Virtual Streams: A Generic Interface for Uniform Data Access

by

Saadeddine S. Mneimneh

B.E. in Computer and Communication Engineering

American University of Beirut (1995)

Submitted to the Department of Civil and Environmental Engineering in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

Author .....
Civil and Environmental Engineering
May 1, 1997

Certified by .....
Steven R. Lerman
Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by .....
Prof. Joseph M. Sussman
Chairman, Departmental Commitee on Graduate Studies

Virtual Streams: A Generic Interface for Uniform Data Access

by

Saadeddine S. Mneimneh

Submitted to the Department of Civil and Environmental Engineering on May 1, 1997, in partial fulfillment of the requirements for the degree of Master of Science

Abstract

UNIX™ I/O is based on a file centric model that uses file descriptors, but these map poorly onto communication channels such as network sockets. The stream represents a more general I/O model that overcomes this problem. We illustrate the design of a generic interface that uses a stream abstraction rather than a file. The interface is implemented as a layer on top of file I/O, network and Web services. The goal of our design is not only to allow streaming of data over the network but also to provide a unified interface for accessing information whatever storage it might be located on. A stream may originate from a memory buffer, a local file, an inter-process communication pipe, or a network connection. Hence the name, Virtual Streams.

Our strategy addresses three problems. First, uniformity of interface to achieve independence of the media source; second, platform-independence based upon widely available services like FTP and HTTP; and third, semantics for seeking into a stream.

Thesis Supervisor: Steven R. Lerman
Title: Professor of Civil and Environmental Engineering

Acknowledgment

During my stay at MIT, I heard many stories about Students and Advisors, many of which made me realize how great a person my advisor is. To my advisor Steven Lerman, I owe most of my thanks.

I also would like to thank Dr. Judson Harward for the advices he gave me and for his continuous help and support.

I thank all my lab mates for making my time enjoyable at CECI.

I thank all the professors who taught me at MIT, especially Arvind, Michael Sipser, and David Gifford. They gave me a special twist of mind.

I thank all my friends (Lebanese and non-Lebanese) for making life possible at MIT. Among them, I would like to thank most Walid Fayad.

This research was supported by LOCKHEED SANDERS.

To Mom, Dad, Weddo, Hana, and Abed

I Love You...

Second Dedication

To the Perfectionist
who couldn't do anything perfect.

I know you... I admire you...

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

This thesis addresses the idea of virtual streams. The virtual stream is a powerful concept that hides the technicalities behind the way data is stored and accessed. This chapter describes some of the problems encountered in traditional, non-stream based I/O operations and motivates the idea of having a unified interface for accessing data across different sources. This is achieved through a comparison between the different ways of accessing information from various sources including disk files, network sockets, and inter-process communication pipes.

A description of some attempts to provide streaming and unification of data access follows. This is done by showing the efforts made to hide the discrepancies in accessing data across a variety of sources which is done through the inclusion of a set of I/O libraries that implement a layer of high level operations on top of traditional I/O operations.

Next, I argue that these attempts are not sufficient because they don't span all kinds of I/O operations. This leads to a brief description of the objectives of this research, and I conclude the chapter by describing how the rest of the thesis is organized.

1.2 Traditional I/O

UNIX™ I/O is based on a file centric model that uses file descriptors. A file descriptor is simply a data item (usually an integer) that represents the source/destination of data [1]. Although file descriptors provide a convenient way of accessing data across many sources, for every source they have to be treated differently. In particular, file descriptors map poorly onto communication channels like a network socket or an inter-process communication pipe. Too many parameters have to be considered in each case which is counter-intuitive to what a stream should be. As the following sections will suggest, file descriptors do not provide a powerful abstraction of data access. Next, I will explain file descriptors and describe some examples of using them in different contexts including disk files, network sockets, and inter-process communication pipes.

1.2.1 File Descriptors

A file descriptor is a integer Id that represents a file, a socket, or a pipe. File descriptors are sometimes referred to as file handles in other operating systems such as MS Windows™; however, they have the same functionality.

From the programmer's perspective, it is most convenient to program in a way that is independent of the type of source or destination; however, this is not always possible as the examples in the following section will demonstrate. For instance, a file descriptor associated with a network socket has to perform more initializations in order to get the host address and connect to the socket on that remote host. Moreover, the errors that we need to watch for in every case are almost completely different in nature.

The next section will list some of these examples that will illustrate the points mentioned above.

1.2.2 Examples of Using UNIX File Descriptors

There is always a file descriptor open with every disk file or network socket that a process has opened. A pipe (used in inter-process communication) is an exception because it has two file descriptors associated with it as it will be seen shortly [1].

The first example of a file descriptor usage will illustrate how disk files are manipulated. Dealing with files on disk is the easiest of all file based I/O operations and the most intuitive to think about. The following example written in C shows how to copy data from one file on disk to another.

Example 1:

```
void copyFromTo(char * from, char * to) {
int fromFD, toFD;
int bytesread, byteswritten;
char * buf[BUFSIZE];
if ((fromFD=open(from, O_RDONLY))==-1)
    syserr("open for read");
if ((toFD=open(to, O_WRONLY | CREAT, 0666))==-1)
    syserr("open for write");
while ((bytesread=read(fromFD, buf, sizeof(buf)))!=0) {
    if (bytesread=-1) syserr ("read");
    byteswritten=0;
    do {
            if (n=write(toFD, &buf[byteswritten], nread-nwritten))==-1)
                    syserror("write");
            byteswritten+=n;
    } while (byteswritten<bytesread);
}
if (close(fromFD)==-1 | close(toFD)==-1) syserr("close");
}
```

This example illustrates the smallest safe file copy operation that can be implemented using UNIX™ file descriptors. Note that we could not have replaced the copy operation with a single line as follows:

```
if (write(toFD, buf, bytesread)!=bytesread) syserr("write")
```

This is because it is not an error if the count returned by a write operation is less than the requested count [1]. For this reason, we code the copy operation to absorb partial writes and to keep trying until a real error occurs such as reaching the file limit.

Before trying to compare this implementation with other file descriptor usages, it is useful to express the copy operation as pseudocode to understand clearly what it really does. In future examples, every implementation will be followed by its pseudocode equivalence. What will be noticed from the various examples is that their pseudocodes will tend to be very similar, yet the real implementations are different. This was behind the real motivation to come up with a uniform generic interface that will be as close as possible to the pseudocode.

Here is the pseudocode for the previous example:

```
fromFD.Open(from, "_READ");
toFD.Open(to, "_WRITE");
while (!fromFD.Finished) {
    bytesread=fromFD.Read(bufsize, buf);
    byteswritten=toFD.Write(bytesread, buf);
}
fromFD.Close();
toFD.Close();
```

The pseudocode does not deal with internal errors. It just expresses what is desired from the whole operation. In the above pseudocode, two files are opened, one for read and the other for write. Then data is transferred from one file to the other until no more data is available. The next example illustrates how to read information form a network socket. The same file descriptor used before for files on disk can be used to access the network; however, the descriptor has to be treated differently and more complicated system calls have to be used.

Example 2:

```
void NetworkRead(char * hostname, unsigned short portnum) {
struct sockaddr_in sa;
struct hostent    *hp;
int a, netFD;
char * buf;
int n;
int bytesread, br;
if ((hp= gethostbyname(hostname)) == NULL) {
   return(-1);
}
memset(&sa,0,sizeof(sa));
memcpy((char *)&sa.sin_addr,hp->h_addr,hp->h_length);
sa.sin_family= hp->h_addrtype;
sa.sin_port= htons((u_short)portnum);
if ((netFD= socket(hp->h_addrtype,SOCK_STREAM,0)) < 0) return(-1);
if (connect(s,&sa,sizeof sa) < 0) {
     close(netFD);
     return(-1);
}
bytesread= 0;
br= 0;
while (bytesread<n) {
     if ((br= read(netFD,buf,n-bytesread)) > 0) {
                    bytesread += br;
                    buf += br;
     }
     else if (br < 0) {
                    close(netFD);
                    return(-1);
     }
}
}
```

In the above example, we initialize the address structure *sa* with the appropriate host address and port number. Then we create a socket (which is really nothing more than a UNIX™ file descriptor) and connect to the right address. The reading is done in the exact similar way as in example 1. The following illustrates a pseudocode for this example:

```
netFD.Open(hostname, portnumber)
while (!netFD.Finished && bytesread<n) {
     bytesread+=netFD.Read(n-bytesread, buf+bytesread);
}
netFD.Close();
```

The reader can notice how close this pseudocode is to the previous example. The last example illustrates a UNIX™ inter-process communication pipe. Even pipes are nothing more than file descriptors.

Example 3:

```
void pipe() {
int FD[2];
int bytesread;
char buf[100];

if (pipe(FD)==-1) syserr("pipe");
if (write(FD[1], "hello", 6)==-1) syserr("write"); /*6 is the length of the string*/
switch (bytesread=read(FD[0], buf, sizeof(buf))) {
     case -1: syserr("read");
     case 0: fatal("EOF");
}
close(FD[0]);
close(FD[1]);
}
```

As mentioned earlier, pipes have two file descriptors associated with them. Pipes are used for the purpose of inter-process communication where a process writes to one end of the pipe (first file descriptor) and another process reads from the other end of the pipe (second file descriptor) [1]. In the above example, the situation is simplified to one process writing to and reading from the same pipe to emphasize on how the file descriptors are used and not on the complexity introduced by forking another process.

It can be seen clearly how close this example is to the first example of copying from one file to another, yet the code is obviously not the same. Here is the pseudocode for this example:

```
FD.Open();
FD[0].Write(6, "hello");
FD[1].Read(6, buf);
FD[0].Close();
FD[1].Close();
```

All of these examples show that file descriptors present a common tool for accessing data across many sources; however, they do not provide a uniform means of access. In the following section I will describe some of the efforts done to unify data access across standard I/O, memory and disk files. Due to their uniformity of access, we refer to these unified I/O sources as data streams.

1.3 Data streams

Designing and implementing an I/O facility for a programming language is notoriously difficult. File descriptors are designed to handle basic I/O operations; however, setting up these file descriptors to retrieve data from the right sources is not an easy task to do. Moreover, non-trivial programs use many user-defined types, and input and output of those types must be handled. An I/O facility should clearly be easy, convenient, and safe to use, efficient and flexible, and above all complete. Nobody has come up with a solution that pleases everyone [2].

The stream I/O facilities are the result of efforts to meet this challenge. An example of such a facility is the stream I/O library implemented in C++ [2].

1.3.1 Standard I/O and Memory Streams

In C++, high level I/O operations similar in simplicity to the pseudocodes presented earlier are possible. The following example illustrates an operation that outputs a string to the standard output:

```
cout << "this string will be outputed to the standard output";
```

The content of memory is streamed to the standard output (which could be the display or any output device). We can look at this operation from two different perspectives. The obvious (and most usual) view is to regard the standard output as an output stream where data is being sent. Another way to look at it is to give the string a stream-like characteristic

where data is being read and transferred to the output. Regardless of the semantics behind this piece of code, it represents a convenient way of accessing and transferring data. In fact, the other way could also be coded as follows:

```
cin >> s;
```

where the string s receives data from the standard input.

1.3.2 File Streams

In this section, I will illustrate how a function can implement the copy operation in example 1 using the idea of file streams [2]. Here is the code for copy(from, to):

```
void copy(from, to) {
ifstream(from);
ofstream(to);
char buf;
while (from.get(buf)) to.put(buf);
}
```

If we compare this code to the pseudocode illustrated earlier, we can see that they are much alike in structure and semantics. First we open both files, then we start transferring data from one file to another until no more data is available.

The stream library, however, is limited and does not cover all possible cases of I/O operations. The user has to build his/her own utilities on top of this library to provide a stream-like characteristic for all I/O operations. In other words, streams are not virtual; they address a specific type of I/O operations.

After this brief study of file I/O and streams, it is legitimate to ask the following question:

For the various I/O operations, why aren't the codes similar if their pseudocodes are so?

As an answer to this question, the following sections will briefly state the objectives of this research and how this thesis is organized.

1.4 Objectives of the Research

The stream represents a more general I/O model than file descriptors. The objective of this research is the design of virtual streams, a generic stream interface for uniform access of data across all sources. The interface is implemented as a layer on top of file I/O, network, and Web services.

Although streaming makes more sense in the case of network sockets, the goal of our design is not only to allow streaming of data over the network but also to provide a unified interface for accessing data whatever storage it might be located on. A stream might originate from a memory buffer, a local file, an inter-process communication pipe, or a network connection.

The design strategy addresses three problems: First, uniformity of interface to achieve independence of the data source; second, platform-independence based on widely available services such as FTP and HTTP; and third, semantics for seeking into a stream which is an important property currently not available for all kinds of streams.

1.4.1 Uniformity of Access

With the current evolution of the Internet, networked data acquisition and exchange of information are becoming an integral part of computer usage.This implies that data I/O is no longer restricted to accessing local files on disk. As a consequence, I/O operations should be made flexible enough to deal with remote data. File descriptors are not very convenient for such a purpose. Since we are suggesting the stream as a more appropriate model, we need a way to access streams uniformly.

In order to provide this facility, we differentiate between streams and *streamSpecs*. A *streamSpec* is a data structure that specifies the source of the stream. StreamSpecs will be described in chapter 4.

1.4.2 Platform-independence

Platform-independence is mainly supported by using platform-independent services such as FTP and HTTP. By implementing our stream model as a layer on top of system services, we can achieve platform independence easily if these services are available for all platforms. FTP and HTTP are certainly two cross-platform protocols that run under UNIX™ with its different implementations, MS Windows™, MAC OS and others.

On the other hand, supporting platform-independence using cross-platform network services such as FTP and HTTP implies blocking on every stream request because of the blocking nature of these services. How can we provide an implementation of such a virtual stream and keep a responsive application if no operation can be performed before every single bit has been transferred? We overcome this problem with existing network services by using multithreaded stream requests. Every network stream runs in a different thread from the main computation thread.

1.4.3 Seeking Semantics

Among the different kinds of virtual streams, the network stream is the one that represents the greatest challenge both in concept and implementation.

For example, what does it mean to seek into a stream coming over a network connection? Seeking back and forth into a network stream is an important problem because some user interface components, e.g. a scroll bar, reflect directly this kind of operation. A scroll bar represents a seeking operation if, for instance, the user moves it while it is tracking the frames of a video chunk brought over the network.

Another problem arises when we have to deal with the case of missing information. The scroll bar must not allow the user to scroll to a position where there is no data to view. One way of achieving this behavior is by providing a *blocking atomic* seek that decides whether the seek operation is to be completed or not, depending on data availability.

1.5 Organization of the Thesis

Chapter 2 provides an overview on some of the previous work done on streaming. This includes research, some new Internet protocols in progress such as RTP and RTSP, and some of the commercial products available from different companies. Chapter 3 described in detail the virtual stream model. A simple state diagram will illustrate how the virtual stream operates. Buffering is studied from two perspectives: seeking limitations and processing speed. Moreover, semantics of the operations associated with a virtual stream are studied in great detail with a comparison between blocking and non-blocking semantics. Chapter 4 provides a brief overview of Object-Oriented Programming and describes the object-oriented design governing the virtual streams. A detailed description of a network stream as a proof of concept is found in chapter 5. Chapter 6, the last chapter in this thesis, summarizes the main points of the thesis and states the contributions and some of the practical applications of the virtual stream model.

Chapter 2

Previous Work

2.1 Overview

This chapter describes some of the previous work done on streaming. This includes some new protocols under research or development such as RTP and RTSP, some of the commercial product from Microsoft (ActiveMovie) and Duplexx Software (NETTOOB) for streaming video, and from Progressive Networks (RealMedia Architecture) for streaming audio and video files and for providing an SDK for developing cross-platform media tools.

2.2 Research and Protocols

In this section, I will describe some of the streaming protocols that are in progress. Some of these protocols (such as RTSP) are still under research. Others have transitioned from the research phase toward development.

2.2.1 RTP (Real Time Protocol)

In January 96, the Real Time Protocol (RTP) was put as a Request For Comment (RFC) on the Internet. This protocol provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio and video [3]. So by its nature, it is a streaming protocol.

However, RTP does not guarantee quality-of-service for real-time services. This is why the data transport is complemented by a control protocol (RTCP) to allow monitoring of the data delivery and to provide minimal control and identification functionality. RTP and RTCP are designed to be independent of the underlying transport and network layer, so they can both be implemented on top of TCP or UDP [3].

Several RTP applications, both experimental and commercial, have already been implemented from draft specifications. These applications include audio and video tools along with diagnostic tools such as traffic monitors. Users of these tools number in the thousands. However, the current Internet cannot yet support the full potential demand for real-time services. High-bandwidth services using RTP, such as video, can seriously degrade the quality of service of other network services. Thus, implementors should take appropriate precautions to limit bandwidth usage [3]. The network stream discussed in chapter 5 offers a compromise. It makes use of services like FTP and HTTP, which are suitably available for the Internet, to provide access of real-time data with a high level virtual stream protocol. The way this is accomplished will be described in great detail in chapter 5.

2.2.2 RTSP (Real-Time Streaming Protocol)

In October 96, Progressive Networks and Netscape Communications Corporations (NSCP) with 40 other companies announced their support for the Real-Time Streaming Protocol (RTSP), a proposed open standard for delivery of real-time media over the Internet. RTSP establishes and controls either single or multiple streams of continuous media. It does not typically deliver the continuous streams itself, although interleaving of the continuous media stream with the control stream is possible [4].

There is no notion of an RTSP connection, but rather a session maintained by an identifier. An RTSP session is not tied to a transport-level session. During an RTSP session, an

RTSP client may open and close many reliable transport connections to the server to issue RTSP requests. Alternatively, it may use a connectionless transport protocol such as UDP.

The protocol is intentionally similar in syntax and operation to HTTP/1.1, so that extension mechanisms to HTTP can in most cases also be added to RTSP. Here is how RTSP works.

Each media stream and session may be identified by an RTSP URL (Uniform Resource Locator). The overall session and the properties of the media the session is made up of are defined by a session description file. The session description file is retrieved using HTTP, either from the web server or the media server, typically by using a URL. The session description file contains a description of the media streams making up the media session, including their encodings, language, and other parameters that enable the client to choose the most appropriate combination of media. In this session description, each media stream is identified by an RTSP URL, which points to the media server handling that particular media stream and names the stream stored on that server [4].

Several media streams can be located on different servers; for example, audio and video tracks can be split across servers for load sharing. The description also enumerates which transport methods the server is capable of. If desired, the session description can also contain only an RTSP URL, with the complete session description retrieved using RTSP [4].

The first draft of the protocol specification, RTSP 1.0, was submitted to the Internet Engineering Task Force (IETF).

2.3 Commercial Products

Commercial products, best represented by ActiveMovie from Microsoft, NETTOOB from Duplexx Software Inc., and RealMedia Architecture from Progressive Networks, repre-

sent other attempts to provide data streaming.

2.3.1 ActiveMovie

Microsoft introduced the ActiveX components, one of which is ActiveMovie. ActiveX is a new component technology based on OLE (Object Linking and Embedding) extended with network and Web capabilities.

ActiveMovie provides streaming of video and allows decoding of various video formats.

ActiveMovie has a flexible, extensible architecture for easy integration of new technologies and third-party enhancements. The ActiveMovie architecture defines how streams of data can be controlled and processed by using modular components called filters connected in a configuration called a filter graph. An object called the filter graph manager is accessed by applications via programmatic interfaces and controls how the filter graph is assembled and how data is moved through the filter graph. Additionally, a simple application-level visual programming utility, called the filter graph editor, is included in the ActiveMovie SDK, This utility allows developers to construct and test filtergraphs simply and easily. The following figure shows the relationship of applications and the components of ActiveMovie [5].

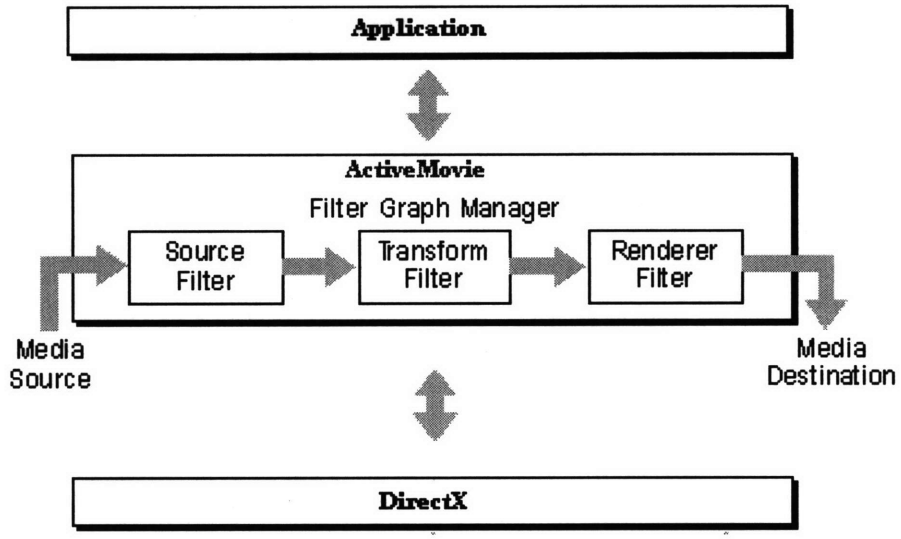

Figure 2.1: The ActiveMovie architecture.

ActiveMovie is accessible at many levels, and the approach used depends on the requirements of the application and the investment in development that is desired. The filter graph manager provides a set of Component Object Model (COM) interfaces to allow communication between the filter graph and the application. Applications can make direct calls to the filter graph manager interfaces to control the media stream or retrieve filter events. Alternatively, applications can use the ActiveMovie OLE control for higher-level programming [5].

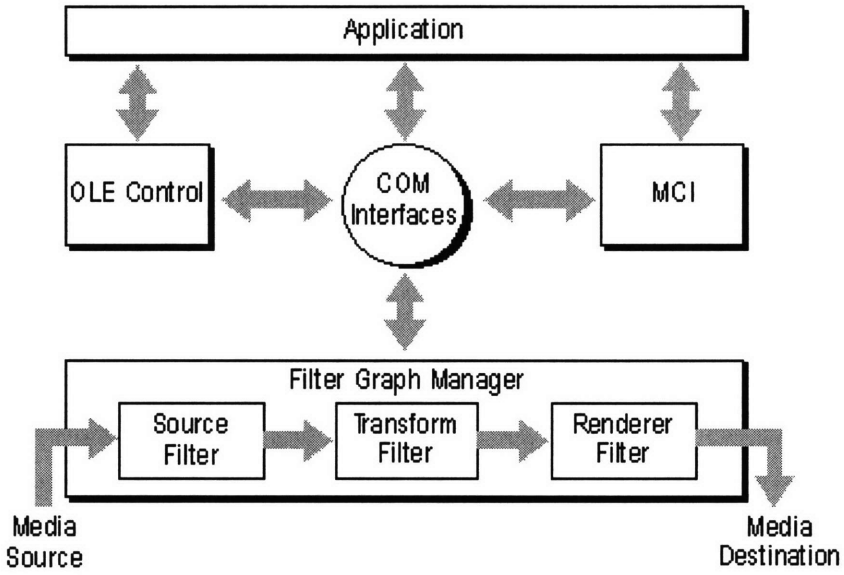

Figure 2.2: Application interface with ActiveMovie.

This architecture is similar to the W3C architecture. W3C is a cross-platform, Web API library written in C. The architecture of W3C library will be described in chapter 5 since it is an essential part of the network stream. This puts the network stream, which is one of the virtual streams, at the edge of technology where it uses similar components as ActiveMovie, namely a set of filters arranged in a predefined setup and controlled by the application level by setting and retrieving events.

2.3.2 NETTOOB

Along the same lines, Duplexx Software Inc. produced a video and audio decoder called NETTOOB. NETTOOB provides a streamer MCI driver (for the MS Windows™ platform) that plays MPEG-1 video files [6].

In this case, the user has to perform the actual streaming of data. All that the driver does is to request data from a DLL (Dynamic Link Library) that has to be implemented by the user. The driver is called a streamer just because it does not require the data to be available at the time it starts; nevertheless, the DLL has to provide the data at an average rate

faster then the rate at which the driver issues the requests. The DLL provides a simple interface to open a stream, read from the stream, seek into the stream, and finally close the stream. The following picture illustrates the interface that needs to be implemented by the user.

- **char * STREAM_Init(long size)**

  Initializes the stream and buffer size and returns a stream name for the driver to identify the stream in case more than one video stream is being played.

- **long STREAM_Read(char * buf, long size)**

  Reads the specified number of bytes from the stream and stores them into the buffer passed to this function by the driver.

- **long STREAM_Seek(long offset)**

  Seeks from the current position into the stream as specified by the offset and returns that offset if successful or zero if it fails to perform the seek.

- **int STREAM_Notify(unsigned int code)**

  Gets an integer notification code such as STREAM_STOP defined to be 1. It returns 0 if whatever it performs succeeds and -1 otherwise.

The interface defines the whole stream, and no streaming can be done unless this interface is fully implemented. So what really NETTOOB relies on is the this idea of a virtual stream. As a matter of fact, NETTOOB represents the test-bed and the main practical application used in this thesis to assess the feasibility of virtual streams.

2.3.3 RMA (RealMedia Architecture)

As part of their support to RTSP, Progressive Networks developed the RealMedia Architecture (RMA) (and RealAudio earlier for streaming audio files) which comprises a set of tools including players to play coordinated media files, servers to stream media for-

mats using the RMA file filter to convert various data-types to the RealMedia file format, and a plug-in that enables live and on-demand streaming of third party data-types [7]. Progressive Networks also produces an SDK that contains player APIs, file conversion utilities, and media file merge tools.

The figure below illustrates the client-side RMA architecture [7]:

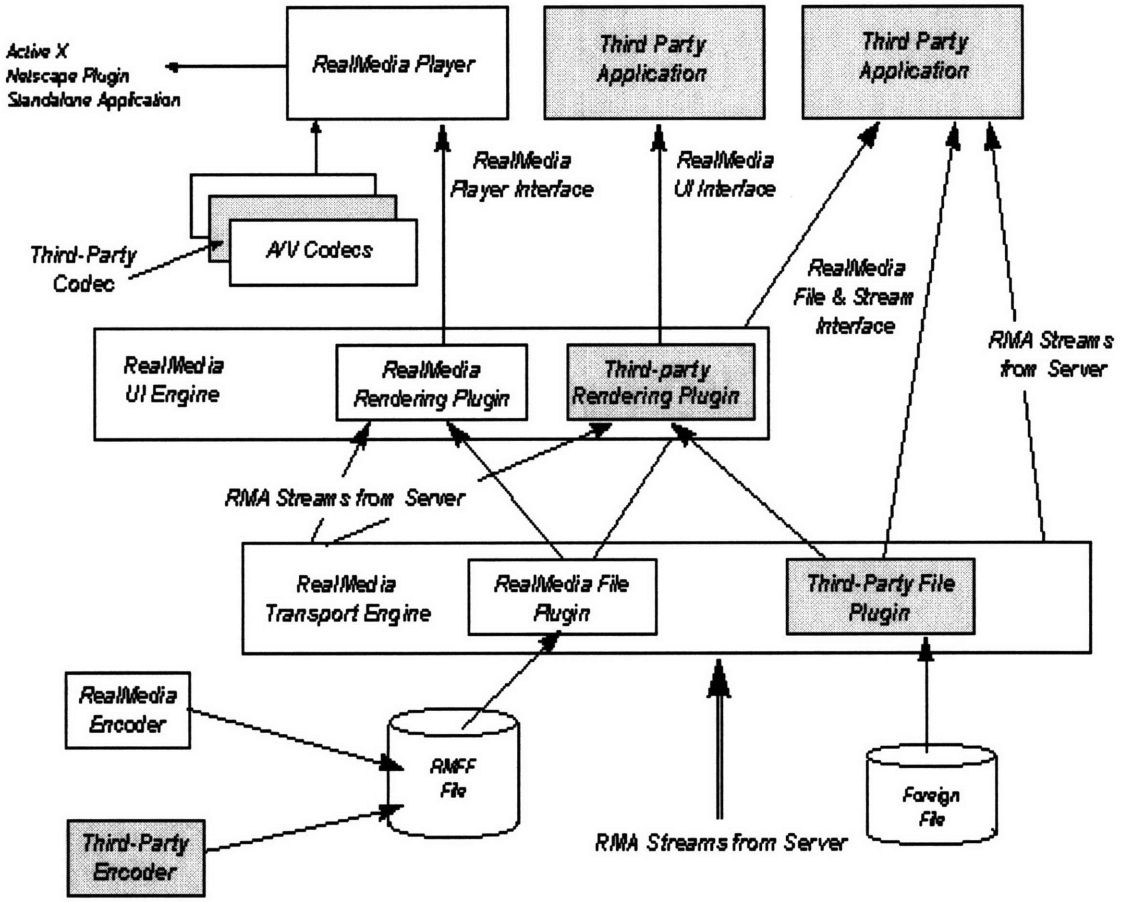

Figure 2.3: The client-side RMA architecture.

The client-side architecture includes:

1. The Player, responsible for presenting a standard UI to the user for control of the multimedia session. The Player can function as a stand-alone application on Windows™

and the Macintosh^{TM}, and can also be embedded in other applications and Web pages as either a Netscape Plug-in or an ActiveX control. The Player also has a pluggable codec interface allowing third-party codecs to be used by the standard Player [7].

2.The UI Engine, providing elements of the Player UI, such as VCR controls and display areas, in the context of a third party application. The Realmedia UI Engine acts as the host for RMA Rendering plug-ins, which allow third party, visual data-types to be displayed in the context of the Player or applications using the UI Engine [7].

3.The Transport Engine, allowing the Player and third-party applications to transparently access RealMedia and native data files and to gain access to streamed multimedia endpoints coming from a server. The RealMedia Transport Engine acts as the host for RMA File plug-ins, which allow third party file-types to be accessed in the context of the Player or other RealMedia applications. Using the RealMedia File plug-in interfaces, third party and standard Realmedia Tools can create and manipulate RealMedia Files [7].

The server side architecture is shown below [7]:

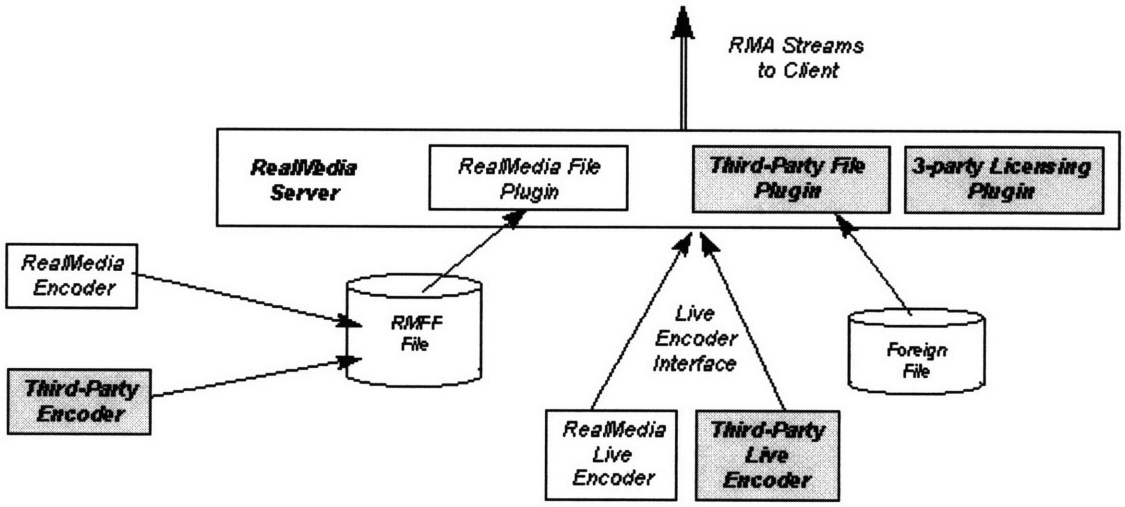

Figure 2.4: The server-side RMA architecture.

The server-side architecture includes:

1. RealMedia File Plug-ins. On Windows™ and the Macintosh™ Server platforms, the same RealMedia File Plug-in developed for the RealMedia client can be used to integrate transparent streaming support for third-party data-types. On UNIX™ platforms the resident shared library architecture is used to provide for the loading and activation of File plug-ins [7].

2. RealMedia License Plug-ins. The Server supports a rich licensing subsystem which allows server stream capacity to be controlled by a vendor-supplied license key on a data-type by data-type basis. If a vendor wishes to support this licensing system, then they must develop a simple shared library which supports the licensing of their data-type [7].

3. Server-side encoding can be done by using the RealMedia File Plug-in in an identical manner to its use on the client for the creation and manipulation of data-types by third-party and RealMedia standard tools, using the RealMedia File format. For live encoding the architecture provides an interface which can be used to connect live streams of data to the RealMedia Live Encoder Interface. This interface operates in a client-server manner using a variant of the RMA streaming protocol to allow live encoders to be distant from the server during broadcasts [7].

Although this is a cross-platform product, it addresses only a specific set of data streams, mainly video and audio formats.

2.4 What Is Missing?

The described protocols seem to be promising; however, they are low level protocols that do not allow the user to access streams of data in a simple and convenient way that approximates the pseudocodes described in chapter 1. For instance, both RTP and RTSP still rely on TCP or UDP as a transport layer; they represent an extension to socket pro-

gramming, and thus assume a network connection. As argued earlier, a higher level protocol is necessary to access different data sources (e.g. local files, URL file, network stream) uniformly.

Therefore, while these protocols might be useful for the future of the Internet in general and the development of more advanced Web browsers, they do not provide programmers with uniformity of data access. The proposed solution, in contrast, provides a high level stream interface that uses existing protocols (FTP, HTTP, ...) and makes it easy to the user to access data uniformly. In fact, the proposed stream model can even be built on top of RTP itself when this protocol becomes available.

As far as commercial products are concerned, they either target a specific platform or a specific data-type to be streamed. The proposed stream model does not impose such restrictions on the kind of data streams.

Chapter 3

The Virtual Stream Model

3.1 Overview

We are proposing the virtual stream as a general model for I/O operations. A stream has been suggested in the literature as a more appropriate model for I/O operations than the file since, using stream semantics, data can be accessed generically from different types of sources like a disk file or a memory buffer [2].

The main contribution in this thesis (in addition to using stream semantics for I/O operations) is that we restore to the stream the property of streaming. In other words, data access and data transfer are two separate operations that can be performed simultaneously. Therefore, we define the virtual stream as a sort of connection from a source to a destination where data can be accessed only at one end of the stream (at the destination). A stream copies data from one end to the other within a certain delay while permitting data access. However, a stream does not guarantee that the data at one end will eventually reach the other end.

We can think of a stream as a handicapped file: while a file is capable of providing data at any time, a stream might not be able to do so. Therefore, it might seem at the first glance that a stream is a restricted form of a file; however, it provides a better abstraction and it is more suited for the kinds of I/O operations we perform today, especially network I/O.

Conversely, we should think of a file as a fast stream that connects the disk as its source to a memory buffer as its destination. Therefore, a file should be considered as a special case of a virtual stream, and it is so implemented in our generic stream interface.

3.2 The Virtual Stream Components

As informally described above, a virtual stream model has to incorporate the following:

1. Stream Source.
2. Stream Destination.
3. Stream Unreliability.

3.2.1 The Stream Source

The Stream Source **SS** is modeled merely as a sequence of bytes at one end of the stream. This will allow for different kinds of data sources to be valid, even multiple sources that are merged together.

3.2.2 The Stream Destination

The Stream Destination **SD** is modeled as a memory buffer. This will not impose any restriction on the destination itself; however, it will add some practicality in thinking about the model. It also allows us to define the stream behavior in terms of observations on the buffer. The memory buffer can represent any intermediate stage before the destination itself. Note that it would be of no practical use to make **SS** a memory buffer as well or else the whole stream model will collapse to a buffer. However, a memory buffer can definitely be an **SS** as suggested in the Memory Stream (a buffer) in the next chapter. The following figure depicts the stream model described above. The dark cell in the buffer indicates the position corresponding to the latest *read*, *write* or *seek* operation.

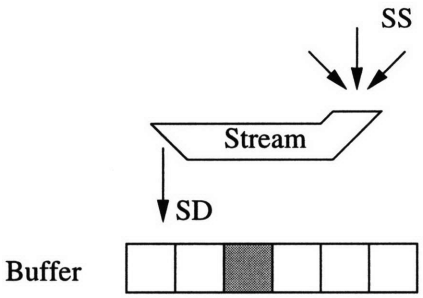

Figure 3.1: The virtual stream components.

3.2.2.1 Buffer Size and Seeking

The size of the **SD** buffer can grow up to a maximum **M** defined as a characteristic of the stream. **M** specifies the maximum displacement in the **SD** buffer that a *seek* operation can successfully perform. For example, a video stream might set **M** to the size of a frame since no information about the previous frame is needed when processing the current frame.

3.2.2.2 Buffer Size and Processing Speed

An interesting case arises if, while processing parts of the frame, other unread parts of the frame are lost. We can choose a buffer size to avoid this situation. Assume that the average rate of reading the **SD** buffer drops to **Ro** due to processing time. Let **Ri > Ro** be the rate at which **SS** operates. If we normalize these rates with respect to **Ri** we get:

r = Ro / Ri for **SD**

and

1 for **SS**.

The first time the buffer becomes full, **Mr** cells will be read. This is because it takes **M** time units to fill the buffer (the normalized rate at the source is **1**). When the previously

read cells are purged (**Mr** time units have elapsed), **(Mr)r = Mr^2** new cells will be read, and so on.

Therefore, before losing any unread cell, the total number of cells that can be read is:

$$Mr+Mr^2+Mr^3+...$$

This is an infinite geometric series that converges to:

$$Mr/(1-r)$$

Let **h** be the size of a frame, then by setting **M** to **h(1-r)/r** for a continuous throughput, we guarantee that a frame can be read and processed before any loss in the frame occurs.

However, some frames will still have to be dropped which is inevitable in the case of slow processing.

3.2.3 The Stream Unreliability

The stream unreliability **SU** lies in that seeking can not be performed beyond the bounds of the **SD** buffer. Of course, seeking becomes less important as the size of **SS** becomes larger, where at the limit, **SS** represents an infinite sequence of bytes, i.e. we have an infinite stream. More importantly, **SU** is imposed by the fact that the stream will not guarantee that all the bytes at **SS** will ever reach **SD**. For simplicity we assume that if at any time, a byte at **SS** is not delivered to **SD**, every byte that follows will not be delivered too, i.e. the stream fails at one point to continue its operation. This makes the model simple and limits the number of errors we have to deal with, and saves us the burden of worrying about retransmission of lost information, after all we are presenting a high level concept of a stream and not a communication protocol. Note that this does not mean that unreliable transmission protocols can not be used with the stream. From the stream perspective, as long as the source is still providing data (even if some data had been lost), the stream will not fail. It is only when the source stops providing information that the stream fails.

3.3 The Virtual Stream as a State Machine

In order to define the stream behavior just described, we introduce the following definitions: A stream is **Open** if the **SD** buffer is still being updated. A stream is **Closed** if the **SD** buffer becomes empty. Otherwise, a stream might be **Loaded**, meaning that all bytes have been transferred from **SS**, or **Broken**, meaning that the transfer from **SS** was incomplete. Obviously, in case of an infinite stream the **Loaded** state can not be reached; moreover, note how the stream failure is modeled by entering the **Broken** state.

A stream can receive commands from the user. These commands are the same as the commands used with a file descriptor namely *open*, *close*, *read*, and others. The following state machine [8] describes how the stream behaves. Edges represent user commands and circles denote the states of the stream as defined above. An edge with no description means that the transition can occur without any user command.

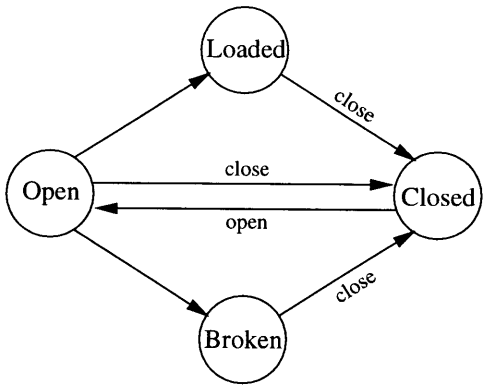

Figure 3.2: The virtual stream state machine.

Read, *write*, and *seek* commands are independently dealt with at the **SD** buffer level. For these commands, a **Loaded** stream and a **Broken** stream look the same since they have the same implied **SD** buffer state (not empty and not being updated), which is a reasonable model to think of.

3.4 The Virtual Stream Semantics

In this section we describe the semantics of the virtual stream. The semantics define how the virtual stream behaves upon receiving different user commands. The interface provides the following five main commands:

1. ***Open***: opens a stream.
2. ***Close***: closes a stream.
3. ***Read***: reads from a stream.
4. ***Write***: writes to a stream.
5. ***Seek***: seeks to a position in the stream.

The first two commands deal directly with the virtual stream state machine as illustrated in the previous state diagram.

The other commands interact with the **SD** buffer only without affecting the state of the virtual stream.

For every one of these commands we have both a blocking and a non-blocking implementation, except for seek which has only blocking atomic semantics. Later, we will show what we mean by "blocking atomic".

Note that for Memory-Only streams (like memory buffers) and file streams, only a blocking implementation makes sense; however, we will describe the semantics in the most general form since we are describing the virtual stream.

First we describe the semantics for the open command.

3.4.1 The Open Command

The *open* command establishes the stream connection. Recall from the definition of a virtual stream that it is a connection between a source and a destination. The *open* command does exactly that. In the blocking version of the *open* command, the operation

returns when the virtual stream becomes either **Loaded** or **Broken**. The return code represents the status of the operation, either a *success* or a *failure*.

A *success* code means that the connection was successfully established. A *failure* code means that the connection failed to be established.

In the non-blocking version of *open*, the operation returns as soon as the connection is established or fails to be established.

3.4.2 The Close Command

The *close* command closes the connection and empties the **SD** buffer. In the blocking version of the *close* command, the operation returns after the state of the virtual stream becomes **Closed** and all necessary clean-up is performed. The return code reflects whether the operation has been successful or not.

In case of *failure*, the stream enters the **Broken** state. The reason behind this is that the *close* command could fail at any point during the operation even after it destroys the connection to **SS**. Therefore, the virtual stream is no longer guaranteed to provide new data from the source. The **Broken** state is the one that best represents this stream state.

In the non-blocking version of *close*, the operation returns directly with success by setting the stream state to **Closed** even if the clean-up stage is not fully performed.

3.4.3 The Read Command

The *read* command returns a certain number of bytes starting at the current position in the **SD** buffer and then advances the current position. In the blocking version of the *read* command, the operation returns only when the exact specified number of bytes has been read or no more bytes can be read.

In the non-blocking version of *read*, the operation returns as soon as some number of bytes have managed to be read which might be less than the specified number. The return code supplies the number of bytes actually read.

3.4.4 The Write Command

The *write* command pushes a certain number of bytes into the **SD** buffer starting at the current position. The same semantics described for the *read* command apply here as well.

Note that for the *write* command, information is written on the **SD** buffer and not at the source. The reason behind this restriction emanates from the fact that a virtual stream has to provide an abstraction to all kinds of streams. Some streams are uni-directional by nature (like HTTP).

3.4.5 The Seek Command

The *seek* command updates the current position in the **SD** buffer. The *seek* command is blocking in the sense that no operation is allowed to start before the *seek* operation ends. This is useful in many cases. Consider for example the case of a scroll bar tracking the frames of a video stream. The scroll bar should not move beyond the region of available information.

On the other hand, the *seek* operation should not block for a long period of time whenever the information is not available because, for example, it is not desired that the user interface waiting for the *seek* command freezes. Therefore, unlike *read* and *write* operations where the current position in the **SD** buffer is updated progressively, *seek* is performed atomically. Therefore, the *seek* operation returns as soon as possible with one of two codes: *success* or *failure*.

In case of *success*, the *seek* operation updates the current position in the **SD** buffer as specified. In case of failure, the *seek* operation does not alter the current position in the **SD** buffer. This is why we describe the *seek* command to be blocking atomic, keeping in mind that it does not really block.

3.5 Formal Description of Stream Semantics

In describing formally [9] the virtual stream semantics, we adopt the following notation:

$$\mathbf{[code, s_2, e]=C[s_1, p]}$$

This means that the command **C** with optional parameter **p** returns with a code **code** when executed on a virtual stream with state $\mathbf{s_1}$ leaving the stream in a state $\mathbf{s_2}$ with a side effect **e** on the **SD** buffer.

For now, **e** is simply an integer reflecting how many cells the current position has to move in the buffer after the operation is carried out. In the future, **e** might include some caching actions performed on the **SD** buffer (Recall that the buffer is just a model for destination and is not necessarily an actual buffer).

The following two figures illustrate the blocking and non-blocking semantics respectively.

The symbol Φ represents the null command which can be considered to be present at the end of any command. This reflects the fact that the virtual stream might enter the **Loaded** or **Broken** state at any time.

In addition to these main commands, other commands are added to check for the stream state, retrieve the current position in the **SD** buffer, and extract some useful information about the stream source.

[success, Loaded | Broken, 0]=open[Closed]

[failure, s, 0]=open[s] s is any state

[success, Closed, 0]=close[Open | Loaded | Broken]

[failure, Broken, 0]=close[Open | Loaded | Broken]

[failure, Closed, 0]=close[Closed]

[m, Loaded | Broken, m]=read[s, n] m < n; s ≠ Closed

[n, s, n]=read[s, n] s ≠ Closed

[0, Closed, 0]=read[Closed, n]

[m, Loaded | Broken, m]=write[s, n] m < n; s ≠ Closed

[n, s, n]=write[s, n] s ≠ Closed

[0, Closed, 0]=write[Closed, n]

[success, s, n]=seek[s, n] s ≠ Closed

[failure, s, 0]=seek[s, n]

Figure 3.3: Virtual stream Blocking semantics.

[success, Open, 0]=open[Closed]

[failure, s, 0]=open[s] s is any state

[success, Closed, 0]=close[s] $s \neq$ Closed

[failure, Closed, 0]=close[Closed]

[m, s, m]=read[s, n] $m \leq n$

[m, s, m]=write[s, n] $m \leq n$

[success, Loaded, 0]=Φ[Open]

[success, Broken, 0]=Φ[Open]

Figure 3.4: Virtual stream Non-blocking semantics.

Chapter 4

The Virtual Streams Hierarchy: An Object-Oriented Design

4.1 Overview

Virtual streams are implemented as a hierarchy of streams that goes from the most general to the most specific. In other words, streams at higher levels in the hierarchy perform general operations, whereas streams at lower levels in the hierarchy perform more specific tasks. This paradigm of programming falls under the category of object-oriented programming.

A stream is regarded as an object. Software objects have *states* and *behaviors*. The state of an object is maintained by its *variables*. The behavior of an object is implemented by a set of *methods* which are functions used to change its variables and state [10]. An object at lower level in the hierarchy can *inherit* the states and behaviors of other objects at higher levels in the hierarchy and refine them.

The following sections will introduce object-oriented programming and the concepts of objects, classes, messages, and inheritance. This will provide enough background to understand the design of the virtual streams hierarchy.

4.2 Object-Oriented Programming

Object-oriented programming is a techniques for writing "good" programs for a set of problems. An object oriented programming language means a programming language that provides mechanisms that support the object-oriented style of programming well. An example of such a language is C++ [2].

But why is object-oriented programming a "good" programming style? Object-oriented programming provides better encapsulation and data abstraction through the idea of objects. An object contains a set of variables that determine its state and a set of methods (functions) that represent an interface to the outside-world (the behavior of the object). Therefore, all the variables and functions needed for a certain operation can be encapsulated inside one object. Data abstraction comes from the fact that a software object can represent any real-life object.

An example is the virtual stream. An object that contains all the variables and functions of a stream represent a data abstraction of the stream. Furthermore, this means that all the operations and states of the stream are encapsulated inside the stream object. The state of the stream is encoded by a set of variables and the behavior of the stream is represented by the interface to the outside-world.

Another advantage of object-oriented programming is the ability to separate between interface and implementation. All streams provide the same interface as seen before; however, every type of stream may have a different interpretation of a certain operation.

So, the reader can see how object-oriented programming represents a "good" programming style. More importantly, it is very well suited for our purpose of implementing a virtual stream, where a stream is merely described by a state and an interface to the outside-world.

The following sections clarify some of the important concepts in object-oriented programming.

4.2.1 Objects

An object can be represented in the following way [11]:

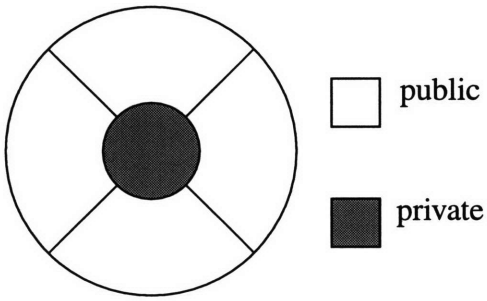

Figure 4.1: An object.

The shaded area represents all the information that is private to the object. This includes private variables as well as private methods. Private information is not accessible to the outside-world of the object. Public information, however, is, and it includes all public variables and methods. The virtual stream operations described in chapter 3 map most logically to public methods. On the other hand, private variables can describe the stream states. Nevertheless, some public methods can still be provided to retrieve private variables (like getting the current state of a stream).

4.2.2 Classes

Like types are to variables, classes are to objects. A class describes what an object should contain. It determines all its variables and methods. It also specifies which parts of the information are to be private and which are to be public.

As we can have many integers or reals in a program, we can also have many objects of the same class. This is why we say that an object is an instance of a certain class. We can have many instances of a class in a program.

4.2.3 Messages

Different objects communicate by sending messages [11].

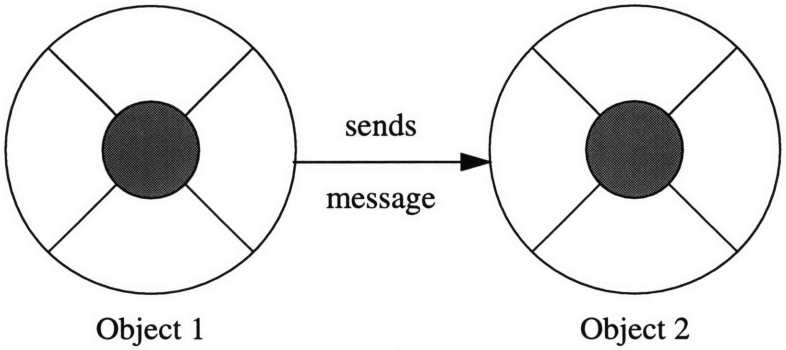

Figure 4.2: Sending messages between object.

As it can be seen from the above figure, object 1 sends a message to object 2 by calling one of its public methods. Hence, a message has to specify two components:

1. The object to which the message is directed.

2. The method that has to be performed by the specified object.

This is why, a method is usually called in the following way:

object.method()

or

pobject->method where pobject is a pointer to the object in question.

4.2.4 Inheritance

Inheritance is a very useful concept in object-oriented programming. when a class B inherits the attributes (states and behaviors) of a class A or is derived from A (or equiva-

lently, a class A is a base class of B), it means that B is an A in addition to what else it might be. More precisely, B is a *kind of* A and therefore, B is more specific than A [2]. An example of inheritance would be an HTTP stream class that inherits from a Network class. The Network class might contain information about the host name and address. By virtue of inheritance, the HTTP class will also contain this information. In addition, the HTTP class will have methods to access the network by using the host information from the Network class and to read incoming data according to the HTTP protocol. The following figure illustrates a typical inheritance hierarchy.

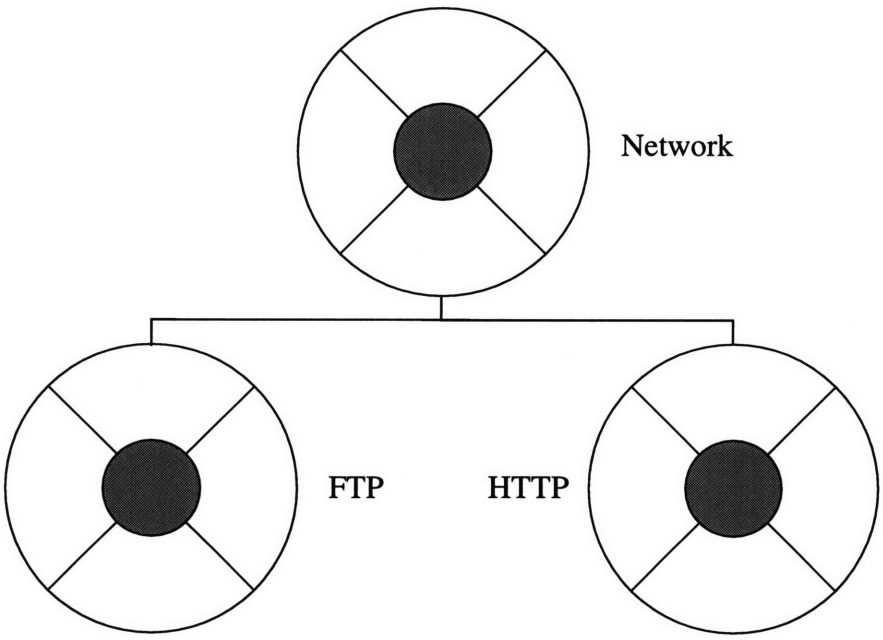

Figure 4.3: An inheritance hierarchy.

Note that the symbol for object is being used here to denote the class of that object. We often talk about objects and classes interchangeably when describing inheritance relationships.

Another important aspect of inheritance is the ability for derived classes to override methods defined in base classes. This is how derived classes specialize more the general operations in base classes. In the example above, FTP and HTTP are more specialized than Network.

For the purpose of calling methods, any derived class is a type of its base class. In other words, any object of a derived class can be used anywhere an object of the base class is expected. As an example, passing an object of class B to a function f that takes as its argument an object of class A is acceptable if B inherits from A.

An interesting scenario occurs if f calls a method of A that has been overridden in B. Since f expects a class A object, it will call the version of the method defined in class A. However, this defeats the whole purpose of separating the interface from the implementation. How can we make f know that what it has is a class B object in reality? Some languages (C++ for example) provide *virtual* methods. By making class A declare the method as *virtual*, the correct version of the method will be resolved at run-time and called.

A small variation to *virtual* methods exists. In C++, a class can declare a method to be *vritual* and set it to zero. By doing so, the class will become an *abstract* class and no object of that class can be instantiated. The whole purpose of this class is just the interface, i.e. in order to use this class, another non-abstract class has to be derived from it.

For more information about object-oriented programming, the reader can refer to: An Introduction to Object-Oriented Programming by Timothy Budd from Addison-Wesley Publishing Company.

Next, I will demonstrate the object-oriented design of the virtual streams.

4.3 The Virtual Streams Hierarchy

As described in chapter 1, in order to ensure uniformity of access for all kinds of streams,

we differentiate between streams and *streamSpecs*.

4.3.1 StreamSpecs

A *streamSpec* is simply a data structure that specifies the source of the stream.

As an example, a *fileSpec* might contain the path to where a file resides, the name of the file, and some flags to indicate whether the file should be open for read or write or both. A *networkSpec* might contain a server name, a protocol to be used to connect to the server, a path and a file name. A *streamSpec* can be considered as an abstraction of a source. This helps in treating different sources the same way simply by making all *streamSpecs* inherit from a *baseSpec* that provides the required interface.

Streams will then be constructed from their *streamSpecs*. As described in chapter 3, all streams provide a uniform interface including methods such as *open*, *close*, and *read*. The only difference is how the stream will interpret these commands. A converter, that we call the stream Manager, will be responsible for converting a *streamSpec* into a real stream that provides this interface.

4.3.2 The Hierarchy

The implementation uses a set of C++ classes that make up a hierarchy of streams derived from the base stream that responds to the above commands by a **null** operation (not doing anything). In terms of the state diagram described in chapter 3, all transitions are still valid; however, the class itself cannot be instantiated.

Parallel to this hierarchy is another hierarchy of *streamSpecs*. The following two figures illustrate the *streamSpecs* and streams hierarchies. The **SG** prefix stands for *Generic Stream*. Dark boxes represent classes that can be instantiated whereas white boxes represent abstract classes.

Different kinds of streams are derived from the base stream: **Blobs** are simple memory buffers and **IPCs** are inter-process communication pipes. These two kinds of streams are *Memory-Only* streams in the sense that no disk access is required.

Other streams contain files and network streams. The **net** stream is a general network stream whereas **ftp** and **http** are more specialized versions of the network stream.

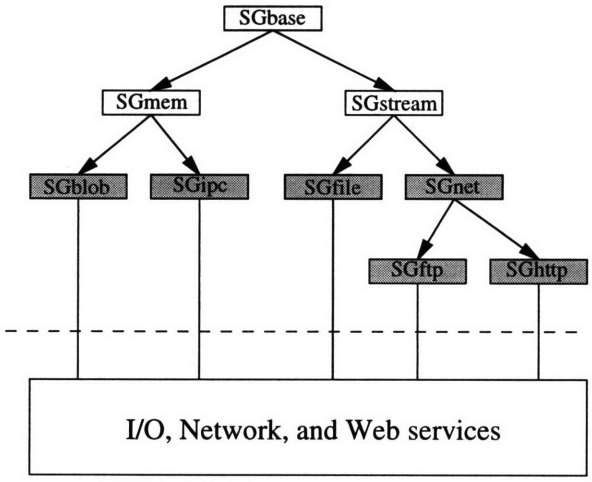

Figure 4.4: Virtual streams hierarchy.

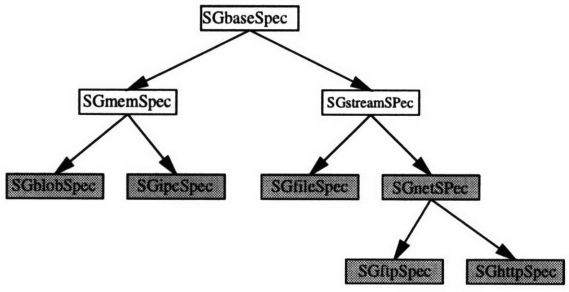

Figure 4.5: StreamSpecs hierarchy.

As it can be seen from the above two figures, *StreamSpecs* form a parallel hierarchy to streams. The only difference is that *StreamSpecs* do not interact directly with services provided by the system. They only describe the stream source.

4.3.3 The Stream Manager

The stream Manager converts a *streamSpec* into its functional stream as the following figure shows:

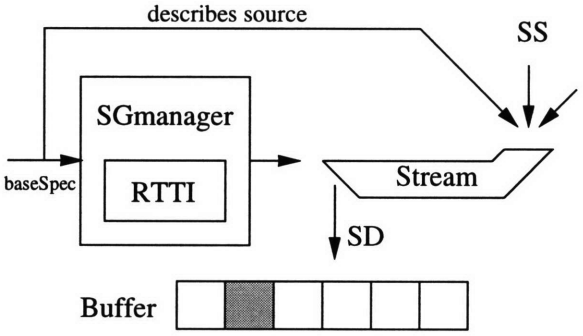

Figure 4.6: The stream Manager.

By virtue of the object-oriented approach, the stream Manager needs to communicate only with the interface of *baseSpecs* and base streams. Now the question is: How does the stream Manager know what the actual *streamSpec* was in order to create the corresponding stream if all of the *streamSpecs* are regarded as instances of the *baseSpec* class? Using a **RTTI** (Run-Time Type Identification) mechanism, the stream Manager determines the actual derived type of the *Spec* in the *streamSpecs* hierarchy and creates the corresponding stream in the streams hierarchy.

RTTI is interesting by itself because it illustrates the concepts of inheritance and virtual methods. **RTTI** is now supported by some compilers; however, one can implement his/her own **RTTI** mechanism. I will not discuss how to implement **RTTI** but I will illustrate the simplest form of an **RTTI** mechanism that relies on the idea of virtual methods.

Consider an **RTTI** abstract class. The class will contain one string and one virtual method set to zero (abstract class). Now consider any class for which we need the Run-Time Type Information. In an object-oriented language which allows multiple inheritance, we can make this class inherit from the **RTTI** class in addition to its base class. When an object of this class is constructed, the string will be set to the name of the class. This class will also implement the virtual method such that it returns the string. Since the method is virtual in the **RTTI** class, when an object of this class is passed as a base object, the right version of the method will be invoked and thus by using it we retrieve the correct string which is nothing but the name of the class.

The next section will illustrate how to use the virtual streams with object-oriented programming.

4.4 The Final Product

After I described the object-oriented design governing the virtual streams, I will illustrate how can the concepts be used practically. First, consider the following example of constructing a file stream in C++.

```
SGbase * file;
SGbaseSpec * spec = new SGfileSpec("fname");
file=SGmanager::StreamFromSpec(spec);
```

The following code reads 5 bytes from the same stream:

```
char a[5];
file->Open();
file->Read(5, a);
file->Close();
```

Any other kind of stream can be constructed and used in a similar way, thus providing the same interface and ensuring uniformity of data access across all streams.

Here's an example of constructing a network stream using the HTTP protocol and reading from it:

```
char a[100];
SGbase * net;
SGbaseSpec * spec = new SGHTTPSpec("web.mit.edu/saad/Public/saad.html");
net=SGmanager::StreamFromSpec(spec);
net->Open();
net->Read(100, a);
net->Close();
```

The reader can compare both codes to see how close they are in structure and semantics. This is also an appropriate point to go back to chapter 1 and compare these programs to the suggested pseudocodes.

The presence of abstract classes in the virtual streams hierarchy provides an abstraction of the idea of a stream. Consider the following example where a function accepts a stream as its argument and reads some data from it:

```
int Reader(SGstream * s) {
char buf[10];
s->Read(10,buf);
if (buf[3]=='a') return 1;
else return 0;
}
```

The presence of the **SGstream** class for instance provides a way of passing streams as arguments to functions without worrying about the nature of these streams. Note that we could have been even more general and specified an **SGbase** as an argument to the function instead.

As mentioned in chapter 1, another goal is to provide a cross-platform implementation of the virtual streams. We achieve this by using platform-independent system services such as standard memory and file I/O operations, widely available network services such

as FTP and HTTP, and third-party libraries available for different platforms such as the W3C library, which is a general purpose Web API written in C and will be discussed in the next chapter on the network stream.

Chapter 5

The Network Stream: A Proof of Concept

5.1 Overview

This chapter describes in detail the implementation of the network stream as an illustrative example of a virtual stream. The network stream is composed of three different modules:

1. The command interface.
2. A *gluing* part that relates and synchronizes the command interface with the actual streaming.
3. The actual streaming of data over the network, which runs in a separate thread.

The command interface is as described before in chapter 3. It consists of the set of operations that a stream needs to perform. The synchronization module is used to coordinate the operations of the network stream with the actual streaming of data since these two components of the stream has to run in parallel for the stream not to block on a request. The streaming of data is performed using the W3C Web API library.

Therefore, in order to go into the details of the network stream, I need to describe parallel computing and some models of synchronization, and explain how the W3C library works.

5.2 Parallel Computation

Parallel computation is often used for the purpose of speeding up operations. The most common configuration in parallel computation is when different processors operate simultaneously on independent sub-tasks. The final result would depend on the sub-tasks performed.

However, it is not always possible to divide a task into sub-tasks that are completely independent. A process performing a sub-task might need some information computed by another process in order to proceed with its own part of the computation. The need for synchronization arises whenever there are parallel processes [12].

5.2.1 Synchronization Models

Different synchronization models can be used depending on the nature of computation performed. Here are some examples [12]:

1. Forks and Joins:

In parallel computation, a parallel process may want to wait until several events have occurred [12]. For example, in the figure below, a task is split into n processes, P_1 to P_n, the results of which are then joined together.

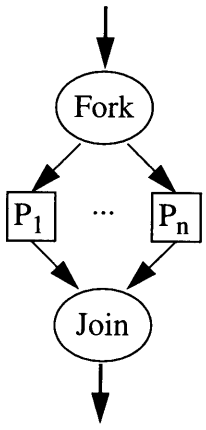

Figure 5.1: Forks and Joins.

2. Producer-Consumer:

A consumer process must wait until the producer process has produced the required data [12].

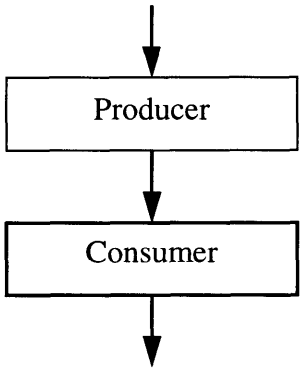

Figure 5.2: Producer-Consumer.

3. Exclusive use of resources:

This synchronization model is used to ensure that only one process uses a shared resource at a given time [12].

This is the model of synchronization used in the network stream. Recall from the virtual stream model that we represent the stream destination **SD** by a buffer. This buffer (which is the resource in question) is accessed by two different processes: the process running the command interface from the user part, and the streaming process from the W3C library part. In some situations, we need to make sure that only one of theses processes uses **SD** at a given time to ensure correctness. An example of a situation where failure to synchronize can create errors is the following: the user closes the stream and destroys the **SD** buffer while the streamer is still accessing it. The **SD** buffer in this case is the shared resource. The streamer has to be notified that the **SD** buffer is about to be destroyed so that

it stops streaming before the buffer is actually destroyed. Therefore, synchronization is needed.

5.2.2 Implementing Synchronization

In this section, I will only address the third model of synchronization presented above. We often refer to this model as Critical Sections and Mutual Exclusion: Two processes need to exclude each other from some critical code (accessing the resource) [13].

As a first attempt to implement mutual exclusion, consider the following protocol based on two globally shared variables c1 and c2. Initially, both c1 and c2 are set to 0.

```
Process 1                          Process 2

    c1=1;                              c2=1;
L:  if c2=1 then goto L            L:  if c1=1 then goto L
    <critical section>                 <critical section>
    c1=0;                          c2=0;
```

Figure 5.3: Mutual exclusion using two shared variables.

It might seem that this protocol works; however it is wrong. A careful look at the protocol reveals that process 1 and 2 may both wait forever. Generally, it is very difficult to design correct protocols for mutual exclusion [12]. This situation is referred to as a deadlock.

To avoid the situation described above, let a process give up reservation (i.e. process 1 sets c1 to 0) while waiting.

```
Process 1                          Process 2
L: c1=1;                           L: c2=1;
   if c2=1 then                       if c1=1 then
      {c1=0;goto L}                      {c2=0;goto L}
   <critical section>                 <critical section>
   c1=0;                           c2=0;
```

Figure 5.4: Revised protocol for Mutual Exclusion.

Deadlock is still possible (because process 1 and 2 may reserve and give up reservation at the same time repeatedly) but much less likely. Another problem with this protocol is that an unlucky process may never get to enter the critical region and always the same process succeeds to enter [12]. Here is a correct protocol for mutual exclusion by T. Decker. It is based on three shared variables c1, c2, and turn. Initially both c1 and c2 are set to 0.

```
Process 1                          Process 2
   c1=1;                              c2=1;
   turn=1;                            turn=2;
L: if c2=1 & turn=1                L: if c1=1 & turn=2
      then goto L;                       then goto L;
   <critical section>                 <critical section>
   c1=0;                              c2=0;
```

Figure 5.5: Correct protocol for Mutual Exclusion.

In the above protocol, turn=i ensures that only process i can wait. The generalized solution for n processes is even more tricky.

The reader can see how difficult is to implement synchronization in software. This is why some hardware support has been added to facilitate the task.

5.2.3 Semaphores and Events

Dijkstra (1965) proposed semaphores, two new atomic primitive operations that considerably simplified the programming of synchronization problems.

P and V operations operate on non-negative integer variables called semaphores in the following way:

P(s): if s>0 decrements s by 1; otherwise waits.

V(s): increments s by 1 and wakes up one of the waiting processes.

The crucial aspect is that all actions on semaphores are done atomically. Here's an example of mutual exclusion using semaphores:

Process 1	**Process 2**
P(s);	**P(s);**
<critical section>	**<critical section>**
V(s);	**V(s);**

Figure 5.6: Mutual exclusion using semaphores.

The advantage of semaphores is that hardware support has been provided in order to implement P and V operations easily. The description of how such support is possible is beyond the scope of this introduction, and the reader can refer to [13].

Events are similar to semaphore. Events are useful in a number of situations to notify a waiting process that something of relevance has occurred. A process can wait for an event and proceed when the event is triggered by another process [14].

In the network stream, I use both semaphores and events. Semaphores will keep track of the number of active network streams for the stream Manager to know when to perform cleanup. Events will guarantee that the command interface will not access wrong information while the streamer is initializing the stream and that the streamer will not access wrong information while the command interface is cleaning up the stream (upon closing for example). Mutual exclusion is therefore implemented using events.

5.3 The W3C Library

The W3C library is a general code base that can be used as a basis for building a large variety of World-Wide Web applications. Its main purpose is to provide services to transmit data objects rendered in many different media types either to or from a remote server using the most common Internet access methods or the local file system [15].

The W3C library provides standard C reference implementations of those specifications and is especially designed to be used on a large set of different platforms. Even though plain C does not support an object-oriented model, many of the data structures in the library are derived from the class notation. This leads to situations where forced type casting is required in order to use a reference to a derived class where a base class is expected [15].

The forced type casting problem and inheritance in general would be solved if an object-oriented programming language was to be used instead of C, but the current standardization and deployment level of object-oriented languages in general would imply that a part of the portability would get lost in the transition.

As in the case with ActiveMovie described in chapter 2, W3C library uses the concept of cascaded filters. Data propagates from the source through all the filters until it reaches its destination. The configuration of filters can be set by the user.

A network connection in the W3C library is modeled in the following way:

Before the network connection is established, a list of 'Before Filters' are executed in sequence. Every filter is associated with a response code. Moreover, every filter forwards a status code to the next filter. A filter executes only if its response code matches with the forwarded status code [16].

After going through the sequence of 'Before Filters', the connection is established and a 'Converter' (or more than one) is used once to modify the data (like extracting MIME Header). After that, data is transferred to its destination by an 'Output Stream'. If the transfer is complete or an error occurs during the transfer, the connection is closed and a sequence of 'After Filters' is executed in the same way 'Before Filters' are [16]. The following figure depicts the model:

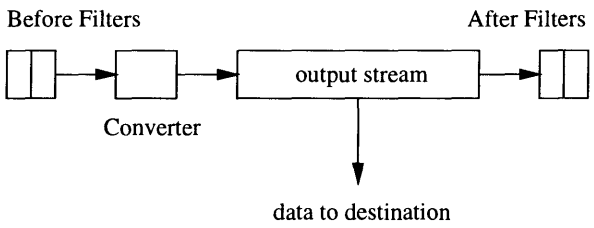

Figure 5.7: A network connection in W3C library.

5.4 The Network Stream as a Proof of Concept

In this section I will describe the implementation of the network stream. The network stream is constructed by providing the stream Manager with either an *SGnetSpec*, an *SGftpSpec*, or an *SGhttpSpec*.

Both *SGftpSpec* and *SGhttpSpec* are specialized versions of *SGnetSpec* where the protocol to be used is defined a priori. The source of a network stream is a **URL** (Uniform Resource Locator). The **URL** is encoded in the *Spec* used for creating the stream.

Since *SGftpSpec* and *SGhttpSpec* both inherit from *SGnetSpec*, the **URL** is actually contained in the latter.

The synchronization model based on events uses a synchronization object which is a data structure that contains the following:

1. The event name.

2. An event handle (operating system level).

3. The event code.

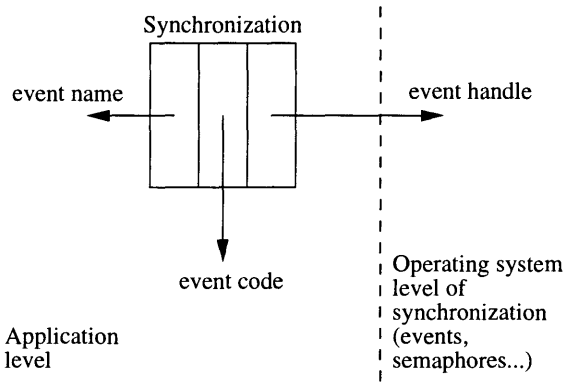

Figure 5.8: The synchronization data structure.

When an event occurs, it is said to be signaled. Two kinds of functions deal with events: Functions that signal events and functions that wait for events to be signaled.

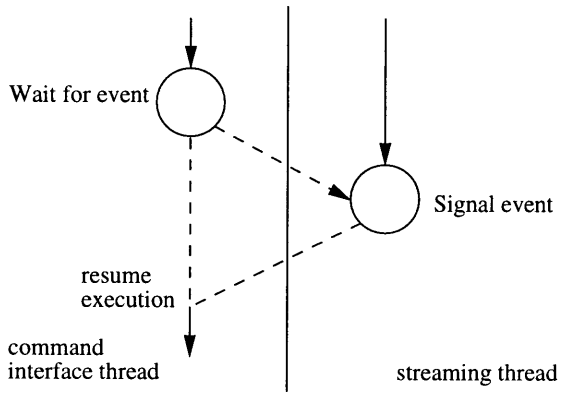

Figure 5.9: Synchronization based on events.

All synchronizations are done through the Stream Manager. The synchronization interface can be described as follows:

sync=NEW_SYNC()

returns a new synchronization with a fresh event name.

DELETE_SYNC(sync)

deletes the synchronization **sync**.

SIGNAL_EVENT(sync, code)

signals the synchronization event and sets its code to **code**.

WAIT_FOR_EVENT(sync)

waits until the synchronization event is signaled and returns the event code.

We can think of synchronizations as tokens accessible by all processes. The event name is used internally as a means for the different processes to communicate since in the

implementation, processes can access events only through their names. It is not a critical part of the model.

The actual streaming of data is done using the W3C library as mentioned before. The implementation of the network stream makes use of the following:

1. One Converter.

2. Two After Filters.

3. A File Output Stream.

The Converter used is the identity Converter which does not alter the incoming data. It is there just for the sake of synchronization as it will be shown later on.

The two After Filters are basically used for the same purpose; however, they provide extra functionality in that they represent the **null** command Φ described in chapter 3.

The Output Stream is used to store incoming data into a temporary file for later access using the command interface. So, in terms of the stream model developed in chapter 3, the **SD** buffer is actually a file. This file will eventually have the size of the original file at the source because I found it useful to implement file semantics for the network stream. The complexity of *read*, *write* and *seek* operations is dealt with at the file I/O level by the operating system.

In the current implementation of the network stream, only two synchronization structures are used: **openSync** and **closeSync**. When the network stream is first constructed, both of these synchronization structures are created by using **NEW_SYNC()**. Upon destruction, the two synchronization structures are deleted using **DELETE_SYNC()**.

In the remainder of this section, I will describe the non-blocking version of the network stream with blocking semantics for the *close* command only (to make sure that the clean-up takes place before we exit).

When *open* is called to open the network connection, the *open* command initializes all necessary parts of the network stream and waits by using:

code=WAIT_FOR_EVENT(openSync)

The job of the converter is actually to signal this event with a success code, so it performs the following:

SIGNAL_EVENT(openSync, success)

Just by reaching the converter we ensure that the network connection was established successfully and thus resume execution of the *open* operation.

What if the converter is never entered i.e. the connection failed? In this case one of the After Filters (called *ErrorFilter*) will respond. The response code for *ErrorFilter* is HT_ERROR which is a generic error code for all errors in W3C. So if *ErrorFilter* performs the following

SIGNAL_EVENT(openSync, failure)

then the *ErrorFilter* will also set the state of the stream to **Broken**. This represents the following operation in the semantics:

[success, Broken, 0]=Φ[Open]

In both cases, the *open* operation will resume execution by setting the network state to either **Open** or **Closed** depending on the **openSync** event code.

Another detail worth mentioning is that *ErrorFilter* will also signal the **closeSync** synchronization event for reasons that will become clear in a while.

If everything works fine, the stream will start storing incoming data into a temporary file.

When the user issues the command to close the stream, the following happens:

The *close* command will first check the state of the network stream. If it is **Closed**, then the operation returns directly with a *failure* code, otherwise it sets the state to **Broken** and waits by performing:

code=WAIT_FOR_EVENT(closeSync)

This is needed because the *close* operation can not perform its clean-up before making sure that the streaming process is not using some shared data (like the file handle for example). The Output Stream will continuously check for the state of the stream and whenever it finds that it is **Broken**, it will stop by returning the code HT_ERROR. This is why *ErrorFilter* also performs the following:

SIGNAL_EVENT(closeSync, success)

After signaling this event, the *close* operation can resume its execution and clean-up. If it ever fails in the clean-up process, the state of the network will remain **Broken**, otherwise it will be set to **Closed**.

So far, nothing has been said about the second After Filter. The second After Filter (called *OkFilter*) responds to the HT_OK code which is a generic success code in W3C library. The *OkFilter* will set the state of the network stream to **Loaded** which represents the following operation in the semantics:

[success, Loaded, 0]=Φ[Open]

By entering the *OkFilter*, we ensure that the transfer has completed. In order to be able to close the connection, the OkFilter has to perform the following:

SIGNAL_EVENT(closeSync, success)

Semaphores are used within the stream Manager to keep track of all the active network streams. The idea is that the stream Manager needs to perform some clean-up upon exiting the main application, and there is no way to tell whether there exist some network streams

that are still active. Therefore, the stream Manager keeps a count that is accessible to all streams. When a stream is open, the stream increments the count. When a stream is closed, the stream decrements the count. Since the streams represent different processes that are accessing a common piece of code, mutual exclusion via semaphores as described in figure 7 in necessary.

5.5 Applications of the Network Stream

As an illustrative application, I implemented a streamer DLL that uses the network stream and coupled it with the NETTOOB MCI driver for MPEG. One problem with introducing streaming in NETTOOB is that the driver assumes end of data (**EOD**) whenever the DLL provides it with less than the requested number of bytes [6]; this makes it difficult to differentiate between a slow network connection and an actual **EOD**. This can be solved by using the blocking *read* operation or the non-blocking one in conjunction with some test functions that retrieve the state of the network stream. The following example illustrates the idea:

```
int i=0;
while ((i!=count) && (net->State()==Open))
    i+=net->Read(count-i, &buf[i]);
```

The C++ code for the NETTOOB driver can be found in Appendix A.

Chapter 6

Conclusion

6.1 Summary

I presented virtual streams as a more appropriate model for general I/O operations. Virtual streams are implemented on top of basic I/O and Web services. The implementation uses a object oriented approach, and hence is done in C++.

Virtual streams form a hierarchy of classes derived from a base class that represents the base stream. The base stream provides the necessary interface which is generic for all other streams by virtue of inheritance. Therefore, the goal of achieving an interface for uniform data access is accomplished.

Platform-independence is obtained by using cross-platform services such as the HTTP and FTP protocols, and the W3C library, a Web API C based tool.

The semantics of virtual stream were also described in detail, with special attention to seeking which is the most subtle property of a stream. Virtual streams use the notion of a blocking atomic seek, the result of which is based on information availability. It blocks only to determine whether information is available or not and then returns; however, it does not block until the information becomes available for reasons described in chapter 3.

I described the network stream as an illustrative example of all the concepts behind virtual streams. The network stream uses multithreaded stream requests in order not to block on every request. Synchronization of the different threads or processes is therefore necessary. The network stream makes use of the W3C library which I also describe with emphasis on the connection model and the concept of filters.

6.2 Evaluation

Virtual streams provide a convenient and uniform way of accessing data from different sources. As described in chapter 5, the network stream has been used successfully in streaming MPEG.

Another practical application currently under investigation is how to modify third-party libraries to make use of the virtual streams instead of the basic system I/O services. An easy solution is to provide what we call the I/O table. The I/O table is a simple data structure that maps indices to SGbase objects. All *read*, *write*, and *seek* operations will be replaced by similar ones that are simple redirections to the *read*, *write*, and *seek* operations of the SGbase object in the table. The SGbase object will be accessed through its index in the table which is equivalent to a UNIX™ file descriptor. In fact this is what we are currently doing for the Multimedia Toolkit in **AthenaMuse®2**, a Multimedia Authoring System that we have developed.

6.3 Future Work

In addition to implementing **IPCs** as part of the virtual streams and extending them to networked **IPCs**, and providing caching to speed up file and network streams, we are currently looking at problems of stream states as a response to *write* operations. The idea is that *write* operations must always be carried out (even if the *seek* operation fails to reach

the specified position for writing) since all the data needed for a *write* operation to take place is available.

The information to be written can be stored in a separate buffer called the **Bypass** buffer. The **Bypass** buffer bypasses the **SD** buffer whenever a *read* operation is performed by providing its data instead. This is a way of providing pending writes. Pending reads will also be considered but they are more tricky. All of this will imply using additional stream states and semantics to handle the new buffer and ensure correctness among different data accesses.

Other work lies in investigating better stream models and semantics as well as adding more synchronization and priority constructs in the stream command interface.

Appendix A

NETTOOB Stream Driver Code

```
#define STREAM_STOP 1
#define MOVIE "web.mit.edu/saad/Public/example.mpg
#include <SGmanager.h>

SGbase * net;
SGbaseSpec * spec;

LPSTR WINAPI STREAM_Init(long size) {
   SGmgr::Init();
   spec=new SGnetworkSpec("http", MOVIE, eFALSE, eTRUE);
   //specifies protocol, movie file, blocking?, seekable?
   SGmgr::StreamFromSpec(spec, net);
   SGstatus status=net->Open();
   return "TEST";
}

long WINAPI STREAM_Read(LPSTR buf, long size) {
   int i=0;
   while ((i!=count) && (net->State()==Open))
   i+=net->Read(count-i, &buf[i]);
   return i;
}

long WINAPI STREAM_Seek(long offset) {
   SGstatus::Estatus status=net->Seek(offset,SEEK_CUR);
   if (status==SGstatus::eError) return 0;
   else return offset;
}

int WINAPI STREAM_Notify(UINT code) {
   if (code==STREAM_STOP) {
   net->Close();
   SGmgr::Terminate();
   return 0;
   }
   else return -1;
}
```

References

[1] Rochkind, M. *Advanced Unix Programming*. Prentice Hall 1985.

[2] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1991.

[3] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. RTP: *A Transport Protocol for Real-Time Applications*. Internet Engineering Task Force, Audio-Video Transport Working Group, Jan. 96. RFC-1889.

[4] Rao, A., Netscape Communications, Lamphier, R. Progressive Networks. *RTSP: Real-Time Streaming Protocol*. Internet Engineering Task Force, Nov. 1996. Internet Draft.

[5] Microsoft. *Interactive Media Technologies*. Available from http:// www.microsoft.com/imedia/.

[6] Duplexx Software. *NETTOOB* Computer Software. Duplexx Software Inc., 1996. Available from http://www.duplexx.com.

[7] Progressive Networks. *Real Media Architecture*. Available from http://www.realaudio.com/prognet/rm.

[8] Sipser, M. *Non-deterministic Finite Automata*. Introduction to the Theory of Computation. PWS Publishing Company and International Thomson Publishing Company, 1996.

[9] Gifford, D. K., Furbak, F., Reistad, B. *Applied Semantics of Programming Languages*. Draft of Sept. 1995.

[10] Mary, C., Kathy, W. *The Java Tutorial*. Addison-Wesley 1996.

[11] Shehayeb, F. Object-Oeiented Programming. American University of Beirut, course notes.

[12] Arvind. Computer Architecture. Laboratory for Computer Science, MIT, course notes.

[13] Hennessy, H., Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kofmann, 1996.

[14] Microsoft. *Win32 Programmer's Reference*. Volume 2. Microsoft Press 1993.

[15] Frystyk, H. N. *The W3C Reference Library*. Available from http://www.w3c.org/pub/WWW/library.

[16] Frystyk, H. N. *The W3C Architecture*. Available from http://www.w3c.org/pub/WWW/library.

[17] Mneimneh, S., Bazzi, I., Morcrette, C. *Generalized Data Stream Interface*. ATIRP first annual technical conference, Maryland, 1997.