Microprocessing and Microprogramming 38 (1993) 669-678
North-Holland

Modeling and design of the multimedia subsystem of a distributed authoring environment

A. Kameas[a] S. Papadimitriou[b] and P. Pintelas[c]

[a]Department of Computer Engineering & Informatics, University of Patras, 26110 Patras, Greece

[b]Dept. of Computer Engineering & Informatics, University of Patras, 26110 Patras, Greece

[c]Dept. of Mathematics, Sector of Computational Mathematics & Informatics, University of Patras, 26110 Patras, Greece

In this paper, the model and architecture of the multimedia subsystem of a distributed authoring environment called VAS (Virtual Authoring System) are presented. VAS is an authoring environment that integrates various tools and provides a common communication basis for the tools and the designers who use them. Its underlying model is based on existing actor models, implementing all the society-modeling capabilities these models exhibit, while the Client Facilitator Consultant model is used to model the systemwise tool behaviour. The Multimedia Distributed Filing System (MDFS) of VAS meets many requirements users have from multimedia systems and uses VAS model to provide system users with transparent access to multimedia objects. MDFS consists of many co-operating servers, which implement VAS actor-based model. To take advantage of many well-established standards, MDFS communications have been implemented upon the Berkeley Unix socket library. The distributed nature of VAS offers a new perspective to the authoring process and the system's object-oriented model has led to an open-ended architecture, which permits the integration of new tools and supports the system's evolution to meet new emerging requirements.

1. Introduction

VAS is not an authoring system in the classical sense. It may rather be described as a society of tools cooperating towards courseware production (VAS does not exist as a system - hence the term "virtual" - but is composed of autonomous, communicating tools which may be distributed over a network), leading to a concurrent execution of the phases of courseware development; still this process will appear integrated and continuous to developers. To model systemwise tool behaviour, we use a model based on an analogy of human collaborative efforts, namely the *Client - Facilitator - Consultant (CFC)* model [1].

In this paper, we concentrate on the multimedia subsystem of VAS, and especially on the Multimedia Distributed Filing System (MDFS), [10] which in fact consists of many cooperating servers. All tools communicating with this subsystem are modeled as clients to the multimedia servers, which are the consultants of the system. There exist however certain servers which provide meta-services these are the facilitators of MDFS (figure 2). The general idea behind this scheme is to take advantage of the aggregate storage and computational capabilities of the distributed system.

Multimedia objects are fundamentally different from the classical text data and database records, and impose heavy performance requirements on any underlying system. Among others, these include the need to cope with the peculiarities of each media type, the provision of a flexible and efficient scheme in order to meet very large storage space requirements, and the support of synchronization structures in order to meet real-time constraints that result from continuous media streams and associated operations (like recording and retrieval of continuous media streams, which may in fact be of different type).

While many multimedia filing systems have been implemented [12, 5] attempting to meet these requirements, most of them are centralized, non-extensible designs with restricted transparency, extensibility, availability, reliability or performance. Many such systems comprise a set of tools built around a centralized media objects storage [4], with attention drawn to the types and attributes of stored objects rather than to physical access issues (which are delegated to the "underlying network"). In designing VAS we adopted many solutions used in MUSE system [6] (like usage of Unix environment, creation of multiple server processes, modularity in design etc).

MDFS is *extensible and capable of incremental growth*, so that new media servers, client applications and machines may be added or removed at any time, permiting VAS not only to improve the quantity of services offered (e.g. addition of new tools), but also to evolve in quality (e.g. definition of new roles). The modular design of MDFS also permits the modification of the servers in order to meet the particular requirements of the media each supports, while the CFC model also permits tools to dynamically change their roles in VAS society (facilitators), contrasted with traditional client–server models [3]. The real time constraints imposed by continuous media are confronted by using buffering at the client side and by supporting a flexible communication scheme that permits critical messages to gain priority of transmission. Finally, an online performance monitor is provided, which can detect heavily–loaded nodes and provide a detailed view of each node workload.

To achieve efficient usage of expensive multimedia machines and to meet storage requirements, VAS adopts a distributed storage scheme for media objects. This distributed architecture allows for data replication and built–in redundancy in all the critical resources that may fail. Thus the system can overcome arbitrary single–point failures and offers a high degree of availability. Communication between servers is based on the widely available Berkeley UNIX socket library. The Internet Domain socket type permits the servers to communicate uniformly independently of whether or not they reside on the same machine, achieving *location independence*. Using this library also promotes *portability*: MDFS servers can run on any UNIX machine, and with minor modifications on non–UNIX machines, provided these support socket–based interprocess communication. In addition, VAS supports both synchronous and asynchronous communication modes using special tools that act as "facilitators".

Robustness is achieved by making servers stateless, since the response of a stateless server does not depend in any way on the history of previous requests. In order to make a server stateless, we have avoided using descriptor based addressing and maintaining position and current directory information. Instead, to make the system *efficient*, we used a combination of caching and buffering techniques. Caching exploits the locality of program behaviour: each server caches the descriptors of the most frequently used objects. The cache replacement policy currently in use is LRU (Least Recently Used), which exploits the locality usually observed in most program access patterns (for example, a server that uses LRU reallocates the cache resources used by a client that failed). Buffering takes advantage of the fact that most requests handle information sequentially.

In the next section, the architecture and underlying model of VAS are presented. We then focus on MDFS and present its architectural design and most important characteristics. Subsequently we present some implementation issues that arise from the distributed nature of the system and the solutions we adopted. Finally we conclude with the presentation of future research directions. The reader should note that technical solutions to design issues, where provided, are not described extensively due to space limitations (to be frank, we believe that each solution to a design issue would require a separate publication in order to be properly described).

2. VAS model and architecture

VAS underlying model is heavily based on existing actor models (i.e. [2]). An actor system is composed of *actors* that communicate with each other by exchanging *messages*. Everything in the system, at any level of granularity, is represented as actors capable of receiving and responding to messages. Messages are themselves actors, too.

To envisage VAS architecture, one must think of a network of interconnected tools that interact continuously with each other and with the system environment using links [8, 9]. Both tools and links are actors. To make, however, the modeling and analysis of VAS easier, we have grouped the tools it incorporates into five classes (subsystems) [9]: the computational subsystem, the expert subsystem, the multimedia subsystem, the reusability subsystem, and the user interface subsystem.

A tool is an autonomous, integrated computational unit that produces a certain element of the course (e.g. a frame), or offers a service in the authoring process. The internal tool structure is made up of a behavioural and a functional part (figure 1.a). The *behavioural part* incorporates the functional constraints of the tool: what kinds of input (commands and data) it responds to, which are the other tools it may communicate with, what happens to the inadequate messages it receives, etc. The *functional part* is the pure computational part: it is composed of the procedures and functions the tool uses to transform tool input into output. Links are used to represent the tool's communication environment and facilitate the transportation of messages. The link notion descended from dataflow models; only in our model, links are active entities represented with actors. The set of links forms the interfacing part of a tool. A link's *behavioural part* contains the physical addresses of tools together with type matching information, while in its *functional part*, the message transportation and transformation functions are included. Messages being themselves actors, have a *behavioural part* that may be interpreted as a header and a *functional part* that activates the appropriate handler.

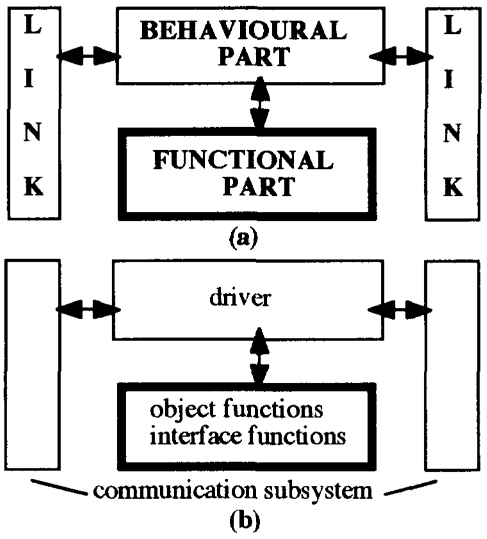


Figure 1. VAS model of actors and links, and its incorporation in MDFS

3. MDFS society of tools

In this section we present the tools of the multimedia subsystem of VAS and describe the role each has within VAS society.

3.1. The facilitators

This class of tools provides services (e.g. control and meta-information) to consultants or clients. Tools functioning as facilitators exist at node level (e.g. the daemon that handles node communication) as well as at system level (e.g. the portmapper server). The addresses of facilitators are well-known addresses; well-known addresses are assumed to be inherently known to all tools.

3.1.1. The portmapper server

This server is needed because services offered by MDFS may bind to different ports each time the corresponding tools are activated as well as when they are executed on different machines. The portmapper server contains information on the logical-physical address correspondence of these services, so that these may be accessed by any tool through this server. Servers bind a port dynamically and inform the portmapper server about this binding.

The adopted solution of using a portmapper server provides great flexibility since every new service becomes available systemwide once it identifies itself to the portmapper. Thus, based on VAS model, MDFS demonstrates an extensible architecture that enable new services to be added to the system by application programmer-

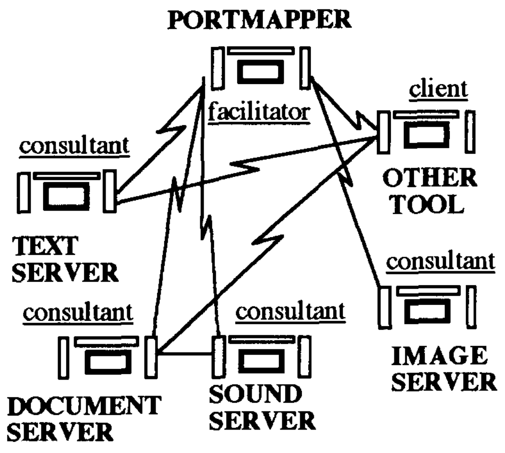


Figure 2. The structure of VAS society of MDFS tools

s and developers. The alternative of maintaining the correspondence between remote services and ports in a file duplicated on each node has the disadvantage of requiring the updating of all those files whenever new remote services are introduced.

3.1.2. The internode communication daemon

Communication between servers residing on different nodes is accomplished by using a special daemon process at every node. Every tool that needs to communicate with a remote tool, sends its message to the local communication daemon; the message is enqueued in the node outgoing queue, and then the daemon transmits it to the remote site using a well-established procedure [10]. Thus, the daemon acts as a node facilitator which provides logical-physical address mapping for all system nodes. The main advantage of this design is that it permits *asynchronous communication* to take place. It also enables the designer to take advantage of a message priority scheme that would apply to all messages leaving the node, independent of the priorities of the tools that produced them. Note that the daemon is used only for communication with a tool residing on a different node. For communication with tools of the same node, the tool links are sufficient.

3.2. The consultants: the structure of VAS MDFS servers

The structure of each MDFS server follows the actor-link model of VAS (figure 1.b). The server's links are implemented within the *communication subsystem*, while the server's behavioural part contains the *driver*, which is the uppermost operation layer. The server's functional part contains the *interface functions* and the *object functions*. The former receive the appropriate parameters from the behavioural part, handle the details of the caching and buffering subsystem using appropriate object handlers and finally call the object functions, which are low-level implementation functions that perform operations specific to the media objects that the server handles.

The driver and the communication subsystem, which are both presented in the rest of this subsection, have similar structure for all the MDFS servers independently of the media on which they operate. In contrast, the remaining components (especially the object functions) depend heavily on the object's properties (e.g. real-time constraints for the playback of digital audio/video); a discussion of these would introduce technical details and is beyond the scope of this paper.

3.2.1. The driver

The driver starts by initializing parts of the communication subsystem of the server (e.g. it restores the cache contents saved after the last activation of the tool). Then the entire caching subsystem is initialized, which as we have described is used to speed-up execution. Also buffers for the buffering subsystem are allocated. These buffers are of great importance to continuous media servers, since they can be used to reduce media playback variations. After performing those initialization chores the driver creates a child process that inherits all the environment (descriptors, priority, values of variables etc.) of the parent process except that it runs in background. The background execution of the server is possible since the operating system shell which creates and executes all the user entered commands waits only for those commands to exit and not for their children. In this way all the servers that constitute the MDFS can be spawned without requiring from the user an explicit specification of the background processing. The parent process exits and closes automatically any open descriptors.

The child process continues its operation by waiting to accept requests on its socket. Each request specifies the request type in one of its header fields. The other fields are dependent on the particular request. Both the server and its clients know the format of the request and reply.

In addition, the driver performs two tasks by calling the request type-specific handler: it retrieves parameters from packets, and it calls the interface functions which are responsible for cache management and for preparation of the appropriate calls to the object functions

Finally, the driver queues all the correct incoming messages before passing them to the functional part. The three phase message processing scheme adopted in VAS (tool interface - behavioural part - functional part) enables the pipelining of the process and consequently speeds up execution (by increasing the degree of multiprogramming on a system that runs many VAS servers). Note that the driver, if needed, can adapt incoming messages to a format accepted by the interface and the object functions of the tool (e.g. transform the format of a picture message).

3.2.2. The tool communication subsystem

The communication subsystem of MDFS encapsulates all the communication protocol-related functionality. It is responsible for correctly accepting and transmitting messages. Furthermore, its performance is enhanced by caching the logical-physical address mapping of the most frequently used destination tools. It also performs network-to-host conversion of incoming packets (e.g. reassembly of original incoming messages out of network packets) and validity checks to detect erroneous transmissions. Each MDFS server keeps a table mapping the request types to request handlers. Each request handler is called automatically on the arrival of a request packet from the client; the packet specifies the request type in its header. This design falitates the extension of the server by adding new request types along with their corresponding handlers. Messages are then passed to the behavioural part of the tool. As far as outgoing messages are concerned, the tool communication subsystem is responsible for forming packets acceptable by the network, and for calling the appropriate routine for packet transmission.

4. Distribution of multimedia objects

The design of MDFS provides a totally transparent access to the stored information by adapting the developed technology for distributed file systems to the particular needs of the stored multimedia objects (e.g. images, voice ropes, text).

VAS application developers that will use MDFS as the underlying support for the development of more complex applications must be provided with simple operations on the basic types of objects (e.g. images, voice ropes, text strings etc.) that act the same independently of where the object resides. Clearly, there exists a performance penalty in accessing the remote objects, but it must be kept within affordable limits. This is the well known principle of *access transparency* [14].

In order to obtain efficient distribution of multimedia objects, we use a distributed file system tree that links these objects, a naming mechanism that translates the uniform naming scheme to the node-specific one and caching and buffering techniques at the client's site.

4.1. Distribution of the file system tree

The basic mechanism that supports the distribution of multimedia objects in MDFS is the distribution of the file system tree. In this section, the terms file and object are used interchangeably. An object only resides on the storage resources of a single node of the network.

MDFS divides the accessible name space from each node into two parts:

- the local namespace
- the shared namespace

We examine each of these parts in turn and then we investigate load-balancing issues of the shared namespace.

4.1.1. The local namespace

The local namespace is used to keep private files and temporary, intermediate results of the various operations. It is important to localize intermediate operations in the production of VAS frames in order to be able to add client tools without increasing significantly the load at the servers. For example, the voice rope editing operations of a VAS voice editor tool create many temporary voice ropes that do not concern a remote voice server. This local storage system can either use the underlying operations of the local file system or build new file system structures in order to cope effectively with the pecularities of the stored media (e.g. real–time constraints of continuous media, large sizes of images). The former approach is more suitable for prototyping since it has the advantages of simplicity and requires little additional code to be added. In the current design, MDFS uses ordinary UNIX (specifically SunOs 4.0) files for the storage of both the local and the remote files. The improved redesign of the UNIX filesystem introduced with 4.3 BSD distribution [11] seems to be quite adequate to satisfy the demands of most multimedia applications. Clearly, applications that handle heavy loads of requests for continuous media must follow the latter approach by using specialized storage and communication structures to cope with the real–time demands of those requests (for more details about how these requirements may be technically solved the reader is refered to [15]).

4.1.2. The shared namespace

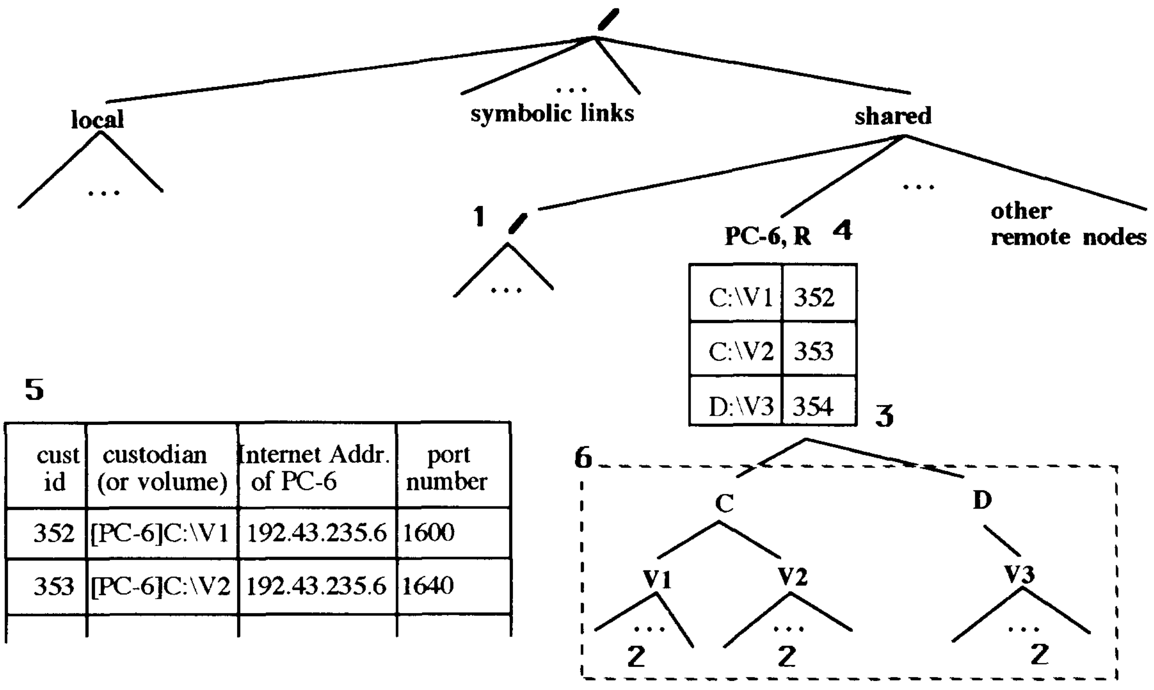
Symbolic links can be specified in order to access commonly used remote objects more conveniently and efficiently. The symbolic link acts as a direct pointer to the underlying object. For example to identify a remote file of a UNIX file system a symbolic link must specify both the machine identification and the corresponding i-node number. In this way a full search of the shared namespace for the file is avoided. This optimization is heavily dependent on the file system where the remote object resides. A file system that offers low-level identifiers (e.g. i-nodes) for the stored objects allows for more efficient remote access than one in which the only way to access an object is the full path-name translation, because the symbolic links exploit the low-level identifiers in order to refer directly to the remote object.

Direct accessibility of objects within each node is restricted to the shared objects that reside on it. Thus, in a large distributed system a very small fraction of the shared namespace is maintained within each node. Remote objects and directories are replaced by stubs (point 3 in figure 3 shows two stubs, one for each custodian, for PC-6 MS-DOS machine; devices C and D denote hard disk partitions), which indicate where to continue the search for an object. More specifically they identify the *custodian* [13] for that object. The custodian is a server process which is responsible for keeping and optimizing the storage for all the objects which form a subtree in the file system tree representation and for servicing the requests for those objects (more like a server facilitator). The custodian organisation of the shared namespace is valuable in achieving expandability and load balancing, while avoiding bottlenecks, and results in a system practically of unlimited growth, as will become more apparent from the discussion that follows.

A *custodian location table (CSL)* that maps the custodians to their Internet addresses is replicated at every node. The system performs transparent remote access when it falls upon stub points by looking up this map. More specifically, the stubs keep only the custodian identification. The actual address is retrieved from the custodian location map. This permits changing the location of a custodian without affecting the stub entries of the object system trees. Only the map replicas need to be updated (point 5 in figure 3 shows two example custodian location table entries).

4.1.3. Redistribution and Load balancing

Since large multimedia applications require enormous amounts of storage and computing power it would be better if they were distributed over many machines. In order to obtain the benefits of distribution it is necessary to have a mechanism that maintains the load balanced over the network. Thus, the current UNIX–based design of MDFS is supported by an on-line performance monitor. This monitor collects performance da-



C:\V1	352
C:\V2	353
D:\V3	354

cust id	custodian (or volume)	Internet Addr. of PC-6	port number
352	[PC-6]C:\V1	192.43.235.6	1600
353	[PC-6]C:\V2	192.43.235.6	1640

Figure 3. The distributed file system tree as seen by a UNIX node (e.g. "naxos5")

ta both at the individual node level and at the custodian level. The collected data can be used by off-line tools which after performing statistical analysis on them can suggest the best possible (probabilistically) actions in order to redistribute the load uniformly. Such a redistribution is a very heavyweight operation; that is why it must be performed off-line. The monitor design and implementation in a UNIX LAN environment will be reported in a future work.

In order to redistribute multimedia objects we rely on the custodian organisation of the object system tree. A collection of objects that is managed by the same custodian is referred to as a *volume*. Clearly, a volume constitutes a subtree at the tree representation of the MDFS name space. In the current design there exists a one-to-one mapping between custodians and their volumes. The organization into custodians permits the usage of different indexing and data storage structures for each volume since these details are encapsulated in the volume's custodian. This is extremely important since the system must operate on the various types of multimedia data which have diverse properties and thus call for the use of specialized structures to cope with the particular needs of each. It is also important to observe that the size of CSL grows linearly with the number of custodians. This, combined with the fact that every custodian can handle a very large number of objects (e.g. all the course frames of one WORM drive) allows VAS to keep huge amounts of information with very little overhead at each node.

As a specific example consider the case where each workstation is equipped with 500Mb local storage. An entry in the custodian-location map needs at least 6 bytes (4-byte Internet address plus 2-byte port number). By letting the custodian identifier be 2 bytes, about 10 bytes for every entry are wasted. Then, assuming a network of 50 nodes, a space of 0.5Kb for the volume location map at every node yields access to $500 \cdot 50 = 25$ Gb. With a course frame of approximately 1Mb we can have courses consisting of 25000 frames. The total wasted space is 0.5K at each of the 50 network nodes, thus only 25Kb.

The redistribution of objects can be performed both by object movement, where an object moves between volumes, and by volume movement, where a volume moves between nodes. We are in the process of performing extensive simulation experiments concerning the performance of the volume movement algorithm. This algorithm is as follows:

1. keep a flag copy-on-write initialised to FALSE
2. construct a machine independent representation of the volume
3. ship that representation to the remote site
4. regenerate the multimedia objects at the remote site
5. if copy-on-write changes to TRUE (writes on volume's objects while copy was executed, dirty volume) then repeat from step 1, retransmitting only the changed objects. The interarrival time of the write requests to the volume's objects is the critical factor in determining volume dirtyness.
6. after the second (possible) update, freeze the volume. For all the services offered by the moved volume inform the portmapper for their change. Also, broadcast a message in order to update the volume location map of each node by recording the new position of the moved volume.

A similar algorithm was used at the migration process of files in the distributed file system described in [13] but little work has been done on the important topic of the migration of multimedia objects.

4.2. Naming

MDFS adopts a uniform naming scheme for its objects. This means that independently of where an object resides and what it represents (e.g. text, image) it is accessed using the same naming scheme. We chose to adopt the UNIX tree-structured file system naming scheme since it is flexible and widely used. The naming mechanism performs the task of mapping a path name that uses the MDFS conventions (which are UNIX-like) to the appropriate name of the Local Multimedia File System (LMFS). Under the root of the MDFS tree the directories */local*, which contains the local namespace, and */shared* are positioned. The only files under the root are the symbolic links to the more frequently used remote files. Directly under the */shared* directory the locally stored part of the shared namespace (local file system) is mounted. For example, at a UNIX workstation named "naxos5" the local UNIX filesystem is mounted as indicated in figure 3 (point 1), under the */shared* directory there exists one stub for each custodian (point 3, same figure), while a flag R (point 4, same figure) denotes that machine PC-6 is remote.

An MS-DOS image object named X-Ray10 residing at the MS-DOS directory *C:/ radiology/ xrays* at machine PC-6 (point 2, same figure) will translate to an MDFS path name */PC-6/C/ radiology/ xrays/ X-Ray10.* The MDFS design approach consists of using prefix patterns and specifying replacements [3]. A prefix pattern is one that begins from the root (e.g. */PC-6/C/ radiology/ xrays* is a prefix pattern while the above pattern as subpattern in */PC-5/D/PC-6/C/ radiology/ xrays* is not and it cannot be replaced).

As an example of the functionality of the naming mechanism consider a course frame named *Athens* with full local pathname on a DOS machine *C:/ Geography/ cities/ Athens.* The syntactic namespace software translates the uniform pattern used by MDFS (i.e. */local/ Geography/ cities/ Athens*) to the locally understandable one (i.e. the above DOS pathname).

4.3. Caching and Buffering

In order to obtain efficiency an additional caching scheme is used at the client's site. This caching scheme is particularly adequate for large objects like course frames. A course frame is read in its entirety in one operation and is cached at the client's site. Since most course frames are read-only their access is particularly efficient since no call-backs are required. The call-back mechanism [13] provides the server with a way to inform the clients of a shared object that some operation has changed the object. Thus, the clients are notified that the object that they have cached locally is not the most current.

In order to support continuous media as voice it is necessary to provide an effective buffering scheme. Buffering can be provided at the server's site or at the client's or at both. The MDFS design uses buffering at the server site in order to simplify the development of client applications. The server through the read-ahead that performs with its buffering subsystem tries to guarantee a media stream to the client that satisfies the continuity requirements [12]. Optionally, the client can use additional buffering in order to further smooth the variations.

5. Conclusions

We have presented the design and architecure of the multimedia subsystem (MDFS) of a distributed authoring environment (VAS). We have addressed a number of design issues that come up in multimedia filing systems (it is not technically feasible to address all the important design issues of such systems in a work of this kind), and have shown that this design meets many of the performance and functionality requirements that users have of such tools. This is a result of both the underlying model that VAS supports (based on the notion of actors) and the implementation decisions we made along the way.

By using this model in the MDFS design, we have constructed a robust and efficient distributed system that may evolve in time with the addition of new services. Moreover, CFC model supports the dynamic nature of VAS by enabling certain tools (i.e. facilitators) to "dynamically" change their roles in VAS society. In order to implement a prototype of MDFS, we have used the Berkeley UNIX socket library, creating a system that may be distributed over a network as common as Internet (non–UNIX nodes that support sockets can also access MDFS). Other important design desicions include the distribution of the file system tree together with the use of a uniform naming scheme for media objects to achieve location transparancy, the support of caching and buffering at each client site together with the use of pipelining in message processing by tools to speed up execution and to handle continuous media objects, the use of custodians to support distributed media storage and media migration, and the use of an on–line performance monitor for load balancing and redistribution.

VAS design takes into account the different classes of users that may use the system. A coarse–grain classification distinguishes three such classes: end–users (VAS application users), system users (VAS application developers) and system administrators. Although in many cases these classes may overlap, care has been taken to define roles in the process and to represent each role with an autonomous tool. In this way, users are forced to assume different roles for different tasks and thus become part of the system model.

We are currently finishing with the implementation of basic MDFS functions, and we are developing a special database server that offers database services using the same model as in MDFS. In the near future we plan to conduct simulation studies on the performance of the system using mathematical models and simulation tools [7]. These will enable us to relate the implementation of MDFS to the network performance (which is critical parameter for system performance) more tightly, and hopefully to come to new conclusions about the "dynamics" of such systems. Furthermore, we will concern ourselves with issues related to object specification, object retrieval and system usability.

REFERENCES

1. S.S. Adams and A.K. Nabi, *Neural Agents - A Frame of Mind.* Proceedings of the OOPSLA-89 Conference, New Orleans, USA, Oct. 1-6, 1989, pp 139-149.
2. G. Agha and C. Hewitt, *Concurrent Programming Using Actors.* In Object-Oriented Concurrent Programming (A. Yozenawa and M. Tokoto eds), The MIT Press, 1987, pp 37-53.

3. D. Commer, *Operating Systems Design Vol. II.* Prentice–Hall, 1987.
4. M. Derks and W. Bulthuis, *A framework for authoring tool integration.* In Learning Technology in the European Communities (S.A. Cerri and J. Whiting eds), Kluwer Academic Publishers, 1992, pp 549-563.
5. J. Gait, *The optical file cabinet: A random access file system for Write–Once optical disks.* IEEE Computer, 21(6), 1988, pp 11-22.
6. S. Gibbs, D. Tsichritzis, A. Fitas, D. Konstantas and Y. Yeorgaroudakis, *MUSE: A multimedia filing system.* IEEE Software, 4(2), 1987, pp 4-15.
7. A. Javor, *An AI supported tool for simulation in informatics.* Syst. Anal. Model. Simul. 8, 1991.
8. A. Kameas, S. Papadimitriou and P. Pintelas, *Modeling an authoring environment.* Presented at the Systems Simulation and Scientific Modeling conference run by the Chinese Association of Systems Simulation, Beijing, Peoples Republic of China, Oct 20-23, 1992.
9. A. Kameas, S. Papadimitriou and P.Pintelas, *Modeling and design of a Virtual Authoring Environment.* Technical Report TR 93.04.17, Computer Technology Institute, 3 Kolokotroni st., 26221 Patras, Greece, 1993.
10. A. Kameas, S. Papadimitriou and P.Pintelas, *The modeling and performance tuning of the Distributed Multimedia Filing System of a Virtual Authoring Environment.* Technical Report TR 93.05.18, Computer Technology Institute, 3 Kolokotroni st., 26221 Patras, Greece, 1993.
11. S.J. Leffler, M.K. McKusick, M.J. Karels and J.S. Quarterman, *The design and implementation of the UNIX 4.3BSD operating system.* Addison–Wesley, 1989.
12. P. Venkat Rangan and H.M. Vin, *Designing file systems for digital video and audio.* Proceedings of the 13th ACM Symposium on Operating Systems Principles, Pacific Grove, USA, Oct 13-16, 1991, pp 81-94.
13. M. Satyanarayanan, *Distributed file systems.* In Distributed Systems (S. Mullender ed), ACM Press, 1991, pp 149-188.
14. P.K. Sinha, M. Maekawa, K. Shimizu, X. Jia, H. Ashihara, N. Utsunomiya, K.S. Park and H. Nakamo, *The Galaxy distributed operating system.* IEEE Computer, 24(8), 1991, pp 34-41.
15. D.B. Terry and D.C. Swinehart, *Managing stored voice in the Etherphone System.* ACM Trans. on Comp. Systems, 6(1), 1988, pp 3-27.