should open a new file context. This operation could be implicitly assumed by the FSU if other requests are received and the associated file context has not been initialized. However, because the *Open* operation could fail (because of permissions, etc.) this operation allows the user to verify that he can access a particular file.

- *Close*

  The *Close* operation requests that the FSU close a file context. If the agent network synchronously writes data, this operation might be asynchronously processed, otherwise this is a synchronous action (if asynchronous writes are implemented).

- *Read*

  The *Read* request allows the user to request the retrieval of sections of file data.

- *Write*

  This operation requests that the passed modifications to the file context be applied. If synchronous writes are implemented by the agents that service the FSU, this return of this operation guarantees that data has been reconciled

with the file system. If asynchronous writes are used, this operation may return immediately.

- *Stat*
  This *Stat* operation requests file information (modification date/time, file type, etc.) be returned.

- *Set Parameters*
  This operation allows users to set file information parameters.

Each of these message types has an associated response message. From the perspective of the application, every message is sent in its entirety such that no fragmentation of messages (above the transport layer) is done. For this reason, FOTP transactions can be conceptualized as a remote procedure call with a return value. For instance, a *Read* transaction request would be accompanied with parameters indicating the position in the file to begin reading, and the number of bytes to read. The response would indicate the success status of the transaction, the number of bytes read, and the data.

The initial version of the FOTP specification outlined a byte-by-byte protocol, defining for each message a list of Network Byte Order (NBO) fields of varying sizes. However, this approach is cumbersome because there is a great deal of optional data, and many messages have numerous parameters, each of which are composed of multiple aggregates (see Section 4.2.2). In addition, the amount of code required to effectively implement such a protocol would be unintuitive and very architecture specific. For this reason it was decided that FOTP would utilize a data encoding scheme that would facilitate transparent message transfer. Ideally, from the perspective of the application, the creation of FOTP messages would be as simple as filling in a data structure and then calling an auto-generated routine to encode that structure. Although there are numerous encoding schemes and toolsets that could accomplish this degree of automation, the most portable and widely used are XML, RPC, and ASN.1.

Perhaps the most flexible solution would be to encode transactions in the Extensible Markup Language (XML)[4]. Because industry is beginning to move toward the XML standard in the development of open interfaces, this approach would provide a generic communication framework to the IAFS. However, currently the XML specification has not fully optimized the transferral of large quantities of binary data: all data is transmitted as ASCII/Unicode characters. Industrial partners of XML such as Ericcson, IBM, Motorola, and Phone.com have drafted recommendations involving the compression of binary streams in XML (such as WBXML) which can be used in transferring data using the over the Wireless Access Protocol (WAP)[16]. However, a general-purpose binary-data optimization scheme has not yet been generically implemented. For this reason, XML would likely prove too inefficient for the IAFS at this time.

Sun's RPC would also work well as a framework for message transfer, but this approach could be counterproductive. First, RPC implementations have been notoriously non-portable. In addition, RPC has been closely tied with C/C++, although some Java RPC implementations exist. Because the primary goal of IAFS is interoperability, neither would RPC be the optimal solution at this point.

Whereas XML and RPC are widely used in the used in the Internet community, the ITU's Abstract Syntax Notation number One (ASN.1) structure encoding scheme[6], has long been utilized within the Telecommunications industry. ASN.1 is an open standard with a number of freely available implementations. In addition, many such implementations of ASN.1 have been highly optimized for speed and performance. Most importantly, however, the ASN.1 syntax allows protocols to be precisely and intuitively specified, providing an clear protocol definition to be used in the ultimate migration to other data encoding frameworks such as XML.

4.2.2 Protocol Definition

The ASN.1 protocol specification for FOTP is detailed in Appendix A. Following is a textual description of each FOTP message type:

FOTPMessage

Every FOTP Abstract Packet Data Unit (APDU) is encapsulated in the *FOTPMessage* APDU. The *FOTPMessage*, in addition to containing the transaction message, contains fields that is required for all message types. Among these are the Transaction ID (*transID*), and File System ID (*fsID*).

The *transID* serves to allow a FOTP request to be matched for its reply. For all practical purposes, the transaction ID should merely be unique for all currently pending transactions. In practice, however, it is assumed that this is a concatenation of the PID of the process initiating the transaction, and a counter of transactions for that process which is incremented for each request. The *fsID* is sent because one logical FOTP circuit may carry traffic from multiple "mounted" file systems to a single FSU. The FSU must then route responses back to the mounted file system which initiated a transaction.

FOTPRegisterFS

The *RegisterFS* message requests that the FSU recover the "root" inode ID (that of "/"). The *RegisterFSReply* may return only an *FOTPRetStatus* indicating a failure due to access rights, etc. or a successful status with the root inode ID (*rootInode*). Upon a successful *RegisterFS* transaction, the file system may be considered "mounted" in the system kernel.

Typically, the *RegisterFS* transaction only takes place at the time of FS mount. However, like any other inode in the IAFS, the location of the root inode may change. In this case, a status of *notFound* would indicate to the kernel that an re-registration is necessary.

FOTPOpen

In the case of existing files the *open* message serves as a explicit indication to the FSU to retrieve the an Inode and allocate a context (if this context does not already exist). Because the *read* or *write* transactions must also supply an inode ID, this can

actually be seen as a redundant message. However, in the interest of performance it could be useful to force the FSU to "pre-cache" the inode for future use.

If the request is pertaining to the creation of a new file, the *inode* field is filled according to the ID and location of the parent inode. This information is an optimization that allows the agents dispatched by the FSU to "connect" the inode to a directory without the an extensive search for the inode.

FOTPClose

Like the *open* transaction, the close message is a explicit indication to the FSU that a context is no longer in use and can be deleted. Because the FSU synchronously modifies the inode upon request and immediately dispatches *write* requests, the context is primarily a caching mechanism. The *close* message includes an inode ID and the *closeReply* includes the return status of the close request.

FOTPRead

The *FOTPRead* APDU requests the FSU to dispatch agents to read *length* bytes from the file *inodeID* beginning at *offset* bytes from the beginning of the file. *readReply* returns the *status* of the transaction and the *data* which was read. It should be noted that because *length* is a two-octet integer at most 65535 bytes can be read in one request.

FOTPWrite

The *write* transaction requests the synchronous modification of *length* bytes of the file *inodeID* beginning at *offset* bytes. Because *length* is a two-octet integer, the amount of data that can be written in one transaction can be at most 65535 bytes and is contained in *data*. The *writeReply* message returns the *status* of the transaction indicating the success or failure. If the transaction was at least partially successful, the number of bytes successfully written is stored in *length*.

FOTPStat

The *stat* request enables the kernel to retrieve information regarding a file *inodeID*. Among the parameters that can be returned in *statReply* are the file permissions (*mode*), User ID (*uid*), Group ID (*gid*), file size in bytes (*size*), last access time (*accessTime*), file creation time (*creationTime*), last modification time (*modificationTime*), deletionTime (*deletionTime*), number of hard links (*linksCount*), number of fragments pointed to by the inode (*fragsCount*), file flags (*flags*), and version of the file (*version*). If the request was unsuccessful, the only field returned will be *status*.

FOTPSetParms

The *setParms* request allows the modification of file attributes by the kernel. The inode ID (*inodeID*), file permissions (*mode*), user ID (*uid*), group ID (*gid*), last access time (*accessTime*), creation time (*creationTime*), modification time (*modification time*), deletion time (*deletionTime*), and file flags (*flags*) are among the parameters that can be set by the kernel. *setParmsReply* indicates the *status* of the request.

4.3 Distributed Agent Delivery System

This section discusses the goals, design and implementation of a particular multi-lingual IMA development kit, the Distributed Agent Delivery System (DADS). DADS operates upon the Agent Delivery Protocol (ADP), which is a multi-lingual solution as it supports multiple agent languages types. In addition, because ADP is a simple, lightweight protocol that does not use any cumbersome encoding scheme, DADS should scale well in performance-sensitive environments. The DADS architecture is consists of the following components:

- IMA-Based Transfer Protocol Implementation
  Agents and nodes must support a common protocol to support the migration of agents. In order for an agent to move itself this protocol must be supported from the perspective of the agent

- Patron-Based Transfer Protocol Implementation
  The patron refers to a node which utilizes agent services. For an agent to migrate to a patron, the patron must support the agent transfer protocol.

- IMA Interpreter (on every participant node)
  Because agents move to and from patrons which are of variant architectures, a platform-independent, interpreted language must be chosen and supported on every patron.

- IMA
  The IMA, itself, is a crucial element towards the development of an agent-based system. Without exhibiting a specific behavior, the infrastructure supporting the mobilization of an agent is useless.

- Domain-Specific Protocol (to be used between agents and nodes)
  Because most (but not all) IMAs will need to communicate with the patrons, an extensible protocol should be developed.

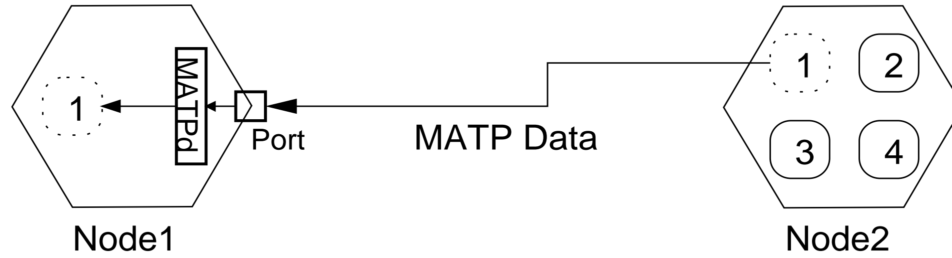

Figure 4.5: Agent Transfer from Node 2 to Node 1

Of these items, only the domain-specific protocol and the agent code fragment are specific to a particular IMA-based system. That is to say that the *IMA Interpreter*, and the implementations of the *IMA-based Transfer Protocol* and the *Patron-Based Transfer Protocol* can be reused in any system that utilizes similar underlying communication protocols. For instance, every agent *will* be transferred from one location to the next via a well-understood protocol over a well-known network port, and every agent can be interpreted by at least one interpreter (see Figure 4.5). DADS thus encapsulates the agent-based transfer protocol, the patron-based transfer protocol, and the integration of one or more interpreters.

In the following sections, the design and implementation of the patron and agent portions of the Distributed Agent Delivery System will be discussed. In addition, a detailed specification of the ADP protocol that allows for generic communication of IMAs across connection-oriented and connectionless networks will be given. Finally, security issues will be addressed followed by a discussion of future research on DADS.

4.3.1 ADP Protocol Specification

The ADP protocol is actually the composition of two distinct protocols, one of which supports connection-oriented, reliable transport, and the other supporting connectionless, unreliable connectivity. Currently, ADP depends upon a reliable connection-oriented transport mechanism such as TCP. However, the ADP specification also

supports a connectionless communication scheme that is designed such that any connectionless transport medium could be used as well. Indeed, ADP is intended to rest atop the transport-layer of a protocol stack (such as IP or X.25) and not at the network-layer as have other IMA implementations such as *Active IP*[33]. An ADP-based agent is thus expected to provide its own *pseudo-application* layer to applications which make use of its services.

Both the connection-oriented and connectionless ADP protocols are similar in content such that they communicate IMA code and data in a segment that is referred to as an Agent Data Unit (ADU). The ADU consists of code and data which is compressed according to the compression specification format specified in RFC1950[9]. The length of these fields are included in the ADU, along with the uncompressed length which is used in the allocation of buffer space to accommodate the uncompressed size.

Connection-oriented ADP is recommended when agents (including their data) will be rather large (more than a few hundred kilo-bytes, for instance) and the overhead of transporting them is outweighed by the need for a reliable connection. Because the transport-layer ensures the ordering and integrity of the data, there is no need to support reverse acknowledgements. An example MSC of a successful ADU transfer can be seen in Figure 4.6. Only two possible outcomes exist in the transfer of an ADU across a reliable connection: the ADU will be transferred successfully and the connection will be closed by the initiating agent, or the connection will be lost and the receiver will be left with an empty or truncated ADU. Since the partial delivery is supported under connection-oriented ADU transfer, the ADU format only contains the fields necessary for the delivery of agent text and data (see Figure 4.6).

In the support of an unreliable, connectionless transport medium (such as UDP), the associated MSC would contain additional messages (see Figure 4.8). Because the transport layer is unreliable, reverse acknowledgements are needed. Many connectionless transport-layer protocols set bounds on the maximum size of a packet, or the Maximum Transport Unit (MTU). For this reason, a provision must be made for the division of the ADP ADU into multiple fragments or "capsules." Since the order of delivery cannot be guaranteed, a two-byte agent identifier plus a one-byte Message

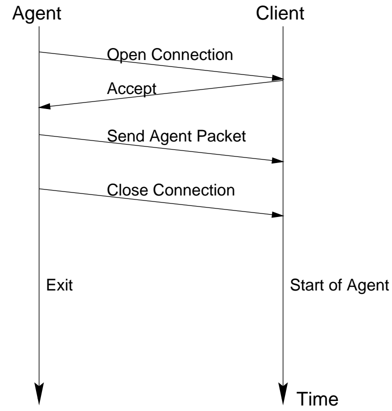

Figure 4.6: Connection-oriented ADP MSC

Field	Octet
Agent Type (Language)	1
Compressed Code Length	2
Uncompressed Code Length	6
Compressed Data Length	10
Uncompressed Data Length	14
Compressed Code	15 ... n
Compressed Data	15+n

Figure 4.7: Connection-oriented ADP Packet

Sequence Number (MSN) must be added to every fragment that is sent (see Figure 4.9).

Figure 4.8 illustrates the MSC for a successful transfer of a connectionless ADP ADU. ADP utilizes a positive acknowledgement scheme, which consists of the two-byte agent identifier plus the one-byte MSN. The sequence number indicates the last successfully received capsule (starting with 1). The sequence number zero is reserved to indicate packet transfer failure.

From the perspective of the IMA that is migrating to a patron, fragments should be sent in "bursts" consisting of a collection of capsules (the exact number is left as an implementation detail, but should be between 1 and 10). Simultaneously, the IMA should allocate another thread of control that waits for acknowledgements. The MSN of each acknowledgement should be stored, replacing the last. After each burst the sender should validate the stored MSN. If the MSN was contained within the range of at least the last two bursts, the sender should resume the bursts normally. If the stored MSN is not within the range of the last two bursts, the sender should wait for the stored MSN to be updated for two times the average round-trip time (RTT) between the time that a capsule was sent and its acknowledgement was received, at which point the entire burst is retransmitted. If an acknowledgement with a MSN of zero is received, the transfer is halted.

The receiver will simply wait for the reception of capsules of a particular *Agent ID*. Following every new successfully received, consecutively delivered capsule, a MSN counter should be incremented and a new acknowledgement containing the MSN should be sent. All capsules should be ordered according to their MSN and duplicate capsules should be discarded. If no packet has been received within the timeout period, or if the transfer must be stopped due to a catastrophic software failure on the Patron, an acknowledgement with a MSN of zero should be returned. The exact value of this timeout period can be configured, but should be long enough to ensure that an error has occurred, and short enough that the timeout delay does not significantly impact performance (from the perspective of the user).

The initial phase of the DADS (supporting connection-oriented ADP with with no support for key-based security) has been completed. Although this comprises the

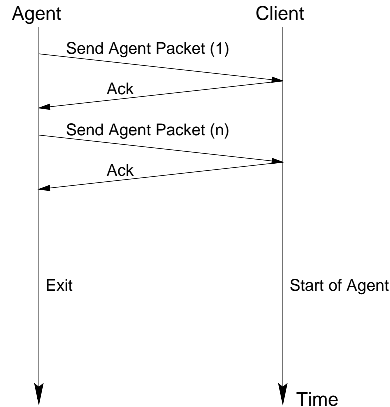

Figure 4.8: Connectionless ADP MSC

bulk of the work toward the completion DADS development project, the completion of the connectionless portion of the ADP protocol suite will enable DADS to represent a viable solution for generic, multi-lingual agent development.

Current work is focusing on the implementation of the connectionless (UDP) ADP protocol. UDP support will be critical for the ultimate support of the Intelligent Agent-Based File System (IAFS), which will depend on reliable high-performance data transfers. Because the connection-oriented ADP implementation has already been completed, the addition of UDP support should be straightforward.

4.3.2 DADS Design

In the following sections the *Patron-Based Transfer Protocol* requirements and design will be presented along with some pertinent implementation details. This section will include information regarding the *IMA Interpreter* that will be used to execute an agent. In addition, a Perl[23] *IMA-Based Transfer Protocol* implementation will be described, along with notes regarding the API that an agent designer would use to write an *IMA*. Finally, an example agent system will be described, outlining the use of a *domain-specific protocol*.

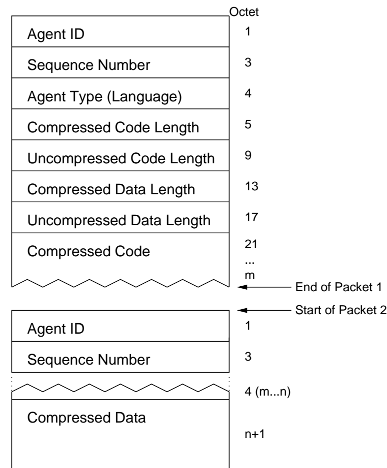

Figure 4.9: Connectionless ADP Packet

ADP/Patron Design

In the creation of a network service that receives, executes, and manages agents for an entire node, the first assumption is that there *can* be a large number of agents that will be in motion. For this reason, the first requirement dictates that the "Agent Daemon" be as efficient as possible and support multiple streams of control in order to optimize the processing of multiple concurrent agents. In addition, such a daemon is expected to be written only once, so it must be largely platform independent such that the portability of the daemon can be maintained. This implies a loose coupling between platform-unspecific modules and the rest of the system. Finally, since the ideal agent implementation language has not yet been determined, DADS must be extensible. The daemon must support multiple IMA language types and allow for the support of future types in a generic fashion.

The implementation of the ADP/Patron can perhaps be best represented at a high level as a protocol stack (see Figure 4.10). As is typical of most protocol stacks, arriving data (agents) can be understood as moving *up* the stack, whereas outgoing data moves *down* the stack. Since the ADP/Patron *accepts* agents, the data flow will be described as moving *up* the stack.

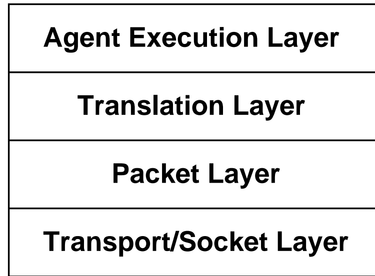

Figure 4.10: ADP/Patron Protocol Stack

The Transport/Socket Layer can be understood as the level of the application which interfaces with the BSD Socket layer (or any transport interface). This layer merely writes (acknowledgements) and reads data (transport units), passing read data to the packet layer, which assembles transport units into an ADP ADU. The Translation Layer reads the fields in the packet into a ADU object, and decompresses agent code and data. Finally, the Agent Execution Layer is responsible for executing the agent according to the type held in the ADU object. This layer represents the core of the agent daemon.

Currently, the ADP/Patron only supports Perl agents, which are perhaps one of the more complicated agents types that will be supported. In the current implementation, all agent types extend the base class *Agent*. Support for Java,[11] Python, etc. could be easily by creating a new agent class and by inheriting the functionality provided by the *Agent* class.

Because the Perl agents must be executed by a monolithic Perl interpreter (a binary which is typically rather large), execution of large numbers of Perl agents (each of which causes the *fork()* and *exec()* of a new interpreter) could be more costly than the interpretation of the agent itself. For this reason, DADS maintains a process table, or a "pool" of running Perl processes, each of which is merely waiting for input in the form of Perl code. This code and the persistent data that an agent saves prior to its migration is transferred to the Perl processes prior to the agents resumed execution.

ADP/Agent Design

In comparison to the requirements of the ADP/Patron, those of the ADP/Agent are seemingly trivial: an agent must be able to move from one host to another and initialize itself at the point of execution at each new node. However, because the programming interface for the development of new agents will, in effect, be the primary point of contact between users and the Development Environment, the interface must be intuitive and above all, simple.

The design of a particular ADP/Agent implementation would be roughly the same as that of the lower three layers of the ADP/Patron. However, in the case of ADP/Agent the data can be described as moving *down* through the stack (see Figure 4.10). Thus, the translator will compress data/code, and the packet layer will compose transport units rather than assembling them. The agent execution layer is not relevant because the agent sends itself, and it is *already* executing.

Because DADS currently supports Perl, this discussion will focus on the implementation of the Perl module which will provide an Application-Program Interface (API) for the developer of a new agent. The implementation of APIs for other agent types should follow the same model and would likely be nearly as straightforward. The following represent the entirety of the interface:

- *Init()*

  Agents that have just migrated need to recover the data that has accompanied them. This method should be called immediately following a migration.

- *Move(LOCATION)* To migrate to another host, an agent should call this method to move to a new location. The task of packing and compressing the data and agent code is encapsulated within this routine.

- *Event(SEVERITY, EVENT)* Agents that need to report an event can use this interface. Events will be fed to a logging mechanism on the currently running host.

In addition, the agent is expected to place all data and state information that will be accessed across moves in the global stack *@MyData*. This data will be serialized

in network-byte order and compressed prior to a move, and decompressed and deserialized following a call to *Init()*.

In order to preserve their state across migrations, agents must record all persistent data and provide a way to regain that state after a move. All agent implementations, regardless of their purpose, will thus naturally take on a characteristic appearance. For instance, the following configuration management agent illustrates the structure of a simple state-preserving agent (for information regarding the Perl module *ADP*, see Appendix B):

```
require ADP;
use ADP;

# State Definitions
$collectConfigurationData = 1;
$reportConfiguration = 2;

# Global variable declarations
$state;
$omcHost;

sub CollectConfiguration
{
     # Obtain Configuration Information Here
     ADP::Event( $ADP::INFO, ``Collecting configuration data...''  );
     ReadConfig(); # Not implemented

     # Now, save our state and move to the network management node and report
     # the configuration
     push( @ADP::myData, $reportConfiguration );
     ADP::Event( $ADP::INFO, ``Moving to OMC...''  );
     ADP::Move( $omcHost );
```

```
}

sub ReportConfiguration
{
     ADP::Event( $ADP::INFO, ‘‘Writing Configuration data...’’  );
     WriteConfigData(); # Not implemented
}

sub main
{
     ADP::Init();

     # Retrieve our state
     $state = shift( @ADP::myData );

     # Copy the Network Management Host from the top of the stack
     $omcHost = $ADP::myData[ 0 ];

     ADP::Event( $ADP::INFO, ‘‘Agent Initialization Complete.’’  );

     if( $state == $collectConfigurationData )
     {
          CollectConfiguration();
     }
     elsif( $state == $reportConfiguration )
     {
          ReportConfiguration();
     }
}
```

```
void main();
```

This configuration management agents merely reads configuration data from hosts and relays the data to an Operational Maintenance Center (OMC). This particular agent expects its state to be at the top of the saved stack *MyData* and the OMC host address to be immediately below. The agent first recovers the persistent data into variables that can be globally accessed and resumes execution according to the saved state. Regardless of the complexity, all agent implementations will likely follow this pattern.

Domain-Specific Protocol

In developing an agent system that *forces* the separation between the details of a communication system and the data that it communicates, designers must develop an abstract protocol for the transfer of the data from the agent to the rest of a system. Because the agent can manipulate the data presentation, this data protocol can take the most convenient and/or most intuitive form [17].

For instance, an agent which communicates to patrons that are running embedded telephony software with an SS7[8] interface could present an RPC[28] interface to the patron. Requests could be forwarded and serviced as SS7 transactions, but from the patron's perspective the interface to the embedded system is RPC, not SS7. Likewise, an agent could relay Web data across a series of firewalls or gateways, and present HTTP data directly to the requesting patron. In these examples, however, the agent must have a method to communicate with the patron. One approach is to designate an internal port (such as a TCP socket, named pipe, etc.) that allows a generic agent to communicate with a patron. However, specific agent types, such as Java, have built-in IPC mechanisms that would allow an agent to interface directly to the patron (without a secondary protocol).

Agents allow designers to disregard the location and format that data takes on its path across the network, but one must nevertheless consider the ultimate format that the data should take when it is presented by the agent. In developing a protocol

to communicate with IMAs, it should be flexible and extensible. The advantage of agents could be undermined by a poor protocol that limits its communication with the rest of a system[34]. It should be noted that some agent systems will not require the creation of such a protocol. For instance, an agent which simply travels across a series of workstations and executes a command on each might only return an exit status to the initiating patron.

4.3.3 Security

If an agent system is to be widely used and accepted by the Internet community one of the most important concerns is security. Agents are ideal mechanisms for the transfer of "trojan horses," and should thus be dealt with as to minimize and possibly eliminate the danger posed by a malicious or accidental attack[2].

One approach is to only accept agents that are known to have been authenticated by a trusted agent authenticator. For instance, suppose that an organization created a distributed system that used agents as the medium of data transfer. In general, all agents would be the same, so all agents could be encrypted with the authenticator's private key who's public key would be known by all. Any agent who is not authorized to run on this system by the authenticator would not be successfully decrypted at all.

This approach would work well in a system who's agent population is relatively homogeneous, but would be cumbersome if new kinds of agents were constantly being created. In this case, DADS would provide a "sandbox" for agents to execute within. The creation of such a sandbox would greatly depend on the agent type and platform. For instance, Java's built-in sandbox would perhaps be sufficient whereas other agent types would necessitate the creation of a new type of sandbox. On multi-user platforms, one approach would be to execute agents as an untrusted user, restricting their privileges to that of a common user of the system.

The preferred approach would be to provide sandboxes to untrusted agents, and to allow *trusted* agents to run with extended privileges. Currently, all agents execute in a sandbox. The implementation of privileged agents would necessitate the addition of two fields in ADP: a trusted host (IP) field, and a trust level.

The inclusion of security features will need to be completed in order for the system to be practical in all but the most secure networks. In this effort, the importance of involving as many security domain experts as possible in an open development forum is paramount. Because an agent-based network solution is so very susceptible to attack by malicious "hackers," the ADP must be rendered as secure as possible.

CHAPTER 5

Conclusion and Future Work

The discussion of the administrative aspects of an Intelligent Agent-Based File System represents only one facet of the growingly large pool of research areas uncovered by the IAFS project. Indeed, the analysis and further development of the IAFS agent protocols, a thorough analysis of IAFS performance characteristics versus the type of agent used, and a comparison against existent distributed file systems are among the future areas of investigation.

This paper has described how the administration of a distributed file system can be achieved through the use of Intelligent Mobile Agents. A functional description of IMAs and distributed file systems has been presented, and the evidence of the thesis of this work has been presented within the context of the following domains of research:

- Distributed Administration
- Transient Contribution of Resources
- QoS Requirements
- Administrator Location
- Dynamically Reconfigurable Behavior

Conventional centralized and distributed file systems have not effectively confronted the decentralization of file system administration and the dependence upon a human administrator. IAFS solves this problem through the use of a system of Administrative Agents and the network of asynchronous mobile agents which work on behalf of either the Administrator, the File System Contributor, and the File System User.

IAFS also assumes that storage contributors are for short-term use and thus provides a facility for the migration of data off of soon-to-be-removed resources (Darwinian Contributors). The assumption upon static contribution (typical among conventional distributed file systems) is treated as a special case of this general framework. Thus the addition and removal of resources is simplified and eliminates user intervention.

The distribution of file services upon a large number of unreliable resources naturally implies a number of Quality-of-Service issues. The IMA infrastructure of IAFS (DADS) allows for simplified customization (by the user of IAFS) of fault tolerance and file availability issues through the development of user-specific agents.

Since the entirety of the communication of the file system is accomplished through the use of agents, and because the relationship between the agents and the network elements has been formally (and generically) defined, the behavior of the agents has been effectively encapsulated. Such encapsulation also allows the existence of multiple variations of a single agent type to be present with an IAFS instance. Particular users can customize security, fragment placement, etc. without having to alter the behavior of the overall IAFS network.

Through the investigation of the preceding issues, we have found that the use of agents in distributed file services can facilitate many architectural and implementation optimizations and can, indeed, present a new paradigm for distributed application development. Most importantly, however, Intelligent Mobile Agents allow the problem of the decentralization of administrative duties to be effected.

5.1 Future Work

Currently, the implementation of the IAFS file system is nearing completion. The development of the File System Contributor is complete and the File System User is also nearly finished. The development of a generic set of OS interfaces (perhaps a NFS front-end) is required for the testing and further development of IAFS. In addition, the low-level storage mechanism for file fragments (implemented by the FSC) is currently very UNIX-centric and should be expanded to accommodate a

wider range of operating environments.

The completion of the initial implementation of IAFS will be followed by an extensive testing period: the implementation of IAFS has grown to be rather large and will doubtlessly yield the discovery of minor design oversights and implementation errors. Nevertheless, it has already been shown that the major components of IAFS (FSU, FSC, and DADS) work in isolation, so this phase will be primarily a period of the integration of the various subsystems.

Following the first stable release of IAFS, the period of true research, discovery, and experimentation can begin. Among the areas of research that will be initially explored are the orientation and communication patterns of agents in various types of environments such as heavily loaded, low bandwidth, and high-latency networks.

Finally, after producing a reliable, robust, and efficient distributed data storage platform, IAFS will be used to research application domains that could benefit from a distributedly-administered architecture. For instance, real time multimedia streaming could benefit from the IAFS architecture: data could be dynamically re-located to more efficiently serve clients requesting high-throughput or low-latency QoS.

The work done on the IAFS project has already yielded extensive information regarding the development of large-scale agent based systems. At the very least, the understanding of agent-based communication architectures has been forwarded through this research project. It is reasonable to believe that further research will yield further information regarding distributed administration through the use of agents, and of course concerning the overall viability of agents in high-performance architectures. Because there are a number of other distributed file systems that can serve as a benchmark for performance comparisons, IAFS should prove to be an excellent testing ground for proving or disproving the viability of agents as a multi-purpose communication paradigm.

APPENDIX A

FOTP Protocol Specification

```
--
--^=============================================================
--
-- Name:     FOTP.asn
--
-- Version:  1.1
--
-- Purpose:  This is the File Operations Transaction Protocol.
--
-- =============================================================


FOTP DEFINITIONS ::= BEGIN

UInt1  ::= INTEGER (          0 ..        255)
UInt2  ::= INTEGER (          0 ..      65535)
UInt4  ::= INTEGER (          0 .. 4294967295)

SInt1  ::= INTEGER (       -128 ..        127)
SInt2  ::= INTEGER (     -32768 ..      32767)
SInt4  ::= INTEGER (-2147483648 .. 2147483647)

IPAddr ::= UInt4
BinString ::= OCTET STRING (SIZE (0 .. 524280))

UInt8 ::= SEQUENCE {
    lsWord                 [1] IMPLICIT UInt4,
    msWord                 [2] IMPLICIT UInt4
}

FOTPMessage ::= SEQUENCE {
    transID                [1] IMPLICIT UInt4,
    fsID                   [2] IMPLICIT IPAddr OPTIONAL,
    message                [3] CHOICE {
        registerFS         [1] IMPLICIT FOTPRegisterFS,
```

```
        registerFSReply      [2] IMPLICIT FOTPRegisterFSReply,
        open                 [3] IMPLICIT FOTPOpen,
        openReply            [4] IMPLICIT FOTPOpenReply,
        close                [5] IMPLICIT FOTPClose,
        closeReply           [6] IMPLICIT FOTPCloseReply,
        read                 [7] IMPLICIT FOTPRead,
        readReply            [8] IMPLICIT FOTPReadReply,
        write                [9] IMPLICIT FOTPWrite,
        writeReply           [10] IMPLICIT FOTPWriteReply,
        stat                 [11] IMPLICIT FOTPStat,
        statReply            [12] IMPLICIT FOTPStatReply,
setParms             [13] IMPLICIT FOTPSetParms,
setParmsReply        [14] IMPLICIT FOTPSetParmsReply
    }
}

FOTPRegisterFS ::= SEQUENCE {
}

FOTPRegisterFSReply ::= SEQUENCE {
    status                   [1] IMPLICIT FOTPRetStatus,
    rootInode                [2] IMPLICIT FOTPInodeID OPTIONAL
}

FOTPInodeID ::= SEQUENCE {
    ipAddr                   [1] IMPLICIT IPAddr,
    time                     [2] IMPLICIT UInt4,
    random                   [3] IMPLICIT UInt4
}

IPAddrLst ::= SEQUENCE OF IPAddr

FOTPFragLocation ::= SEQUENCE {
    numCopies                [1] IMPLICIT UInt1,
    ipAddrs                  [2] IMPLICIT IPAddrLst
}

FOTPInodeInfo ::= SEQUENCE {
    inodeID              [1] IMPLICIT FOTPInodeID,
    inodeLoc             [2] IMPLICIT FOTPFragLocation
```

```
}

FOTPOpen ::= SEQUENCE {
    type                    [1] IMPLICIT ENUMERATED {
        existingFile (1),
newFile (2)
    },
    -- In the case of creating new inodes, this is filled in
    -- according to the parent directory.
    inode                   [2] IMPLICIT FOTPInodeInfo
}

FOTPRetStatus ::= ENUMERATED {
    ok      (0),
    notFound     (1),
    noSpace  (2),
    noData  (3),
    accessDenied (4),
    full         (5),
    unknown   (6)
}


FOTPOpenReply ::= SEQUENCE {
    status                  [1] IMPLICIT FOTPRetStatus,
    inode                   [2] IMPLICIT FOTPInodeID OPTIONAL
}

FOTPClose ::= SEQUENCE {
    inodeID                 [2] IMPLICIT FOTPInodeID
}

FOTPCloseReply ::= SEQUENCE {
    status                  [1] IMPLICIT FOTPRetStatus
}

FOTPRead ::= SEQUENCE {
    inodeID                 [1] IMPLICIT FOTPInodeID,
    offset                  [2] IMPLICIT UInt8,
    length                  [3] IMPLICIT UInt2
```

```
}

FOTPReadReply ::= SEQUENCE {
    status                  [1] IMPLICIT FOTPRetStatus,
    data                    [2] IMPLICIT BinString OPTIONAL
}

FOTPWrite ::= SEQUENCE {
    inodeID                 [1] IMPLICIT FOTPInodeID,
    offset                  [2] IMPLICIT UInt8,
    length                  [3] IMPLICIT UInt2,
    data                    [4] IMPLICIT BinString
}

FOTPWriteReply ::= SEQUENCE {
    status                  [1] IMPLICIT FOTPRetStatus,
    length                  [2] IMPLICIT UInt2 OPTIONAL
}

FOTPStat ::= SEQUENCE {
    inodeID                 [1] IMPLICIT FOTPInodeID
}

FOTPStatReply ::= SEQUENCE {
    status                  [1] IMPLICIT FOTPRetStatus,
    mode                    [2] IMPLICIT UInt2 OPTIONAL,
    uid                     [3] IMPLICIT UInt2 OPTIONAL,
    gid                     [4] IMPLICIT UInt2 OPTIONAL,
    size                    [5] IMPLICIT UInt8 OPTIONAL,
    accessTime              [6] IMPLICIT UInt4 OPTIONAL,
    creationTime            [7] IMPLICIT UInt4 OPTIONAL,
    modificationTime        [8] IMPLICIT UInt4 OPTIONAL,
    deletionTime            [9] IMPLICIT UInt4 OPTIONAL,
    linksCount              [10] IMPLICIT UInt2 OPTIONAL,
    fragsCount              [11] IMPLICIT UInt8 OPTIONAL,
    flags                   [12] IMPLICIT UInt4 OPTIONAL,
    version                 [13] IMPLICIT UInt4 OPTIONAL
}

FOTPSetParms ::= SEQUENCE {
```

```
    inodeID                 [1] IMPLICIT FOTPInodeID,
    mode                    [2] IMPLICIT UInt2 OPTIONAL,
    uid                     [3] IMPLICIT UInt2 OPTIONAL,
    gid                     [4] IMPLICIT UInt2 OPTIONAL,
    accessTime              [5] IMPLICIT UInt4 OPTIONAL,
    creationTime            [6] IMPLICIT UInt4 OPTIONAL,
    modificationTime        [7] IMPLICIT UInt4 OPTIONAL,
    deletionTime            [8] IMPLICIT UInt4 OPTIONAL,
    flags                   [9] IMPLICIT UInt4 OPTIONAL
}

FOTPSetParmsReply ::= SEQUENCE {
    status                  [1] IMPLICIT FOTPRetStatus
}


END -- End of Definitions

--
```

APPENDIX B

ADP Perl Module

```
#
# Perl ADP Module
#

package ADP;

require 5.002;
use FileHandle;
use Socket;
use Compress::Zlib;


# Debug Flag
$DEBUG = FALSE;

# Code/Data constants
$CODE_FD = 10;
$DATA_FD = 11;
@myCode;
@myData;

# ADP Network Port
$ADP_PORT = 20000;

# Header data
$EXEC_TYPE = 2;
$EXEC_TYPE_LEN = 1;

$CODE_SIZE_LEN = 4;
$DATA_SIZE_LEN = 4;

# Event Levels
$DEBUG = 1;
$INFO  = 2;
$ERROR = 3;
```

```
$CRITICAL = 4;

sub Event
{
    my( $level, $error ) = @_;

    if( $level == $DEBUG )
    {
        if( $DEBUG = TRUE )
        {
            print( STDERR "DEBUG: $error\n" );
        }
    }
    elsif( $level == $INFO )
    {
        print( STDERR "INFO: $error\n" );
    }
    elsif( $level == $ERROR )
    {
        print( STDERR "ERROR: $error\n" );
    }
    elsif( $level == $CRITICAL )
    {
        print( STDERR "CRITICAL: $error\n" );
        exit( 1 );
    }
    else
    {
        Event( $ERROR, "Agent: Unknown debug level $level\n" );
    }
}

sub Move
{
    local( $host ) = shift( @_ );
    local( $SockFD );

    if( !( local $iaddr   = inet_aton( $host ) ) )
    {
        Event( $ERROR, "Agent: Cannot lookup $host" );
```

```
    return FALSE;
}
local $paddr   = sockaddr_in( $ADP_PORT, $iaddr );
local $proto   = getprotobyname( 'tcp' );
if( !socket( SockFD, PF_INET, SOCK_STREAM, $proto )  )
{
    Event( $ERROR, "Agent: socket() failed: $!" );
    return FALSE;
}

if( connect( SockFD, $paddr ) == FALSE )
{
    Event( $ERROR, "Agent: connect() failed to $host: $!" );
    return FALSE;
}

Event( $INFO, "Agent: Connected to $host" );

# Send ExecType
Event( $DEBUG, "Agent: Sending agent type: $EXEC_TYPE" );
local( $execType ) = pack( "C", $EXEC_TYPE );
syswrite( SockFD, $execType, $EXEC_TYPE_LEN );

# Compress Code/Calculate Code lengths
local( $codeLength ) = 0;
for( local( $i ) = 0 ; $i <= $#myCode ; $i++ )
{
    $codeLength += length( $myCode[ $i ] );
}
local( $NBcodeLength ) = pack( "N", $codeLength );

local( $compCodeLength ) = 0;
local( $codeDefStream );
if( !( $codeDefStream = deflateInit() ) )
{
    Event( $ERROR, "Agent: Cannot create agent text deflation " .
           "stream" );
    return FALSE;
}
```

```
local( $compCodeOutput, $codeDefStatus );
local( $compCode );

local( $codeline );
foreach $codeline (@myCode)
{
    ( $compCodeOutput, $codeDefStatus ) =
        $codeDefStream->deflate( $codeline );

    if( $codeDefStatus != Z_OK )
    {
        Event( $ERROR, "Agent: Error compressing agent text" );
        return FALSE;
    }

    $compCode = "$compCode$compCodeOutput";
    $compCodeLength += length( $compCodeOutput );
}

( $compCodeOutput, $codeDefStatus ) = $codeDefStream->flush();

if( $codeDefStatus != Z_OK )
{
    Event( $ERROR, "Agent: Error compressing agent text" );
    return FALSE;
}

$compCode = "$compCode$compCodeOutput";
$compCodeLength += length( $compCodeOutput );

local( $NBcompCodeLength ) = pack( "N", $compCodeLength );

# Send Compressed Code Length
Event( $DEBUG, "Agent: Sending compressed agent text length: " .
               "$compCodeLength" );
syswrite( SockFD, $NBcompCodeLength, $CODE_SIZE_LEN );

# Send Uncompressed Code Length
Event( $DEBUG, "Agent: Sending uncompressed agent text " .
               "length: $codeLength" );
```

```
syswrite( SockFD, $NBcodeLength, $CODE_SIZE_LEN );


# Compress Data/Calculate Data lengths
local( $dataLength ) = 0;
local( $data );
for( local( $i ) = 0 ; $i <= $#myData ; $i++ )
{
    $data = "$myData[ $i ]\n$data";
}
$dataLength = length( $data );
local( $NBdataLength ) = pack( "N", $dataLength );

local( $compDataLength ) = 0;
local( $dataDefStream );
if( !( $dataDefStream = deflateInit() ) )
{
    Event( $ERROR, "Agent: Cannot create agent data " .
                   "deflation stream" );
    return FALSE;
}

local( $compDataOutput, $dataDefStatus );
local( $compData );

( $compDataOutput, $dataDefStatus ) =
    $dataDefStream->deflate( $data );

if( $dataDefStatus != Z_OK )
{
    Event( $ERROR, "Agent: Error compressing agent data" );
    return FALSE;
}

$compData = "$compDataOutput";
$compDataLength = length( $compDataOutput );

( $compDataOutput, $dataDefStatus ) = $dataDefStream->flush();

if( $dataDefStatus != Z_OK )
```

```
    {
        Event( $ERROR, "Agent: Error compressing agent data" );
        return FALSE;
    }

    $compData = "$compData$compDataOutput";
    $compDataLength += length( $compDataOutput );

    local( $NBcompDataLength ) = pack( "N", $compDataLength );

    # Send Compressed Data Length
    Event( $DEBUG, "Agent: Sending compressed agent data length: " .
                   "$compDataLength" );
    syswrite( SockFD, $NBcompDataLength, $DATA_SIZE_LEN );

    # Send Uncompressed Data Length
    Event( $DEBUG, "Agent: Sending uncompressed agent data length: " .
                   "$dataLength" );
    syswrite( SockFD, $NBdataLength, $DATA_SIZE_LEN );


    # Send Code
    print( SockFD $compCode );

    # Send Data
    print( SockFD $compData );

    # Close our connection
    close( SockFD );

    return TRUE;

}

sub Init
{
    local( $codeFD ) = new_from_fd FileHandle( $CODE_FD, "r" );
    local( $dataFD ) = new_from_fd FileHandle( $DATA_FD, "r" );

    @myCode = $codeFD->getlines();
```

```
    close( @codeFD );

    local( @data ) = $dataFD->getlines();

    local( $dataline );
    foreach $dataline ( @data )
    {
        unshift( @myData, $dataline );
    }
    close( @dataFD );
}
```

BIBLIOGRAPHY

[1] Anderson, Tom, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randy Wang. "Serverless Network File Systems." *15th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems.* 1995.

[2] Berkovits, S., J. Guttman, and V. Swarup. "Authentication for Mobile Agents." *Mobile Agents and Security LNCS 1419.* 1998.

[3] Bhattacharjee, Samrat, Kenneth L. Calvert, and Ellen W. Zegura. "An Architecture for Active Networking." Submitted to *IEEE Infocom '97.* 1997.

[4] Bray, Tim, Jean Paoli, and C. M. Sperberg-McQueen. "Extensible Markup Language (XML) 1.0." *W3C Recommendation.* 1998.

[5] Brenner, Walter, Rudiger Zarnekow, and Hartmut Wittig. Intelligent Software Agents: Foundations and Applications. Berlin: Springer-Verlag. 1998, 22-28.

[6] CCITT. "Data Communications Networks Open Systems Interconnection (OSI) Model and Notation, Service Definition: Recommendation X.208, Specification of Abstract Syntax Notation One (ASN1)." CCITT Blue Book. 1989.

[7] CCITT. "Data Communications Networks Open Systems Interconnection (OSI) Model and Notation, Service Definition: Recommendation X.209, Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN1)." CCITT Blue Book. 1990.

[8] CCITT. "Introduction to CCITT Signalling System No. 7: Recommendation Q.700." Fascicle V.17. 1988.

[9] Deutsch, P., and J-L Gailly. "ZLIB Compressed Data Format Specification version 3.3." *Network Working Group, Request for Comments: 1950.* 1996.

[10] Elbert, Stephen, Quinn Snell, Armin Mikler, Guy Helmer, Chris Csanady, Kim Stearns, Brian MacLeod, Matt Johnson, Bryan Osborn, and Iain Verigin. "Gigabit Ethernet and Low Cost Supercomputing." *Ames Laboratory.* 1997.

[11] Gosling, James, and Henry McGilton. "The Java Language Environment: A White Paper." *Sun Microsystems, Inc.* 1996.

[12] Geier, Martin, and Franz J. Hauck. "Scalable Migration for Mobile Agents." *5th Mobile Object Systems Workshop.* 1999.

[13] Hopper, S. A., A. R. Mikler, and J. Mayes. "The Design and Implementation of a Distributed Agent Delivery System." Submitted to *Workshop on Distributed Computing on the Web.* 2000.

[14] Hopper, S. A., A. R. Mikler, H. Unger, P. Tarau, and F. Chen. "Mobile Agent-Based File System for the WOS: An Overview." *1999 Advanced Simulation Technologies Conference*. 1999.

[15] Karimi, Jahangir. "A Software Design Technique for Client-Server Applications." *Concurrency: Practive and Experience*. Vol. 11(1). 1999.

[16] Martin, Bruce, and Jano Bashar. "WAP Binary XML Content Format." *W3C NOTE 24*. 1999.

[17] Maes, Pattie. "Modeling Adaptive Autonomous Agents." *Artificial Life Journal*. Vol. 1, No. 1 & 2. MIT Press, 1994.

[18] McKusick, Marshall, William N. Jayt, Samuel J. Leffert, and Robert S. Fabry. "A Fast File System for UNIX." *ACM Transactions on Computer Systems*. 1984.

[19] Meyer, D. "Administratively Scoped IP Multicast." *Network Working Group, Request for Comments: 2365*. 1998.

[20] Mikler, Armin R., and Kaja Abbas. "High Precision Knowledge Acquisition Model using Intelligent Agents." *1999 Advanced Simulation Technologies Conference*. 1999.

[21] Morris, James H., Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. "Andrew: a distributed personal computing environment." *Communications of the ACM*, No. 3. 1986.

[22] Oliveira, Luiz A. G., Paulo C. Oliveira, and Eleri Cardozo. "An Agent-Based Approach for Quality of Service Negotiation and Management in Distributed Multimedia Systems." *Mobile Agents: First International Workshop, MA '97*. Berlin, 1997.

[23] O'Reilly, Tim, Cameron Laird, Larry Wall, and Nathan Torkington. "Perl: a technology white paper." *O'Reilly and Associates*.

[24] Pawlowski, Brian, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. "NFS Version 3 Design and Implementation." *Proceedings of the Summer USENIX Conference*. 1994.

[25] Rosenblum, Mendel, and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System." *ACM Transactions on Computer Systems*. 1992.

[26] Satyanarayanan, Mahadev. "Scalable, Secure, and Highly Available Distributed File Access." *IEEE Computer*. Vol. 23, No. 5. 1990.

[27] Schwartz, Beverly, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partridge. "Smart Packets for Active Networks." *Openarch*. 1999.

[28] Srinivasan, R. "RPC: Remote Procedure Call Protocol Specification Version 2." *Network Working Group, Request for Comments 1831*. 1995.

[29] Sterbenz, James P. G. "Protocols for High Speed Networks: Life After ATM?" *Protocols for High Speed Networks IV*. London, 1995.

[30] Tarau, Paul, and Veronica Dahl. "Mobile Threads through First Order Continuations." *In Proceedings of APPAI-GULP-PRODE'98*. Coruna, Spain, 1998.

[31] Tennenhouse, D. L., and D. J. Wetherall. "Towards and Active Network Architecture." *Multimedia Computing and Networking '96*. 1996.

[32] Wang, Randolph Y., and Thomas E. Anderson. "xFS: A Wide Area Mass Storage File System." 1993.

[33] Wetherall, David J., and David L. Tennenhouse. "The Active IP Option." *Proceedings of the 7th ACM SIGOPS European Workshop*. Connemara, Ireland, 1996.

[34] Wooldridge, M., and N. R. Jennings. "Pitfalls of Agent-Oriented Development." *Proceedings of the Autonomous Agents '98 Conference* Minneapolis, 1998.