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 considering specific infrastructures or have been tailored to specific application requirements, such as stream types or clients which will be supported by the service. 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 was designed to distribute video in a way that is independent of the video format and to interact with different types of clients. The service may be used to distribute video over any digital network, however, it is focused on the Internet. The main feature of the DynaVideo is the possibility of configuring dynamically the service 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 was designed to distribute video in a way that is independent of the video format and to interact with different types of clients. The service may be used to distribute video over any digital network, however, it is focused on the Internet. The main feature of the DynaVideo is the possibility of configuring dynamically the service to a specific demand. Applications that deal with video distribution, such as broadcast of digital television, normally have to deal with abrupt variations on its demand. The number, type and location of the clients of the service vary in short time. This occurs every time that an interesting program begins to be exhibited. Nowadays, many systems can distribute video over digital networks like the Internet. Real System of the Real Networks [5] transmits video streams in many formats, but the focus is on the support of its proprietary format RM. The video may be transmitted with IP Multicast, TCP, UDP or HTTP. The Microsoft Windows Media Service [7] encode the video stream with its proprietary format ASF. The
following video formats: BMP, WAV, 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 its configuration to demand variations. 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 and guide its conception. The idea is that the service continually tries to find an optimized configuration to attend 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].

![DynaVideo Architecture](image)

**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 the ability of moving 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 to update a data structure, the *route graph*, which records the 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 to the service. *CO* is executed in background, searching for a better configuration for the video distribution service, considering the current demand represented by the route graph. Once it is computed a configuration better than the current one, *CO* requests *DM* to:

- move clients from a server to another one;
- add or delete *Secondary Servers*;
- move *Secondary Servers* from a local 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* was added and three clients were 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 the 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 its replacement 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](image)

In the following Sections, we will discuss the specification and implementation of *DM, PS*, and *SS* Modules. The *TM* module was described in [1]. *CO* and *FM* are under development.
2.1 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 allows the 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 methods ServerRegistration(), AddClient(), AddServer(), PrimaryServerRegister(), DeleteClient() and MoveServer(). The choice of the server that will support a new client is made by the SelectServer() method, which searches at the ServiceConfiguration tree (similar to the tree shown in Figure 2) a server with capacity to support the new client.

2.2 Primary Server

The Primary Server (PS) can be configured in conformance with the 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 classes specification depends on the stream generator entity. If there 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 Buffer 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() method. The MainManager class controls the exchange of information among the other classes of PS. It controls data insertion and recovery in a Buffer object, using the methods PutDataOnBuffer() and GetDataFromBuffer(). This class also stores and retrieves the PS parameters configured by the service administrator or informed by some other object, through the SetParameter() and GetParameter() methods. The MainManager class interprets and executes the commands through the Execute() method and adds and removes clients, using AddClient() and DeleteClient() methods.

The Buffer 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 moment of the data
insertion in the buffer. This feature allows the classes, which are accessing 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 to the classes ClientManager and ClientInterface. ClientManager controls the 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 the 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 the 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 to the requirements of the client type associated with the class, and transmits the data. For instance, this class adapts MPEG streams allowing its transmission using the RTP protocol.

2.3 Secondary Server

One of the main features of the DynaVideo is the possibility of dynamically adjusting the service configuration. The idea is to identify network links with unnecessary connections and try to eliminate them. When it is not possible to use multicast, one fine 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 finishes the execution of an agent.
3 Implementation Issues

3.1 DynaVideo Manager

One of the target applications 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 party 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 the `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 Figure 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.

![Add Client Case Sequence Diagram](image-url)
3.2 Primary Server

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 memory with a Microsoft Windows NT 4.0 operating system. In this server, it was created a Named Pipe with an 8 MB buffer. The Apollo software was configured to write the video stream in 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 remains reading the MPEG stream from the pipe and transmitting 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. ClientInterface class has a main method that reads data from the buffer and sends them to all clients whose descriptor 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 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: 450000000". 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 of 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 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

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/cloned) 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, whose Class Diagram is shown in Figure 4, 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 asks the Streamer to Start Stream. The Streamer starts to receive the video stream from the Primary Server 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 it to the clients in the list (that are initially empty).
**Agent Lua** 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 presents two important features: it is a non-blocking one, once it uses an event-driven approach, and it 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.** Aglet Secondary Server Class Diagram

**Fig. 5.** SS Activity Diagram in an aLua Environment

**Fig. 6.** Utilization of the PS network link

To move an SS from a location to another, the agent dispatcher (written in Lua) receives a command to move one agent (SS) from one host to another. Then, it creates an agent in a new host through the `Com_spawn()` command and sends a `Com Exit()` command to the agent being moved. The activity diagram shown in Figure 5 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.

4 Results

In order to test the performance of the DynaVideo service we did some experiments considering 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. Figure 6 shows the utilization of the link that connects PS to an IBM 8260 Switch. This figure shows two different areas. In one, PS is off and the utilization is nearly null. In the other, PS is transmitting UDP datagrams to a multicast address and the utilization is 9.5% (9.5Mbps). Since a video stream is generated in 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 has 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 Figure 7. 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 a SS to a machine at the LCC network. In this scenario (second part of Figure 7), 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.

![Diagram](image-url)

*Fig. 7. DynaVideo service experimentation*
The DynaVideo service was tested in a local network with the following clients: \textit{MTV} (MPEG-1 using TCP, UDP, HTTP and IP Multicast), \textit{VideoLan} (MPEG-2 using UDP), \textit{Windows Media Player} (MPEG-1 and MPEG-2 using HTTP), \textit{RealAudio} (MPEG-1 using HTTP), and \textit{JMF} (MPEG-1 using RTP). In all clients, the video was played with a good quality. Figure 8 shows a MPEG-1 video being exhibited by an \textit{MTV} client. Figure 9 shows an MPEG-2 video being exhibited by a \textit{VideoLan} client.

5 Conclusions

This paper has presented the architecture and implementation of a 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: \textit{Dynam Video Manager} that controls the service execution; \textit{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 \textit{Secondary Server} that acts like a reflector, receiving the video stream from a \textit{Primary Server} and forwarding it to the service clients.

The main feature of the \textit{Secondary Server} is that it can move through the network. This allows the dynamic reconfiguration of the service. The importance of using mobile agents for dynamic reconfiguration of systems is discussed in [13]. This piece of work proposes an agent based infrastructure to distribute video, but does not deal with different video formats and third part clients as DynaVideo. Other pieces of work [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 presented a good quality. In the implementation, we have adopted a configuration-based approach to the DynaVideo Manager. This module does the integration among 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 fine 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 module and the Fail Manager module are under development.

References