DynaVideo - A Dynamic Video Distribution Service

Luiz Eduardo Leite, Renata Alves, Guido Lemos, and Thais Batista

Informatics Department - DIMAp
Federal University of Rio Grande do Norte - UFRN
{leduardo, renata, guido, thais}@natalnet.br
http://www.natalnet.br

Abstract. Most solutions proposed to implement audio and video distribution services have been designed for specific infrastructures or have been tailored to specific application requirements, such as stream and clients types which will be supported by the service. Other important aspect in this context is that the performance of distributed services is becoming increasingly variable due to changing load patterns and user mobility. This paper presents the Dynamic Video Distribution Service - DynaVideo. The service may be designed to distribute video in a way that is independent of the video format and to interact with different types of clients. The main feature of DynaVideo is the ability to configure the service dynamically to a specific demand.

1 Introduction

The cost reduction of high-speed networks, the development of powerful and cheaper microprocessors and the consolidation of audio and video standards to applications such as Digital TV, Video on Demand, High Definition Television and Interactive TV are factors that motivate the development of video distribution services over digital networks, such as the Internet. This paper presents the Dynamic Video Distribution Service - DynaVideo. The service is designed to distribute video in a way that is independent of the video format and to interact with different types of client. The service may be used to distribute video over any digital network, however, it is focused on the Internet. The main feature of DynaVideo is the ability to configure the service dynamically to a specific demand. Applications that deal with video distribution, such as broadcast digital television, normally have to deal with abrupt variations on its demand. The number, type and location of clients of the service can vary rapidily. This could occur every time the broadcast of an interesting programme starts. Nowadays, many systems can distribute video over digital networks like the Internet. Real System of the Real Networks [5] encodes the video streams in many formats, but its focus is on the support of its proprietary format RM. This system may transmit video with IP Multicast, TCP, UDP or HTTP. The Microsoft Windows Media Service [7] proprietary format is ASF. The following video formats BMP, WAV,

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

WMA, WMV, ASF, AVI, and MPEG-1 [3] are also supported and can be transmitted with UDP, TCP, HTTP/TCP or IP Multicast. The IBM VideoCharger [6] transports MPEG-1, MPEG-2 [4], AVI, WAV, LBR and QuickTime video streams, using RTP [9], TCP, HTTP or IP Multicast. Once configured, distribution services based on these platforms remain unchanged and cannot automatically adjust configurations to varied demands. This paper is structured as follows: Section 2 presents the DynaVideo Architecture; Section 3 discusses the implementation details of the DynaVideo components; Section 4 presents some experiments results. Finally, Section 5 contains the conclusions.

2 Dynavideo Architecture

The DynaVideo service can be dynamically configured. This is its main feature. This flexibility allows the service to automatically adjust itself to demand variations. The service continually tries to find an optimized configuration to respond to a given demand. In DynaVideo the demand is defined by the number, type and location of the service clients. The target applications of the service are Digital Television broadcast and Video on Demand. In such applications, the demand can change from a few users to millions of them in a short time. Fig. 1 shows the DynaVideo architecture using the UML component diagram [2].

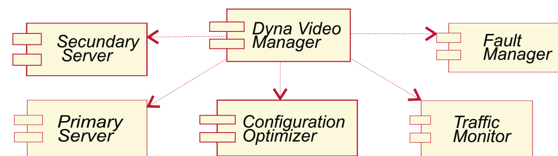

Fig. 1. DynaVideo Architecture

The *DynaVideo Manager (DM)* controls the service execution. When clients request connections, this module looks for a server with capacity to support them. If it finds one, *DM* associates the client with this server. Otherwise, a *Secondary Server* is initialized to support the client. Notice that initially the service policy is to try to serve the client as fast as possible, and not to find an optimized configuration to the video distribution.

The *Primary Servers (PS)* have direct access to video sources, which can be real time encoders or video file servers. *PS* captures a video stream from the source and transmits it to the service clients.

The *Secondary Servers (SS)* are different from *PS* because they do not have direct access to video sources. They receive the video stream from a *PS* and forward it to the service clients, acting as a reflector. The main feature of the DynaVideo *SS* is its ability to move through the network. This way, when it is necessary to optimize the service configuration, *DM* can determine that a certain *SS* moves to a given node of the network.

The arrival of a client determines a change in the service configuration. This event activates the *Traffic Monitor (TM)* [1] to find the routes from the active servers to the client. This way, the role of *TM* is to create and update a data structure, the *route graph*, which records routes from the active servers to the service clients.

When the *route graph* is updated, *DM* activates the *Configuration Optimizer (CO)* module to compute an optimal configuration of the service. *CO* is executed in the background, searching for a better configuration of the video distribution service, considering the current demand represented by the route graph. Once a configuration better than the existing one is found, *CO* requests *DM* to:

– move clients from a server to another one;
– add or delete *Secondary Servers*;
– move *Secondary Servers* from one locality to another.

The goal is to tune the service configuration to optimize the use of transmission, processing and storage resources. Fig. 2 illustrates an example of a reconfiguration, showing the evolution from a configuration with unnecessary streams traversing links and routers over the network to another where this does not occur. One *SS* is added and three clients are transferred from *PS* to *SS*. The other modification was the creation of a multicast group with three clients. This optimization takes into account that R2 does not support multicast, so the only way to eliminate unnecessary transmissions in the route that traverses R2 is to use a *SS*.

Finally, the *Fault Manager (FM)* goal is to identify failures in the service components and to arrange replacements through a service reconfiguration. For example, if a server fails, *FM* detects this event and asks *DM* to move its clients to other servers.

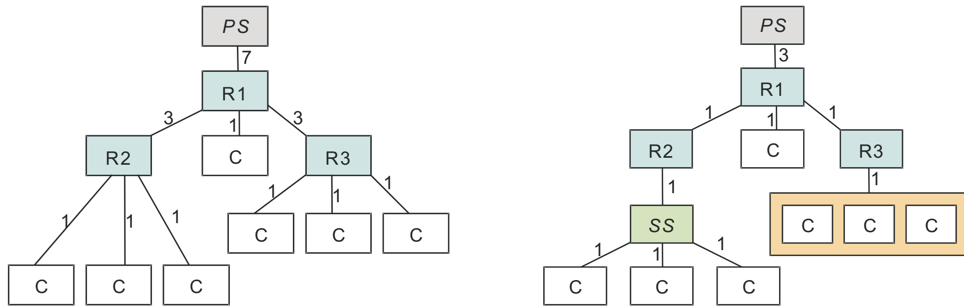

Fig. 2. DynaVideo Reconfiguration Example

In the following sections, we discuss the specification and implementation of *DM*, *PS*, and *SS* Modules. The *TM* module was described in [1]. *CO* and *FM* are under development.

3 Implementation Issues

3.1 DynaVideo Manager

The *DynaVideo Manager (DM)* provides mechanisms to:

- Register *Primary Servers*;
- Add and delete clients;
- Add, delete, and move *Secondary Servers*;
- Select a primary or secondary server to attend a client request.

DM is divided into three parts: *Server Interface*, *Client Interface* and *Manager Controller*. The *Server Interface* function is to control the communication between *DM* and the modules *SS* and *PS*. The *Client Interface* receives a connection request and forwards it to the proper server (primary or secondary). The *Manager Controller* enables service management by controlling which server is active and which server supports each client. This class updates the data structures *ServerTable*, *ClientTypeTable* and *ServiceConfiguration* through the execution of the routines *ServerRegistration()*, *AddClient()*, *AddServer()*, *PrimaryServerRegister()*, *DeleteClient()* and *MoveServer()*. The choice of the server to support a new client is made by the *SelectServer()* routine, which searches at the *ServiceConfiguration* tree (similar to the tree shown in Fig. 2) for a server with capacity to support the new client.

One of the target application of the DynaVideo service is to distribute Television video. Considering this application as an example, when a user wants to play a video using a third part player (DynaVideo client), he must do the following actions: the user must access a Web page and select a TV channel link. At the client browser, an applet will be executed to send a *StreamRequest()* to *DynaVideo Manager*, providing the proper information about the video player that will be used. *DM*, executing the *SelectServer()* method, searches for an appropriate server to support the client. If there is one, it is selected and *AddClient()* is executed to associate this server with the client. Then, the video stream begins to be transmitted to the client. The transmission ends when the client sends a *StopStream()* to *DM*. In response, *DM* removes the client from the service with the execution of the *RemoveClient()* method, as shown in Fig. 3. The same figure illustrates a situation in which there is not an available server to support a new client. In this case, *DM* initializes a *Secondary Server* to serve the new client.

3.2 Primary Server

The *Primary Server (PS)* can be configured to meet service needs (video format, transmission rate and protocol). *PS* supports different kinds of clients (for instance, Microsoft Media Player, Real Player and MTV). The interface between *PS* and the data sources is based in two classes: *SourceManager* and *SourceInterface*. These class specifications depend on the stream generator entity. If there

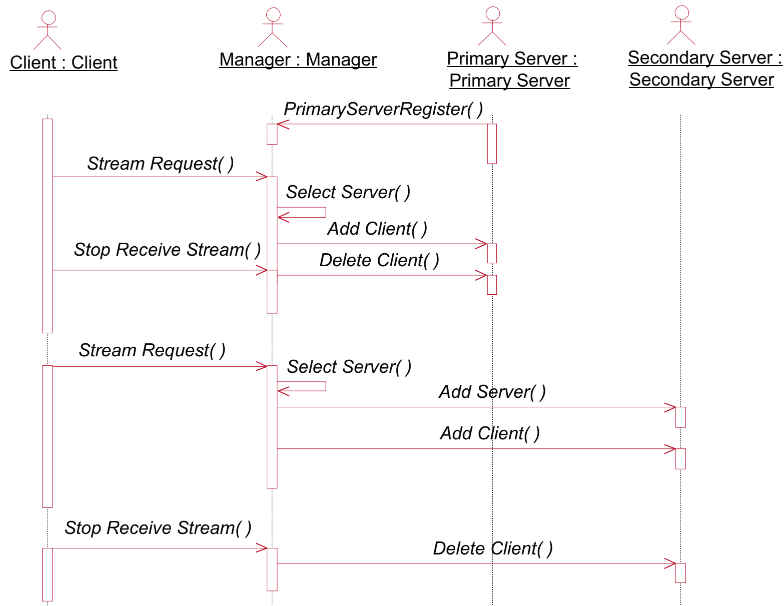

Fig. 3. Add Client Case Sequence Diagram

is more than one data source, an instance of each of these classes is used for each source. The *SourceInterface* class receives video data segments and stores them in an instance of the *DataBuffer* class, invoking the *PutDataOnBuffer()* method of the *MainManager* class. The *SourceManager* class acquires parameters of the data source. For example, if it is a TCP transmitter, the jitter is measured and reported to the *MainManager* class through the *SetParameter()* routine. The *MainManager* class controls the exchange of information between the other classes of *PS*. It controls data insertion and recovery in a *DataBuffer* object, using the methods *PutDataOnBuffer()* and *GetDataFromBuffer()*. This class also stores and retrieves the *PS* parameters configured by the service administrator or defined by some other object, through the *SetParameter()* and *GetParameter()* routines. The *MainManager* class interprets and executes commands through the *Execute()* routine and adds and removes clients using *AddClient()* and *DeleteClient()* routines.

The *DataBuffer* class contains, in addition to a buffer used to store the data stream, mechanisms that provide mutual exclusion between buffer accessing objects. This class has also a *timestamp* that indicates the time of data insertion into the buffer. This feature allows the classes which access objects to decide whether they will use those data or not. The interface between the service administrator and the *MainManager* class, which enables the system administration, is implemented by the *UserInterface* class. The *ManagerInterface* class acts

as an interface between *PS* and *DM*. It interprets commands from the *DM*, executes some, and forwards others to *SS*. Each user of *PS* must be associated with the classes *ClientManager* and *ClientInterface*. *ClientManager* controls each new client's connection without the intervention of the administrator, negotiating its initial parameters. It also notifies the connection or disconnection of clients to the *MainManager* class. The *ClientManager* class controls stream transmission to the clients and can renegotiate the initial transmission parameters. For example, it can change a client connection from point-to-point to multicast in real time. *ClientInterface* class controls access to data stored in the *DataBuffer* object, invoking the *GetDataFromBuffer()* method of the *MainManager* class. The *ClientInterface* class also adjusts the data stream format, in conformity with requirements of the client type associated with the class, and transmits the data. For instance, this class adapts MPEG streams to allow their transmission using the RTP protocol.

Primary Server was implemented to allow transmission of MPEG streams to the following clients: MTV (MPEG-1 using TCP, UDP, HTTP and IP Multicast); VideoLan (MPEG-2 using UDP); Windows Media Player (MPEG-1 and MPEG-2 using HTTP); RealAudio (MPEG-1 using HTTP) and JMF (MPEG-1 using RTP). In the experiments done at NatalNet Laboratory at UFRN, MPEG streams were obtained in real time from an Apollo capture card. The card is installed in a PC Pentium II 400 with 64 MB of RAM and a Microsoft Windows NT 4.0 operating system. A *Named Pipe* with an 8 MB buffer was created. The Apollo software was configured to write the video stream into the pipe. A program named *Streamer* was implemented using two *threads*. One thread waits for connection requests and accepts them if there is no other client (*PS*) connected. The other one continues to read the MPEG stream from the pipe and transmits it to *PS* using a TCP data connection. When the buffer of the *Named Pipe* becomes full, the *streamer* empties the buffer and notifies *PS* using a TCP signaling connection.

PS was implemented using C++ and was installed in a Linux platform. As already mentioned, *SourceInterface* class was implemented to receive the video stream. *SourceManager* class receives control data from the *Streamer*. *SourceInterface* object stores the data segments of the video stream in a *DataBuffer* object, using the *PutDataOnBuffer()* method of the *MainManager* class. *ClientManager* and *ClientInterface* classes were specified and implemented in a different way for each protocol supported by *PS*;

To transmit MPEG-1 streams using the TCP protocol, the *ClientManager* class implementation has a main method that remains blocked until it receives a connection request. When a connection request arrives, it is accepted, its descriptor is placed in a *wait list* and a request is sent to the *MainManager* class to insert a new client through the *AddClient()* method. After this, the method remains blocked, waiting for new requests;

The *ClientInterface* class has a main method that reads data from the buffer and sends them to all clients whose descriptors are in a *send list*. If it is not possible to send data to a given client, its descriptor is removed from the list

and a request is sent to an object of the *MainManager* class to remove this client. If there is some client in the *wait list*, the method searches for an MPEG sequence initial code. When this code is found, the stream is sent to all the clients whose descriptors are in the *wait list* and these descriptions are moved from the *wait list* to the *transmission list*. Then the method returns to reading the buffer, restarting the whole process;

To transmit MPEG-1 streams using the HTTP protocol, the *ClientManager* class was implemented in a similar way to those used for TCP transmission. The only difference is that before inserting a client in the *wait list*, a handshake using HTTP protocol is performed. During this handshake, the server uses the header fields "Pragma: no-cache" and "Cache-Control: no-cache" to tell the client that the content must not be cached. The format of the stream is set up through the header field "Content-Type: video/mpeg". The length in bytes of the video file is set up in the field "Content-Length: 45000000". This field is necessary and must be configured with a high value because some players start video and audio execution only when the file is completely loaded. *ClientInterface* class was implemented in a similar way to those used with TCP protocol;

To transmit MPEG-1 using UDP protocol, *ClientManager* class was implemented to allow insertion and removal of clients. When a client is inserted, a socket is created to it and its identifier is inserted in a *transmission list*. The *ClientInterface* class has a method that reads data from the buffer and sends it to all the clients whose identifiers are in the *transmission list*. For transmissions of MPEG-1 using IP Multicast, the *ClientManager* and *ClientInterface* classes are similar to those used for a UDP transmission. The only difference is that the *ClientManager* class receives a class D IP address, instead of an IP unicast address;

To transmit MPEG-1 video streams using the RTP transport protocol, the *ClientManager* class was implemented in a similar way to those used for UDP transmission. The *ClientInterface* class has a method that reads data from the buffer and scans it in order to identify the data structures of MPEG standard as well as some MPEG Frames header fields whose values are placed in specific RTP permanent header fields. In this way, MPEG-1 video is packed in conformity to the rules posed in [10]. To transmit MPEG-2 streams, part of the code of *VideoLan Server* [8] was used to produce the transmission packets. The code was adapted to read a stream from the buffer and to send it to a *ClientInterface* object. One *ClientInterface* class was implemented to receive *VideoLan Server* streams, instead of reading it from the buffer.

3.3 Secondary Server

One of the main features of DynaVideo is its ability to adjust the service configuration dynamically. The idea is to identify network links with unnecessary connections to try to eliminate them. When it is not possible to use multicast, a good solution is to instantiate *Secondary Servers (SS)* agents. *SS* has two classes. The *controller class* interprets commands received from *DM* requiring: to start or to finish sending streams (Start/Stop Server); to add or to remove a client

(Add/Delete Client); and to move or to clone *SS* (Move/Clone Server). The commands are sent by *DM* in *Protocol Data Units*. Each command has a code and optional information such as the client IP and port number. The *Streamer* class receives the video stream from *PS* and sends it to the clients in the *ClientTable*. *SS* is started through an *Agent Dispatcher*, which is responsible for the creation, destruction, cloning, moving and forwarding commands to *SS*. *SS* presents the following methods:

- *AddClient*: controller adds an IP address and an associated port in *ClientTable*. To send streams, *SS* scans this table.
- *DeleteClient*: controller removes the IP address and the port from the *ClientTable*.
- *StartServer*: controller requests *SS* to wait for UDP packets from the *PS*. When *SS* receives a packet, it scans the client vector in order to forward this packet.
- *MoveServer*: controller requests *SS* to stop sending streams. It moves *SS* (with all its clients) to the location determined by *DM*.
- *CloneServer*: a copy of an agent is created in another context.
- *StopServer*: controller requests *SS* to stop sending streams. It ends the execution of an agent.
- *DecodeCommand*: controller receives, interprets and executes the commands sent by *DM*.

To validate the *SS* design, we have implemented it using two different environments: IBM Aglets [14] and Agent Lua [11].

Aglet One implementation of *SS* was made using the Aglet library [14], developed by IBM. This library is composed of a set of classes written in Java. These classes have methods that allow the agent to move or to clone from one execution context to another (*dispatch/clone*) and to be extended with other functions. Tahiti [14] is one application that implements the Aglet execution context concept and was used in this work.

The *SS* Aglet agent has two classes: the *Controller* and the *Streamer*. The *Controller* uses the methods inherited from the Aglet class to move (*dispatch*) or to clone (*clone*) the agent to another context. When an Aglet needs to go to another place, the method *OnDispatch* is executed to prepare the Aglet to travel. This method calls *Stop Stream* from the *Streamer* class. When the agent arrives at its target, it executes the method *OnArrival* that askes the *Streamer* to *Start Stream*. The *Streamer* starts to receive the video stream from the *PS* and sends it to the same list of clients that it had at the original place. To start a new *SS*, an *Agent Dispatcher* must be running. When it receives a command from *DM* to start an agent in another place, it clones itself to that place. The new *SS* will, then, use the *controller* to communicate with *DM* and receive commands such as *Add Client* or *Delete Client*. The class *Streamer* receives PDUs from *PS* and sends them to the clients in the list that are initially empty.

Agent Lua The DynaVideo *Secondary Server* was also implemented using the Agent Lua library (aLua) [11]. The aLua distributed programming mechanism is an event-driven extension of the interpreted language Lua [12]. It offers support to send messages to remote processes containing codes to be executed at destination. In aLua, each agent is an independent process that communicates with another agent through asynchronous messages. This mechanism has two important features. It is a non-blocking one, once it uses an event-driven approach and considers each event as an atomic block that must be executed as a whole. Since aLua runs in an interpreted environment, it is easy to modify *SS* whenever necessary without disturbing other DynaVideo modules. This feature introduces flexibility to the *SS* implementation.

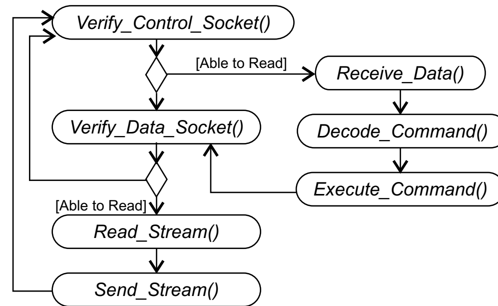

Fig. 4. *SS* Activity Diagram in an aLua Enviroment

The *SS* initialization in the aLua environment happens in this way. Firstly, it is necessary to implement a component that acts as an *agent dispatcher*. Then, *DM* asks the *agent dispatcher* to prepare execution contexts, by sending to it the command *Conf (address)*. In response to this command, the *agent dispatcher* issues a *spawn* command to the respective address. After this, an aLua environment becomes ready in the new location. In order to trigger the execution of a *SS*, *DM* sends an activate command to the *agent dispatcher*. In response, it sends the Lua code of the secondary server to the newly created aLua environment. When the code is received, it begins to be executed.

The activity diagram shown in Fig. 4 describes the *SS* implementation using aLua. An infinite loop verifies if there are, in a control socket, commands available for reading. In this case, the commands are decoded and executed. It also verifies if there are, in a data socket, data available for reading. In this case, it sends them to all the clients listed in the client table.

To move an *SS* from a location to another, *DM* asks the *agent dispatcher* to initialize a secondary server code in the new location. When the sent code is running, it adds all the clients of the old location in the new one. Then, the manager orders the primary server to stop sending video to the old location, and to begin sending it to the new one. And finalizes the agent being moved, by issuing a *Com_Exit()* command.

4 Results

In order to test the performance of the DynaVideo service we did some experiments on the following scenarios. The *Primary Server (PS)* was configured in a PC Pentium MMX 300 with 128 Mb RAM memory, a 100 Mbps Ethernet card and Linux operating system. A 4 Mbps real time MPEG-2 stream was generated and transmitted from the *streamer* to *PS*. This stream was generated by an Apollo card installed in a PC Pentium II 400 with 64 MB RAM memory and with Microsoft Windows NT 4.0 operating system. The utilization of the link that connects *PS* to an IBM 8265 Switch, when *PS* was transmitting UDP datagrams to a multicast address, was 9.5% (9.5Mbps). Since a video stream was being generated at a constant rate of 4 Mbps, the reception and transmission of this stream consume 8 Mbps, in other words, 8% of the band. Considering that no other application was using the link at that moment, we have concluded that 1.5 % of the band was consumed by the overhead of transport, link and network protocols. In another experiment, a *PS* was configured in a PC Pentium III 600 with 128 Mb RAM memory, 100 Mbps Ethernet card and Linux operating system. This server received a 4 Mbps MPEG-1 video stream directly from the *streamer* and transmitted it to twenty (20) MTV clients using the UDP protocol. This scenario is illustrated by the first part of Fig. 5. In this case, the utilization of the link that interconnects *PS* to the network reaches 100%. To prove the feasibility of the *SS* concept, we move an *SS* to a machine at the LCC network. In this scenario (second part of Fig. 5), we can serve 29 clients: *PS* transmitting to 19 clients and to *SS*, and *SS* transmitting to 10 clients. With this experiment, we confirm the scalability of the DynaVideo approach.

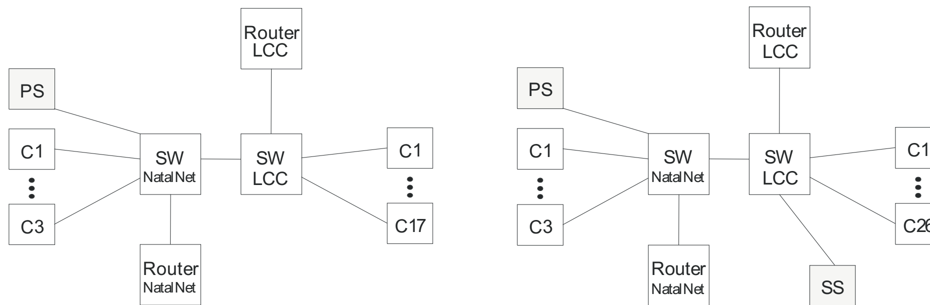

Fig. 5. DynaVideo service experimentation

The DynaVideo service was tested in a local network with the following clients: MTV (MPEG-1 using TCP, UDP, HTTP and IP Multicast), VideoLan (MPEG-2 using UDP), Windows Media Player (MPEG-1 and MPEG-2 using HTTP), RealAudio (MPEG-1 using HTTP), and JMF (MPEG-1 using RTP). In all clients, the video was played at a good quality. Fig. 6 shows an MPEG-1

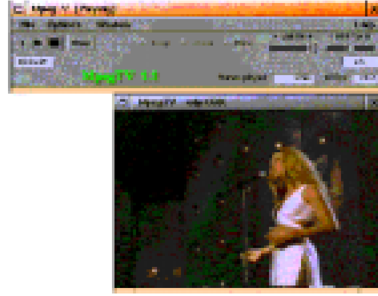

Fig. 6. MPEG-1 Video exhibited by MTV client

Fig. 7. MPEG-2 Video exhibited by VLC client

video being exhibited by an *MTV* client. Fig. 7 shows an MPEG-2 video being exhibited by a *VideoLan* client.

5 Conclusions

This paper has presented the architecture and implementation of the Dynamic Video Distribution Service (DynaVideo) designed for generic data communication environments, supporting different video formats and different client types. In this paper, we have described the following DynaVideo components: *Dynavideo Manager* that controls the service execution; *Primary Server* that has direct access to video sources and whose function is to capture a video stream from the source and to transmit it to the service clients; and *Secondary Server* that acts like a reflector, receiving the video stream from a *Primary Server* and forwarding it to the service clients.

The main feature of the *Secondary Server* is that it can move through the network. This allows the dynamic reconfiguration of the service. The importance of dynamic reconfiguration of video distribution systems is also discussed in [13]. This report proposes an infrastructure to distribute video and uses agents to update and to fix bugs in the code of the replicas. Other works [15][16] have adopted the replication idea, but do not provide support to service reconfiguration during real time video transmissions. DynaVideo allows the transmission of MPEG streams to the MTV clients (MPEG-1 using TCP, UDP, HTTP and IP Multicast), VideoLan clients (MPEG-2 using UDP), Windows Media Player clients (MPEG-1 and MPEG-2 using HTTP), RealAudio clients (MPEG-1 using HTTP) and JMF clients (MPEG-1 using RTP). In the experiments done, the video received by these clients was at good quality. In the implementation, we have adopted a configuration-based approach to the *DynaVideo Manager*. This module integrates the other modules of DynaVideo and acts as a mediator, sending and forwarding commands to other modules. The configuration-based approach facilitates the change of one module without disturbing the others. This issue promotes the reuse and allows integration of new services in DynaVideo

whenever necessary. To support *Secondary Server* mobility, we have developed two implementations. One of the implementation uses the Aglet library, an IBM product. The other one uses aLua, an event-driven mechanism that offers support to move processes. The implementations validated the feasibility of the approach, which is a good alternative to services dealing with unstable demands, like TV distribution. The implementation of the *Traffic Monitor* module was done in another work and it is described in [1]. Currently, the *Configuration Optimizer* and the *Fail Manager* modules are under development.

References

1. Madruga, M., Batista, T., Lemos, G.: SMTA: Um Sistema para Monitoramento de Tráfego em Aplicaçẽs Multimídia. CLEI'2000 (2000)
2. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language for Object-Oriented Development. Documentation Set Version 0.91 Addendum UML Update (1996).
3. Coding of Moving Pictures and Associated Audio for Digital Storage Media up to About 1,5 Mbits/s. ISO/IEC International Standard 11172 (1993).
4. Generic Coding of Moving Pictures and Associated Audio Information. ISO/IEC International Standard 13818 (1994).
5. Realserver Administration Guide RealSystem G2. http://www.real.com/serveradminguideg2.pdf (2001).
6. VideoCharger Server Key Features. VideoCharger, IBM DB2 Digital Library, http://www-4.ibm.com/software/data/videocharger/vcserverkey.html (2000).
7. Windows Media Technologies. http://www.microsoft.com/windowsmedia/ (2001).
8. VideoLan. Ecole Centrale Paris, http://www.videolan.org (2001).
9. Schulzrinne, H., Casner, S., Frederick, R., V. Jacobson: RTP: A Transport Protocol for Real-Time Applications. RFC 1889 (1996).
10. Hoffman, D., et al.: RTP Payload Format for MPEG1/MPEG2 Video. RFC 1849 (1998).
11. Ururahy, C; Rodriguez, N.: Alua: An event-driven communication mechanism for parallel and distributed programming. PDCS'99, Fort Lauderdale, FL (1999).
12. Ierusalimschy, R, Figueiredo, L, Celes, W.: Lua - an extensible extension language. Software: Practice and Experience, 26(6):635-652 (1996).
13. Kon, F., Campbell, R. et al.: Dynamic Reconfiguration of Scalable Internet Systems with Mobile Agents. Technical Report, Department of Computer Science at the University of Illinois Urbana-Champaign (1999).
14. Lange, D., Oshima, M.: Programming and Deploying Java Mobile Agents with Aglets. Addison Wesley (2000).
15. Parveen Kumar, L. H. Ngoh, A. L. Ananda.: A Programmable Audio/Video Streaming Framework for Broadband Infrastructures. Network Storage Symposium - NetStore '99. Seattle, WA (1999).
16. Brian Noble, Ben Fleis, Minkyong Kim, Jim Zajkowski.: Fluid Replication. Network Storage Symposium - NetStore '99, Seattle, WA (1999).