Abstraction Models at System Level for Networked Interactive Multimedia Scripting

by

Jimmy Chi-Ming Lai

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author ....
Department of Electrical Engineering and Computer Science
May 30, 1995

Certified by ....
Shaoul Ezekiel
Professor, Department of Electrical Engineering and Computer Science;
Director, Center for Advanced Engineering Studies
Thesis Supervisor

Accepted by ....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

[illegible]

Abstraction Models at System Level for Networked Interactive Multimedia Scripting

by

Jimmy Chi-Ming Lai

Submitted to the
Department of Electrical Engineering and Computer Science

May 30, 1995

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

With the growth of the Web has come a desire for greater interactivity. Unfortunately, the Web currently suffers from some limitations with respect to four parameters for characterizing networked interactive multimedia: Data flow, Spatial and Temporal Synchronization, Media Integration, and User Interactivity.

This thesis extends the ScriptX™ multimedia development platform by building programming abstractions to support desired characteristics of networked, interactive multimedia. The Data Flow Abstraction Model provides network extensions to support data prefetching in the background. The Temporal Synchronization Abstraction Model enhances ScriptX's support in non-linear synchronization of media playback. The Text-based Media Abstraction Model enables convenient creation of both static and dynamic text media.

With the abstraction models proposed, the extended ScriptX platform has demonstrated to satisfy all of the desired characteristics of networked interactive multimedia.

Thesis Supervisor: Shaoul Ezekiel
Title: Professor, Department of Electrical Engineering and Computer Science;
Director, Center for Advanced Engineering Studies

Acknowledgments

I would like to thank:

Professor Shaoul Ezekiel, my thesis supervisor, for his guidance and support to my research work.
Thomas Y. Lee, my wonderful reader, for his valuable suggestions and patience in proofreading my thesis throughout, especially during the final days.
Kip and Jonathan, for their originality of some of the ideas in this thesis.

I am also in debt to the following people, whose company has immensely enriched my life, both at MIT and *beyond.*

To Dad and Mom, Ah Mui and Dai Lo, for their love and support throughout these twenty-two years.

To Rebecca, for being a special person in my life.

To Brian and Felix, for their brotherly love and "ton dan" with me.

To Alex, Chalee, Daricha, Keileung, Phoebe, Somsak, Tomlee, for sharing my walks in MIT since freshman year.

To Brian, Felix, KK, Mawlo, Ngan, pchan, Yip Tao&Flora, my cell group members, for their important prayers and support.

To Chung Ma, pchanita, Jenny, Perry&Peggy, Phoebe and Vivian, for their "soup-water" that sustained my long hours in the lab.

To Brian, David Ma, Joycelyn, Kin, King, Mawlo and Vivian, my "same road people", for walking together to find and grasp the right priorities in life.

To Ah Shun, B-boy, Fui Hung, Owl and Shrimp, who are far away, yet whose regards have been great encouragement in times.

To Kelly and Philee, who kindly offered to help when they consistently found me in lab around 4am.

To King, Jerome, Joycelyn, Richard and SunMan, the "thesis-fryers", for their accompany on athena in the odd hours of the day.

Last but not least, to the Lord Jesus Christ, who has brought these lovely people in my life and made all of the above a reality.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Web is growing at an explosive rate[1]; it has evolved into a prototype of the National Information Infrastructure to come [2]. The Web is becoming a pervasive means for people to connect to and publish information on the Internet. The Web offers a likely medium for delivering networked interactive multimedia applications because of its enormous, easily accessible repository of multimedia content. This multimedia content could be made available for retrieval and integration into end-users' applications. The Web also offers an open architecture that permits developers to continually extend and integrate the work of others. Despite these advantages, the Web is limited in its ability to provide "truly" interactive multimedia.

In general, the composition and presentation of networked interactive multimedia applications (NIMA) involves at least four essential system components. The *Data Flow Component* describes how the data composing the application is delivered on the network to the user, and how the data is managed locally. The *Synchronization Component* deals with both the spatial and temporal organizations of the media elements in the presentation. The *Media Integration Component* describes the types of different media elements in the presentation and the processes through which they are prepared and rendered. The *User Interactivity Component* describes how the users' inputs are accepted and handled to manipulate the application in an interactive way.

1. The latest Internet statistics survey [1] has it that the total number of Internet hosts has passed 4.851 million by January 1995 and total users amount to at least 30 million. Worth noting is that "WWW-named host computers now constitute the most numerous on the Internet." and it brings forth "the largest jump in the recent history of the Internet with a 26% growth rate for 4th Quarter 1994."

These four system components may be used as assessment criteria to determine the appropriateness of a development / application platform and a delivery medium to author and deliver NIMA. This thesis attempts to address elements of these four components critical to developing and using NIMA.

1.1 Desired Capability for Networked Interactive Multimedia

1.1.1 Data Flow Component: Pre-fetching Model

Due to network bandwidth and local storage limitations, the "download-and-play" model for network distribution of long multimedia presentations is generally not practical. Downloading is expensive, both in time and space. Worse yet, if the user disliked the application, the cost of downloading and waiting has already been incurred.

Pre-fetching offers a possible remedy. Media-rich multimedia presentations generally consists of pre-packaged, individual media objects that are scheduled for rendering at different times. A pre-fetching model permits a multimedia presentation to begin playing with only a part of the entire data, and then pre-fetch subsequent objects during the presentation just before the objects are needed. The model should ensure that the foreground presentation is not affected by the fetching operation. Pre-fetching therefore provides for more efficient usage of network bandwidth and local storage since, at any instance, only a fragment of data is required. The model also provides some flexibility for users to "peek" into the presentation.

1.1.2 Synchronization Component: Spatial and Temporal Orchestration

- **Spatial Orchestration**

A lively multimedia presentation requires media elements to be positioned appropriately on the screen. Therefore, having the ability to position any object anywhere in the presentation space (a window in most cases) is indispensable. Common scenarios further require objects to be draggable, scalable, or to be able to bounce around on the screen.

• Temporal Orchestration

In a movie, the three elements of text, video, and sound should be synchronized. For instance, lip motion should match the audio track. There are two notions of temporal synchronization in a media-rich multimedia presentation: *linear synchronization* and *non-linear synchronization.* In linear synchronization, different media objects (e.g. an audio and a video track) are rendered at a constant rate. For example, playing a sound clip 2 seconds after the presentation starts, a video 5 seconds after, and another sound clip 10 seconds after. The main function of linear synchronization is to synchronize the start times for playing back objects throughout a presentation.

In a non-linear synchronization model, media playback is rendered at rates that change over time. A common example is a moving text stream whose scheduled playback involves flowing out continuously at different rates, pauses, and jumps to render new words at each jump.

1.1.3 Media Component: Common Media and Special Text-based Media

In a media-rich multimedia presentation, media types such as text, graphics, audio, video, and animations are commonly required. However, although rich media enables a lively multimedia presentation, network bandwidth and local storage are always of concern. A 15 minute video clip encoded in MPEG-I [20] at 1.5Mbits/sec takes at least 15 min. to transfer over a T1 line (1.5Mbits/sec), and requires 169M of disk storage. Many existing connections on the Internet are 64kbps leased lines or slower -- at least 23 times slower than a T1 line. A large population of users use dialup services to connect to the Internet through online services like AOL (American Online). Dial-in users are often limited to 28.8 kbps (or slower) modems.

In a networked environment where bandwidth and local storage are concerns, text-based media, though less aesthetic, provides an efficient means for content delivery. Moreover, text-based media could still be attractive. Text can convey a great deal of information. Text can enrich a presentation if it has been richly formatted and is processed by the application. For example, text can be formatted in different colors, sizes or fonts. Text can be dynamically rendered to flow across the screen, or to flash or bounce.

1.1.4 User Interactivity Component: Common Capability and Minimal Latency

User interactivity has to do with providing the means for users to directly manipulate objects within a presentation. Capability requirements vary greatly depending on the type of application. As a minimum, the application platform should provide control mechanisms that are sensitive to user input events, such as mouse moves, mouse position detections, and key presses.

To achieve a "real-time" experience, minimizing the latency in serving a user request is critical. Applications such as interactive adventure games have extreme service latency requirements. Minimizing service latency in processing user inputs generally requires client-side processing. If the interaction is performed by server-side processing, network delay compounds the latency.

1.2 Assessing Interactive Multimedia on the Web

Unfortunately, as a prototype of the NII, the Web cannot meet the criteria specified above for evaluating NIMA.

The Web is based on several interrelated standards, most notably the Hypertext Mark-up Language (HTML) and the Hypertext Transfer Protocol (HTTP). HTTP is the transport protocol used to communicate between Web servers and Web clients. HTML is a mark-up "language" that is used to represent the majority of Web documents. Currently, the Web is oriented towards relatively simple static documents. HTML is very limited in its expressive power, even in terms of page layout of static documents. The infrastructure for more dynamic web-based applications apparently does not exist, although the increasing use of forms and server-side CGI (Common Gateway Interface) scripts on the Web reflects the demand for more interactive, networked multimedia applications. Given the list of desired characteristics in section 1.1 for networked interactive multimedia, severe limitations are found with the current Web.

1.2.1 *Data Flow Component*

The Web only supports pre-loading on a per-document scale and only for text and graphics. Current browsers delay the fetching of inlined graphics within an HTML document, allowing the text portion of the document to be displayed and read by the users first.

Specific Web browsers do support pre-fetching of HTML documents. For instance, Netscape Communications Corporation has implemented a client pull mechanism with its Netscape Web browser that allows a document to be marked with a URL to load after a given number of seconds[4]. Client-pull enables pre-fetching of HTML documents within a presentation. There is no notion of pre-fetching in background mode because an HTML document is presented as static text in the foreground.

True pre-fetching requires that each piece of data in the presentation be able to be prefetched as an independent object and fetched just before use. The Web does not support the prefetching of external media data like video and audio data without user intervention.

1.2.2 *Synchronization Component*

The needs for both spatial and temporal organization in a multimedia presentation are poorly met by the Web. For spatial organization, HTML is extremely limited in its expressive range, even in terms of page layout for static documents. Features such as tables and multi-column documents, which available, are not yet part of the HTML standard. It is possible to somewhat position objects within the browser window through HTML but the ability commands, to drag, scale, or move objects within a static HTML document is impossible.

For temporal synchronization, common Web browsers only provide minimal, coarse-grained synchronization of media-playback. Media is either rendered in the order that users click on links or waits for control signals from spawned external viewers after the media is cached locally. There is no support for "orchestrated" presentations where the temporal relationships are not only explicitly formulated, but also detailed with great precisions as with synchronizing voice and annotated text.

1.2.3 *Media Component*

All the common media types can be rendered currently by Web browsers either locally in the HTML documents (text, graphics) or with help from externally spawned helper applications (audio, video, etc.). For formatted text-based media, HTML is the only standard.

1.2.4 *User Interactivity Component*

The Web currently handles simple user interactions by extending the point-and-click model used to navigate the Web or retrieve data. Fill-in forms are used to accept textual input from users for applications like surveys or online shopping, etc. More advanced interaction is achieved through the use of server-side CGI scripts [5] to generate HTML documents on the fly (e.g. to report real-time stock exchange information) or to redirect users to another URL.

The Web's model of interactivity is server-oriented, namely, in using the fill-in forms and CGI scripts. This has at least two disadvantages. First, the server can be easily overloaded when there are a large number of clients to be served. Second, server-side processing introduces extra delays in service latency compared to client-side processing because the processed result has to be delivered over the network.

Realizing the usefulness of interactive multimedia on the Web and the limitations presented by lack of client-side processing, many different proposals have been forwarded for client-side extensions. The ideas can be classified into several categories:

1. *Client Push/ Server Push*, from The Netscape Communications Corporation [4],
2. *External Programming Support in Browsers* -- such as the NCSA Mosaic CCI (Common Client Interface) [6], Spyglass SDI (Software Development Interface) [7] and Netscape 1.1 OLE2 Automation [8].
3. *Scripting*, namely, CCITcl [17] derived from the combination of SafeTcl [16] and the NCSA Mosaic CCI [6],
4. *Virtual Machine*, namely, the Java project at Sun Microsystems [18]. Virtual machine is a software interpreter for pre-compiled "machine-code".

Solutions proposed by 1. and 2. above are mainly oriented around extending Web browsers and do not address some of the other limitations discussed in section 1.2 such as synchronization. Among the proposals, scripting and virtual machine seem to be the best approach in providing interactive multimedia on the Web. A major reason is that they provide powerful client-side processing power which the current Web browsers do not have.

This thesis proposes using another platform, ScriptX, which combines the idea of scripting and virtual machine, as a substrate to offer the desired characteristics as discussed in section 1.1. Extensions and enhancements are proposed to make ScriptX satisfy all the characteristics.

1.3 Existing ScriptX Platform

ScriptX (Version 1.0) is a multimedia authoring and distribution product from Kaleida Labs, Inc. ScriptX has been implemented for the Microsoft Windows and Macintosh platforms. It combines scripting with a virtual machine. A virtual machine is a software interpreter of precompiled "machine code". Application developers use the ScriptX development platform to write scripts to generate multimedia presentations. The scripts are object-oriented. The scripts are then compiled into platform independent bytecode that is distributed and executed by the virtual machine Kaleida Media Player (KMP). ScriptX includes a rich library of multimedia tools such as timers, drawing tools and external media importers for bitmaps, audio (AIFF and WAV), and video (Quicktime and VFW). It provides extensive user interactivity mechanisms such as an underlying search engine for its data objects, and user interface objects like push-buttons and scrollbars. It also has rich system supports such as a tasking mechanism and thread scheduler.

Using ScriptX as a client platform for networked interactive multimedia is close to realizing the desired goals discussed in section 1.1. Specifically, ScriptX provides extensive support for Spatial Synchronization. An object can be placed anywhere within a presentation window, and there is built-in support for making objects draggable and scalable. The User Interactivity support goes beyond the requirements in section 1.1 due to the extensive interactivity support and client-side processing ScriptX provides.

However, as far as the other three system components are concerned, extensions or enhancements are needed for ScriptX to completely satisfy the desired capability.

1.3.1 Insufficiencies of Existing ScriptX Platform

1. **Data Flow Component: lack of built-in network support**

   The current version of ScriptX is a stand-alone package. It does not provide any network interface. Network extensions have to be built into ScriptX to support the prefetching model as described in section 1.1.

2. **Temporal Synchronization: lack of structured support for non-linear model**

   ScriptX has extensive facilities for timing which readily and easily support linear synchronization. Through the use of built-in master-slave relationships, linear synchronizations of media playback is easily attained (details discussed in chapter 5). However, currently there is no structure for supporting non-linear synchronization, although the underlying tools are already in place.

3. **Media Component: lack of structured support to prepare individual text-based media**

   ScriptX provides facilities to display, format (color, sizes, fonts, etc.) and edit text in a multimedia presentation. Text for enriching presentations are usually "hard-wired" into the scripts, with rigid content and format set into the scripts. On many occasions, it would be useful to have individual pre-formatted text files, separated from the scripts, that can be readily imported into ScriptX to produce special text-based media. A common example is a script for a multimedia courseware template. Groups of individual pre-formatted text files which represent slides in a lecture can be imported to the same template to produce the different lecture materials. Moreover, specialized text-based media such as a stream of text which can be rendered dynamically are worth creating since they are useful in most applications.

1.4 Proposed Solutions

To make ScriptX a better platform for developing and delivering network interactive multimedia on the Web which fully satisfies the desired capability as discussed in section 1.1, this thesis proposes building abstraction models to extend or enhance the existing ScriptX platform.

1. **Data Flow Abstraction**

   A network API to be used by ScriptX developers will be designed and implemented. The API will be able to perform background fetching operations. The API will be implemented for the Windows platform.

2. **Temporal Synchronization Abstraction**

   To design and implement a ScriptX class that provides the structure to support non-linear synchronization.

3. **Text-based Media Abstraction**

   To design and implement ScriptX classes that will provide the structure to create formatted, static text media from pre-formatted text files and to create dynamic text stream media.

There are other issues concerning scripting languages in general such as performance due to overhead in interpretation and security. However, those issues are beyond the scope of this thesis.

The remainder of this thesis is divided into six chapters. Chapter two and three describe the design and implementation of the Data Flow Abstraction, respectively. Chapter four describes the design and implementation of the Temporal Synchronization Abstraction. Chapter five discusses the design and implementation of the Text-based

Media Abstraction. Chapter six demonstrates the integrated use of the three abstraction models by building a multimedia presentation. Chapter seven concludes and lays the foundation for future work. The remainder of this thesis assumes a general knowledge about object-oriented programming principles.

Chapter 2

Data Flow Abstraction: The Design

To alleviate the limitation of ScriptX's lack of any built-in network support, the main goal of building the Data Flow Abstraction is to facilitate *background fetching* of data over the network. Application developers use the ScriptX class `BackgroundFetchingAgent` as a network API, while details of the design and implementation are "abstracted" away. This chapter discusses the design of the `BackgroundFetchingAgent` API.

The rest of the chapter is organized into seven sections. Section 1 discusses the current external support that ScriptX can use to deliver applications over the network and the new model for network use offered by this work. Section 2 discusses issues driving the design of the `BackgroundFetchingAgent` API. Section 3 is a design overview. Section 4 discusses the reasoning behind choosing HTTP as the Network Transport Protocol. Section 5 discusses details of the `BackgroundFetchingAgent` API. Section 6 gives examples of how to use the API. Finally, section 7 evaluates the API in terms of the original design issues.

2.1 Network Use Models

2.1.1 Current Model without Built-in Network Support

The existing ScriptX platform does not have built-in network support, but it can leverage external facilities such as the Web to provide limited network support. ScriptX titles -- compiled ScriptX scripts in bytecode format that are executed by the Kaleida Media Player (KMP) -- can be delivered as a single data object to be played on client machines running the KMP, by using the Web as the network delivery medium. The following steps describe the processes involved in delivering ScriptX titles over the network using the Web:

1. Set up the appropriate MIME-type extensions on the local Web browser to launch the KMP as a helper application for executing a ScriptX title.
2. Using a Web browser, find a ScriptX title on some Web server.
3. Retrieve the title using the Web protocol. When the entire title has been downloaded, the browser will launch the KMP to execute the script.
4. Add-ons to a ScriptX title, such as another title with media effects, can be retrieved the same way incorporated into a running KMP ScriptX title.

Such a model requires switching between the Web browser and the KMP, whenever a ScriptX title needs to be retrieved over the network. It also requires that all of the media objects needed by a title be pre-packaged with the title. Every ScriptX title delivered and played this way should be self-sufficient; which means that the whole bulk of the title with all the constituent data in it has to be transferred over the network before rendering.

2.1.2 Proposed Model with Built-in Network Support

This thesis proposes extending the current ScriptX platform by building network support into it. A running ScriptX title will be able to fetch data files in the background over the network and incorporate the data into the title when the whole data file is downloaded. The network use model with the proposed built-in network support will be more flexible than the original model. First, once a title with network support has been delivered to a client (possibly using the Web by means of the steps (1)-(3) in section 2.1.1.), the ScriptX program does not need help from external applications (like a Web browser) to get data over the network. It can any data on the network on its own and incorporate it into the title. Background fetching means that initially only a part of all the data in the presentation needs to be available to get it started. The rest of the data can be fetched over the network while the presentation is still running. This has certain advantages. First, the bandwidth and local storage is better utilized, since only a part of the whole presentation needs to be transferred and stored at a time. Second, new media can be incorporated dynamically into a title, unlike the original model where media is prepared and packed into ScriptX titles or libraries.

2.2 Design Issues

There are basically three issues driving the design:

• Capability

The two principle capabilities that this API aims to achieve are described by the two key words: *Fetching* and *Background. Fetching* entails a sequence of client operations: set up a network connection with a remote server, request a data object from the remote server, receive the data and store it locally. After the data file has been fetched in whole, it can be incorporated into the presentation. Fetching should occur in the background. This implies that a fetch should run silently, not disturbing other system threads and, for multimedia presentations in particular, not disturb the media rendering and user control in the foreground. Other desired capabilities include checking the availability of a network connection without actually transferring a file, raising an error on a failed fetch, and stopping a running fetch operation.

• Simplicity

Creating multimedia applications is a complex and time-consuming task. To make life easier for application developers, simplicity is very important to save them time when they use the background fetching facilities in composing an application. In general, application developers should worry about nothing more than the remote host/location from which the data is to be fetched, and the local file to which the data is to be saved. This API abstracts implementation details away from multimedia application developers.

• Open Standards

Employing open protocols or open standards in our design is critical. Using the Web as an example, openness ensures that people can extend and integrate other people's work easily into the Web. The explosive growth of the Web, which is built around open standards (like HTTP), undoubtedly testifies the importance of open standards.

2.3 Design Overview

The design consists of two main elements: the `BackgroundFetchingAgent` ScriptX class, and the network extension loadable module. The `BackgroundFetchingAgent` is the API (application programming interface) used by an application developer to access our network extension facilities built for the ScriptX platform. To fetch data over the network, a script will create a new instance of the `BackgroundFetchingAgent` class in ScriptX, and use its instance methods to do the fetching.

The network extension loadable module provides the underlying mechanisms for network extension facilities to ScriptX. The module identifies a Network Transport Protocol to be employed for the ScriptX clients to communicate with remote servers on the Internet. We have chosen HTTP as our Network Transport Protocol. The rationale for choosing HTTP will be given in a later section. The loadable module is written in C and is dynamically loaded (and linked) into the ScriptX environment when the `Background-FetchingAgent` class is loaded, so that a `BackgroundFetchingAgent` instance can access the network functions. The ScriptX Loader is responsible for loading in external object files or libraries into the ScriptX environment. A handle to an "entry-point" function in the loadable module can be obtained from the Loader. This handle allows the entry-point function to be called within the scripts. The `BackgroundFetchingAgent` class has a class variable `ExtFuncPtr` that holds the handle to a function in the network module. As a result, `ExtFuncPtr` can be used to seamlessly call the functions implemented in the network extension loadable module (see Fig. 2.1).

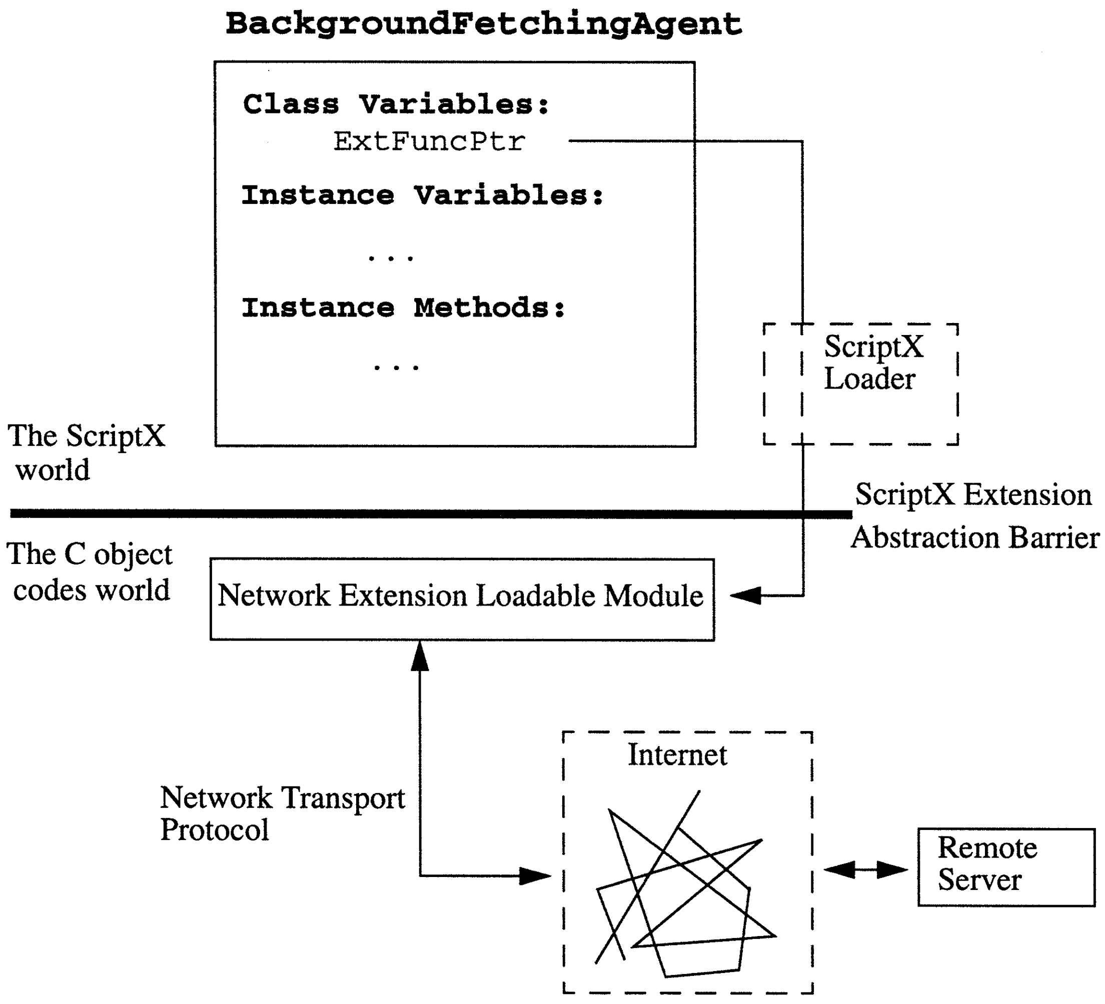


Figure 2.1: Organizational overview of the `BackgroundFetchingAgent` class

2.4 Determining the Network Transport Protocol

HTTP was selected as a natural choice for the network transport protocol because it satisfies the initial design specifications of capability, simplicity and openness. HTTP is a text-based protocol that runs on top of TCP/IP. After opening a TCP connection to a Web server, a Web client requests an object by sending an HTTP request, using the "GET" method including the path to the desired data. Then the Web server sends back an HTTP reply indicating the status of the request. Upon a successful request and reply handshake, the server sends the object over the TCP connection.

HTTP offers simplicity in that no extra work needs to be done on the server side, except setting up the appropriate MIME types in the HTTP server configuration file. Data objects like ScriptX titles can be readily put on a Web server and retrieved by a client from within the ScriptX environment using the network extension (refer to section 2.1.2).

The greatest advantage of using HTTP lies in the fact that it is the open standard atop which the Web is built. The Web is constitutes an enormous repository of multimedia content. To compose a truly interactive multimedia application, users should have tools to incorporate their own media into the application. Given that the ScriptX network extension we built talks HTTP, it can retrieve data from any Web server and the data can be rendered in a ScriptX program.

2.5 Details of the `BackgroundFetchingAgent` Class

This section discusses details about the `BackgroundFetchingAgent` class. The class serves as an API that provides ScriptX developers with network extensions.

Creating and Initializing a New Instance

The following script illustrates how to create a new instance of the `Background-FetchingAgent` class:

```
myAgent := new BackgroundFetchingAgent initMethod: \
  @getSizeNow URL:"http://18.39.0.24/welcome.html" \
  LocalFile:"c/users/jimmy/welcome.htm"
```

`initMethod` takes a keyword argument `@getSizeNow`, which tells the new instance to open a HTTP connection to return the content-length of a specified URL. The new method of `BackgroundFetchingAgent` calls its `init` method and uses the same keyword arguments. The details of calling the `init` method are described below:

init

SYNOPSIS:

`init` *self* `[initMethod:`name`]` `[URL:`string`]` `[LocalFile:` string`]`

Arguments	Values
self	`BackgroundFetchingAgent` object
`initMethod:`	`NameClass`[a] object. Valid values are `@idle`, `@getSizeNow` and `@fetchDataNow` Default value: `@idle`
`URL:`	`String` object. A normal URL pointing to the data needs to be fetched, for instance, `"http://18.39.0.24/welcome.html"` Default value: "" (empty `String`)
`LocalFile:`	`String` object. A path in ScriptX style representing the file to store the fetched data. For instance, `"c/users/jimmy/welcome.htm"` represents `"c:\users\jimmy\welcome.htm"` in DOS. Default value: "" (empty `String`)

Table 2.1: Arguments of `init` method

a. A `NameClass` object in ScriptX is a constant value, usually used as a value for a keyword argument to ScriptX functions.

The `init` method can be applied in one of three ways depending upon the value of the `initMethod` keyword argument:

1. `@idle` indicates that the new BackgroundFetchingAgent instance will remain idle after creation.

2. `@getSizeNow` indicates that the new instance will open an HTTP connection

(sending an HTTP request and getting an HTTP reply) to a remote host to determine the size of the data to be fetched. This option is intended for checking network availability, verifying that the remote host can be reached, and that the specified URL is correct.

3. `@fetchDataNow` indicates that the new instance will open an HTTP connection to the specified remote host and begin retrieving data immediately.

RETURN VALUE: After the creation of a new `BackgroundFetchingAgent` instance, *self*, the instance itself is returned. The *self*.`Status` instance variable should be checked to determine the status of *self*, especially if *self* has been created with the `@getSizeNow` or the `@fetchDataNow` option. *self*.`Status` reflects the status of network operations. The `Status` instance variable is discussed later in this section.

Class Variable

`ExtFuncPtr`

self.`ExtFuncPtr` (read-only) `Primitive` object

`ExtFuncPtr` is a read-only class variable that holds the value of a `Primitive` object. The `Primitive` class is used to define the behavior of executable code objects. A `Primitive` object holds the address of an executable function and the minimum and maximum number of parameters the function requires. `ExtFuncPtr` holds the address of the executable function in the network extension loadable module. Therefore, `ExtFuncPtr` can be used to seamlessly access the network extension facilities. However, under normal circumstances, an application developer will not need to use

ExtFuncPtr. The instance methods of the BackgroundFetchingAgent presents an abstraction that hides the network extensions.

ExtFuncPtr is created as a class variable instead of an instance variable so that the network extension loadable module is loaded the first time the BackgroundFetchingAgent is loaded into the ScriptX environment.

Instance Variables

URL

self.URL (read-write) String object

URL is a read-writable instance variable of type String object. This variable holds the URL that the instance has been initialized with at creation time. For example, "http://18.39.0.24/welcome.html". The value of URL is used to determine the location from which data is retrieved. The value of URL can be reset so that a single BackgroundFetchingAgent instantiation may be used to retrieve multiple data objects. The default value for URL is "" (empty String).

LocalFile

self.LocalFile (read-write) String object

LocalFile is a read-writable instance variable of type String object. This variable holds the complete path to a file on the client machine for storing data objects retrieved from across the network. The path is given in ScriptX syntax: "c/users/jimmy/welcome.htm" represents the DOS path "c:\users\jimmy\welcome.htm". The value of LocalFile can be changed and the next fetch using the BackgroundFetchingAgent instance will use the current value of the LocalFile instance variable. The default value for LocalFile is "" (empty String).

Size

self.Size (read-only) Number object

Size is a read-only instance variable of type Number object. This variable holds the size of the data pointed to by the URL instance variable. Size has the appropriate value if the BackgroundFetchingAgent instance has been created using the @getSizeNow option, or the GetSize instance method has been applied explicitly on the instance. The default value for Size is undefined.

Status

self.Status (read-only) NameClass object

Status is a read-only instance variable of type NameClass object. After a new instance of BackgroundFetchingAgent is created, the status instance variable can be queried to reflect the current status of the instance. The value of the Status instance variable is one of four (NameClass) values:

Value of *self*.status (NameClass object)	Meaning
@ready	The instance has been created with either @idle or @getSizeNow option, and the instance is ready for fetching.
@done	The instance has finished fetching the data.
@fetching	The instance is now fetching the data.
@error	An error has occurred. The logString instance variable can be examined for sources of error.

Table 2.2: The Values of the Status Instance Variable

HTTP_REQUEST

self.HTTP_REQUEST (read-only) String object

HTTP_REQUEST is a read-only instance variable of type String object. It holds the string representing the HTTP request that is sent to the remote Web server specified by the URL instance variable. HTTP_REQUEST has the appropriate value if the Background-FetchingAgent instance has been created using either the @getSizeNow or the @fetchDataNow option, or after either of the GetSize or Fetch instance methods has been applied explicitly on the instance. The default value for HTTP_REQUEST is "" (empty String).

HTTP_REPLY

self.HTTP_REPLY (read-only) String object

HTTP_REPLY is a read-only instance variable of type String object. It holds the string representing the HTTP reply received from the remote Web server in response to an HTTP request. HTTP_REPLY has the appropriate value if the BackgroundFetchingAgent instance has been created using either the @getSizeNow or the @fetchDataNow option, or either of the GetSize or Fetch instance methods has been applied explicitly on the instance. The default value for HTTP_REPLY is "" (empty String).

logString

self.logString (read-only) String object

logString is a read-only instance variable of type String object. It contains a log of the information about the most recent network session attempted by the Back-groundFetchingAgent instance. If an error occurs during the session, as indicated by the `status` instance variable having a `@error` value, the `logString` will contain hints for determining the error sources. For example, common errors such as non-existent URL or network failures will be reported in `logString`.

Instance Methods

`GetSize`

SYNOPSIS:

`GetSize` *self*

Argument	Value
self	`BackgroundFetchingAgent` object

Table 2.3: Argument to `GetSize` Instance Method

`GetSize` opens an HTTP connection with the remote Web server specified by *self*.`URL` and gets the size of the relevant data object. *self*.`Size` will hold the size of data if the operation succeeds.

RETURN VALUE: On a successful return, the size of the data pointed to by *self*.`URL` will be returned as a `Number` object and *self*.`Status` be set to `@ready`. If an error occurs, a negative value will be returned, *self*.`Status` will be set to `@error`, and *self*.`logString` can be examined to locate error sources.

`Fetch`

SYNOPSIS:

`Fetch` *self*

Argument	Value
self	`BackgroundFetchingAgent` object

Table 2.4: Argument to `Fetch` Instance Method

`Fetch` opens an HTTP connection with the remote Web server and transfers the data pointed to by *self*.`URL` into the file indicated by *self*.`LocalFile`. Before fetching, *self*.`status` should be set to `@ready`. During the fetching, *self*.`status` is set to `@fetching`; after completing, *self*.`status` is set to `@done`.

RETURN VALUE: On a successful return, the size of the data pointed to by *self*.`URL` is returned as a `Number` object and the data object itself is stored in the file *self*.`LocalFile`. *self*.`Status` is set to `@done`. If an error occurs, a negative value is returned, *self*.`Status` is set to `@error` and *self*.`logString` can be examined for the source of error.

Summary of `BackgroundFetchingAgent` API:

Class Variable	`ExtFuncPtr`

Instance Variables	`URL`
	`LocalFile`
	`Size`
	`Status`
	`HTTP_REQUEST`
	`HTTP_REPLY`
	`logString`

Instance Methods	`GetSize` `Fetch`

Table 2.5: Summary of the `BackgroundFetchingAgent` API

2.6 Example Usage of the API

This section provides several examples that use the `BackgroundFetchingAgent` API. The examples demonstrate:

1. checking the availability of a network connection without actually transferring a file,

2) two ways to do background fetching,

3) error reporting and error source checking

4) abolishing a running fetch operation.

Example 1

```
myAgent := new BackgroundFetchingAgent initMethod: \
     @getSizeNow URL:"http://18.39.0.23/welcome.html"
```

The above script creates a new BackgroundFetchingAgent object which, after creation, attempts to make an HTTP request to the remote Web server to determine the content-length of the data indicated by the URL keyword argument. If the script returns successfully, it means that the remote host is accessible and that the URL points to a valid location for the desired data object. A successful return also means that myAgent.Status is set to @ready and myAgent.Size will have a non-negative value for the size of the data. If an error occurs, myAgent.Status will be set to @error and myAgent.logString can be used to locate the error source. The same network connection checking can also be done by applying the GetSize instance method on an existing BackgroundFetchingAgent object as shown in the following script:

```
myAgent3 := new BackgroundFetchingAgent initMethod:\
     @idle URL:"http://18.39.0.23/welcome.html" \
     LocalFile: "c/users/jimmy/welcome.htm"
GetSize myAgent3
```

Example 2

In general, there are two ways to perform background fetching using the BackgroundFetchingAgent API. One may either fetch at creation time of a BackgroundFetchingAgent object or one may apply the Fetch instance method on an existing BackgroundFetchingAgent object.

```
(myAgent2 := new BackgroundFetchingAgent initMethod: \
    @fetchDataNow URL:"http://18.39.0.23/welcome.html" \
    LocalFile: "c/users/jimmy/welcome.htm" &)
```

The above script creates a new `BackgroundFetchingAgent` object which, immediately after creation, attempts to fetch the data pointed to by the `URL` keyword argument. The use of '`&`' creates a new thread for the script directing the fetch to run in background mode. During fetching, `myAgent.Status` is set to `@fetching`. If the fetch completes successfully, `myAgent.Status` is set to `@done`.

```
myAgent2.URL := "http://18.39.0.23/picture.gif"
myAgent2.LocalFile := "c/users/jimmy/pict.gif"
global t := (Fetch myAgent2 &)
```

The above script demonstrates the application of the `Fetch` instance method to an existing `BackgroundFetchingAgent` object. `myAgent2` is reused in this example, and its `URL` and `LocalFile` instance variables are set to new values so that a different file is returned. Using '`&`' in this example creates a new thread for fetching and the thread is assigned to the global variable `t` (which is the same `t` in the above script).

```
threadKill t
```

If we want to abolish the running thread `t` any time during the fetching operation, we can use instance method `threadKill` of the `Thread` class in ScriptX to kill the thread.

2.7 Assessing the API in terms of the Design Issues

The `BackgroundFetchingAgent` API in our design appears to satisfy all three design issues -- capability, simplicity and open standards -- discussed in section 3.1.1.

As the examples provided in section 3.7 demonstrate, the API possesses the four capabilities that we initially sought to achieve:

1. background fetching,
2. checking network availability without transferring a file,
3. error reporting and error source checking,
4. ability to abolish a running fetching operation.

As far as simplicity is concerned, in using the API, application developers need only specify the URL to fetch and a local space to store the returned data. The `Background-FetchingAgent` class looks like an ordinary ScriptX core class to the developers. The underlying network extension implementation is transparent to the developers. In addition, choosing HTTP as our network transport protocol leverages off of the enormous power of the Web -- both as a repository of multimedia data and as a tool for discovery of resources on the Web.

Chapter 3

Data Flow Abstraction: The Implementation

The underlying mechanisms supporting the network extension facilities of the `BackgroundFetchingAgent` are implemented as a single, loadable object code module written in the C language. This chapter discusses the implementation issues of that object code module.

This chapter is organized into three sections. Section 1 gives an overview of the platform and tools employed, and then discusses environment specific implementation issues. Section 2 discusses the steps taken to ensure that the network extensions run in background mode. Section 3 describes the current status of the implementation.

3.1 The Implementation Environment

The network extension module is implemented on a Pentium 90 PC running Microsoft Windows 3.1. The WinSock API is used as the network programming interface to access Windows network facilities. The Watcom C/C++^{32} compiler is used to generate 32-bit object modules written in C that are loaded into the ScriptX (Version 1.0) environment by the ScriptX Loader. The Watcom C/C++^{32} compiler is the only compiler currently being supported by the ScriptX extension interface to write external loadable code on the Windows platform. The following subsections offer a brief overview of the WinSock API, the ScriptX extension interface and issues specific to the implementation due to the characteristics of the extension interface.

3.1.1 The WinSock API[2]

The Windows Socket Application Programming Interface (WinSock API) is used to implement HTTP access in our design by providing functions to manage TCP connections and data transfer. The WinSock API is a library of functions that defines a network programming interface for Microsoft Windows based on the "socket" paradigm popularized by the Berkeley Software Distribution (BSD) of Unix. It encompasses both familiar Berkeley socket style routines and a set of Windows-specific extensions designed to allow the programmer to take advantage of the message-driven nature of Windows.

The WinSock API defines the top level of the WinSock Dynamic-Linked Library (DLL). The WinSock DLL is 16-bit. The WinSock DLL can be dynamically loaded by calling functions in a user application either at load time or at run time. The network module in this implementation loads the DLL at run time, and calls functions in the DLL to access Windows network facilities. The following diagram gives an organizational view of our program accessing the network facilities through the WinSock DLL and any standard TCP/IP stack:

2. Most of the materials in this section are derived from the book "Programming WinSock" [11] and the WinSock FAQ [12].

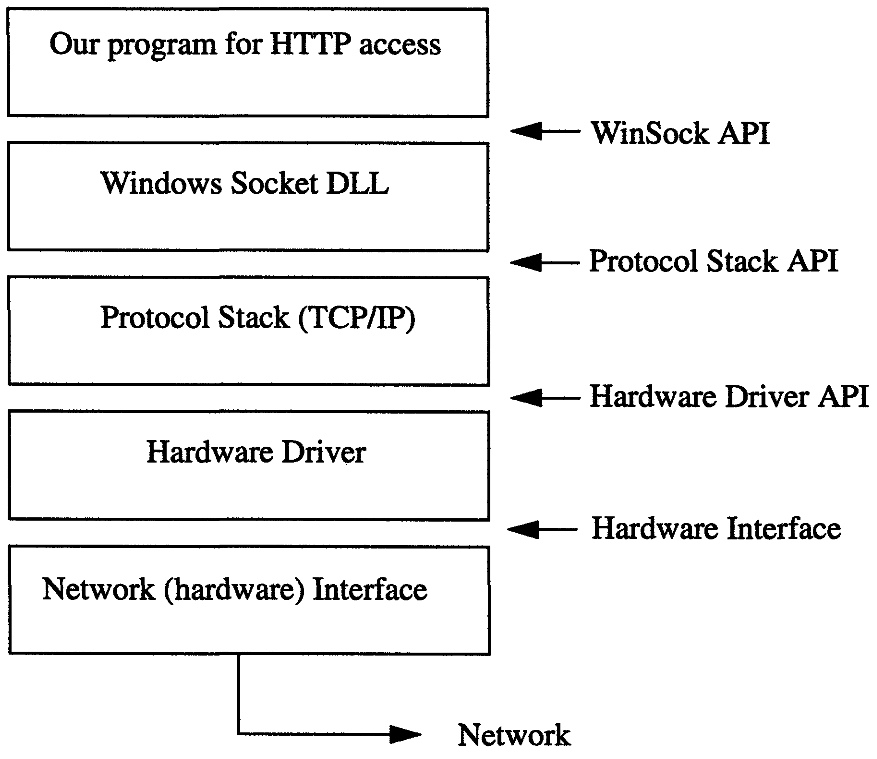


Figure 3.1: Organizational View of using the WinSock DLL

3.1.2 ScriptX Extension Interface and Implementation Issues[3]

The ScriptX extension interface allows platform-specific object codes to be loaded in the ScriptX environment to extend the original ScriptX functionalities. On the Windows platform, the Watcom C/C++^{32} compiler can be used to generate individual 32-bit object files (.obj) or a library file (.lib, a collection of object files) that can be loaded into ScriptX. When the ScriptX Loader loads in a module (single object file or a library file), a single "entry-point" function can be exported to the scripts in ScriptX and called directly by the script passing arguments as ScriptX objects. There are no callbacks from the C world to the ScriptX world except for the `SXwriteString` function which writes a C string to a

3. Most of the background materials in this section are derived from The ScriptX Developer's Guide [10] and The WATCOM C/C++^{32} User's Guide [13].

ScriptX `stream` object. There are only three data types that can be freely passed back and forth across the C / ScriptX abstraction barrier: `int`, `double` and `string`. In this development environment, several issues have received special attention in this implementation:

• Thunking and Indirect Function Calls to 16-bit WinSock DLL Functions

Since the Watcom C/C++[32] compiler produces 32 bit object code while the WinSock DLL is 16-bit, the DLL functions cannot be called directly. Instead, the Watcom compiler provides special functions to handle thunking of arguments and calling the 16-bit functions indirectly. When the address of a 16-bit function in a DLL has been obtained (via `GetProcAddress()` for example), `GetIndirectFunctionHandle()` can be called to obtain a handle to the function, and thunking the arguments will be done automatically when `InvokeIndirectFunction()` is then called to execute the 16-bit function. Thunking also needs to be performed when data such as a pointer from the 16-bit world needs to be used in the 32-bit program.

• Use of `SXwriteString` for function side-effects

Due to the lack of callbacks from loadable code to ScriptX functions (except the `SXwriteString` function), interaction between the script that calls an external function and the function itself is limited to the single value from the external function that is returned to the caller script. For both the `GetSize` and `Fetch` instance methods of the `BackgroundFetchingAgent` API, three instance variables are updated during a call to the network extension function: `HTTP_REQUEST`, `HTTP_REPLY` and `logString`. The updates are performed by calling the `SXwriteString` function on the variables which hold the respective `String` objects (`String` is a subclass of `stream` in ScriptX and therefore works with the `SXwriteString` callback function).

• Single Exported Function for each Loadable Module

The ScriptX Loader only creates a single handle for an exported function in the loadable module. Therefore, if the extension has more than one desired operation, the exported function has to be able to dispatch to execute different subroutines, depending upon a particular argument passed from the scripter level. For the `BackgroundFetch-ingAgent` API, two instance methods represent the two desired operations: `GetSize` and `Fetch`(refer to section 2.5). Therefore, the above dispatching technique is used.

3.2 Fetching in Background Mode

In the implementation, a combination of the following mechanisms was used to ensure that the fetching process would run in a background mode on Windows. Doing so would minimize the effect on system threads such as ScriptX's automatic garbage collection or the multimedia presentation being rendered in the foreground.

• Non-blocking Sockets and Asynchronous Functions

Non-blocking sockets and asynchronous functions are provided by the WinSock API to deal with blocking function calls. Many of the socket functions--such as `connect()`, `send()`, `recv()`--take an indeterminate amount of time to execute. When a function exhibits this behavior, it is said to block; calling the function blocks the further execution of the calling program. Because the Windows platform cannot preempt a task (unlike Windows NT and Unix), all other programs are put on hold until the blocking call returns. This would inhibit the ScriptX system threads as well as many others running in the foreground.

To deal with the blocking calls, non-blocking sockets can be created, and WinSock's asynchronous functions can be used to handle those calls. If a socket is created in blocking mode, the blocking function will not return until the call is completed or a timeout or error occurs. If a socket is created in nonblocking mode, the call to the blocking function returns immediately. A separate function is used to determine the status of the call. The WinSock asynchronous functions were added to WinSock to better fit Berkeley sockets to the message-driven Windows paradigm. Event notification messages are received by an application when a previously called non-blocking function returns. In the meantime, the rest of the program remains fully responsive to the user's actions.

• Voluntary Yield

Voluntary yields are functions provided by the ScriptX extension API. ScriptX has a system tasking mechanism and a thread schedular which, together, allocate time slices to run active threads in a pre-ordered, linear sequence. A thread is said to "yield voluntarily" when it relinquishes control even though it has not returned and its time slice has not been expended. If the network extension program runs for a long time, like when fetching a large data object, the tasking mechanism of ScriptX may be disabled for as long as the program runs. Therefore, in the implementation, when receiving the data, once the HTTP handshake with a Web server is done, the program loops, filling a relatively small buffer in each iteration. The program yields to other ScriptX threads upon completion of each iteration.

• Running as a Separate Thread in a ScriptX Title

In addition to yielding voluntarily, the network extension program should run as a separate thread when called from within a ScriptX title. This is probably the most critical step needs to be taken, because if the fetch is run as the same thread as the title thread, it would freeze some of the title's rendering. ScriptX provides a shorthand for creating a new thread running a script, by adopting the Unix background thread '&' command notation. For instance "(some scripts &)" will create a separate thread. Creating a separate thread to run the network extension program allows the ScriptX thread schedular to allocate time slices for network fetching, decreasing the likelihood that a network request will freeze the system.

3.3 Current Status of Implementation

At the time of writing this thesis, all of the functions provided by the `Background-FetchingAgent` API are supported. Thus the four capabilities noted as design issues in section 2.2 have been satisfied. Currently, the network extension allows fetching a URL in which the remote host is specified by its IP address. A hostname resolution function is not yet implemented. Hostname resolution will be a priority for future work on this project.

Chapter 4

Temporal Synchronization Abstraction

ScriptX's timing component readily supports linear synchronization. However, there is no built-in structure to handle non-linear synchronization, especially in a networked environment. The purpose of the Temporal Synchronization Abstraction is to support non-linear synchronization. For this purpose, the `MasterSlaveContract` ScriptX class has been created. This chapter defines linear and non-linear synchronization and then discusses the design and implementation of the `MasterSlaveContract` class. Examples of how to use the `MasterSlaveContract` class are given.

4.1 Timing and Linear Synchronization in ScriptX

ScriptX provides fine-grain support for timing and synchronizing time-based operations. Clocks, represented by the `Clock` class, are used in ScriptX to time and synchronize animation, sound, video, or any other time-based operations. A clock's time is measured in "ticks", which is the product of the clock's *rate* and its *scale*. A clock's rate is measured in sweeps per second; scale indicates the number of tick marks on the face of the clock. To simplify this discussion, subsequent references to clocks and timing in ScriptX assume a scale of 1 unless otherwise specified. Rate (now measured in ticks per second) and local clock time (measued in ticks) are the two quantities of interest.

A clock's time is controlled by its rate. If the rate is zero, the clock stops. Otherwise, the clock runs at a pace specified by its rate. To synchronize multiple events, clocks can be arranged hierarchically. A *Master-and-Slave* relationship can be set up between two clocks. Through the master-and-slave relationship, the slave is controlled by manipulating two parameters: the *effective rate* (real rate at which the slave runs), and the *offset* (initial difference in time relative to the master). Both quantities are influenced by the master.

In general, the slave clock's *effective rate* is a function of its master's *effective rate*:

$$Slave'sEffectiveRate = Master'sEffectiveRate \times Slave'sRate \qquad [4.1]$$

If the master clock is a top clock, i.e. not itself a slave of another clock, then it has an effective rate equal to its rate. Thanks to the above equation, the slave can be controlled by solely changing the rate of its master. For instance, consider a slave clock, *S*, with its rate initially set to 1 (1 tick per second). *S* won't run unless its master, *M*, has a non-zero effective rate. If *M* is the top clock, its effective rate equals its rate. If *M*'s rate is set at 0, both *M* and *S* pause. If *M*'s rate is set to 1, *S* runs at its normal rate of 1 tick per second. If *M*'s rate is set to a value larger than 1, *S* runs faster than its normal rate. If *M*'s rate is set to a negative value, *S* runs backwards.

The slave's *offset* determines the difference in time between the slave clock and the master clock. Since master and slave clocks can run at different rates, the offset specifies this difference at a specific time: when the slave's time is 0. The difference is expressed in ticks of the master. When the slave's offset is any value other than 0, the slave's clock will initially have a negative value when its master clock is at 0. Then, as the master clock runs, and if the slave has non-zero effective rate, the slave's time will reach 0 when the master clock reaches the value of the slave's offset.

In ScriptX, time-based media rendering can be done through the `Player` class, which is a subclass of `Clock`. Media players such as the audio player, the video player or animation are all subclasses of `Player`. Each `Player` instance or instance of its subclass thus has all the time-based functionalities of `Clock`. In general, to "play," a `Player` object sets its rate to 1; to "stop," its rate is set to 0. Media players such as the audio player and video player start rendering their media when their time reaches 0. To synchronize media rendering in a presentation, a top level master player can be created which has all the media players as its slaves. By specifying the corresponding offsets for each of the slave players and setting each slave's rate to the normal rate at which it should run, each piece of media is rendered at its respectively scheduled time as the master player runs. The master player acts as the top level control for all media. By changing the rate of the master player, the whole presentation can be paused (master's rate = 0), fastforwarded (master's rate > 1), and rewound (master's rate < 0). Each of the constituent media pieces remain synchronized because each slave's effective rate is controlled by the master player.

We define the above model of synchronization as *linear synchronization*. In linear synchronization, the rate of the slave remains unchanged throughout the presentation. By changing the rate of the master alone, the effective rate of each slave changes accordingly. The linear synchronization model is useful for media which is rendered continuously from its start to its end. If we plot out the slave's time versus the master's time (see Figure 4.1), it is a straight line with a constant slope equal to the slave's rate (slave's effective rate divided by master's effective rate, according to equation [4.1]). A slave player renders its media when the slave's time relative to its master reaches 0, when the master player reaches a time equal to the slave's offset.

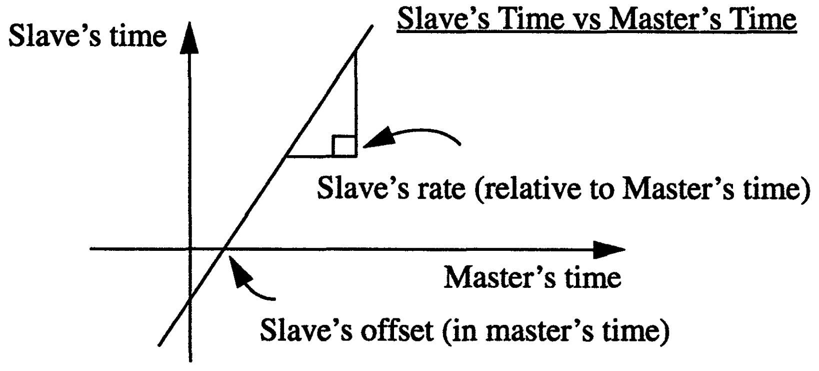

Slave's Time vs Master's Time
Slave's time
Slave's rate (relative to Master's time)
Master's time
Slave's offset (in master's time)

Figure 4.1: Linear Synchronization

4.2 Non-linear Synchronization

The linear synchronization model works well if it is not necessary to change the rate at which the constituent media is being rendered. For instance, a video clip being played continuously starting 5 seconds after the presentation begins. However, very often in a multimedia presentation, the constituent media has to be rendered at different rates over time, independent of the other players: pauses, jumps, plays at different rates. A typical plot of the slave's time versus the master's time looks like Figure 4.2. The plot is non-linear, not like a straight line as in Figure 4.1.

For a simple and common example: pausing a slave video player (while the master player, and thus other slave media playback, continue) for 3 seconds and then resuming afterwards changes the slave video player's rate from 1 to 0 (for 3 seconds), then back to 1. In general, changing a slave's rate explicitly changes the initial linear relationship between the master and the slave. Once this linear relationship is changed, rewinding the master player will not bring the slave back to where it was before the change was made.

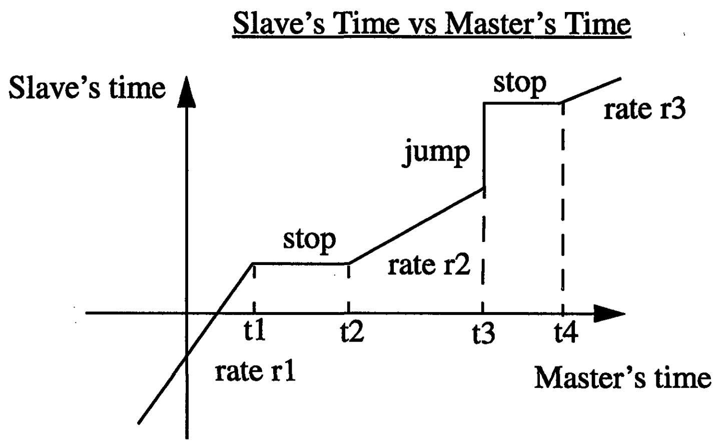


Figure 4.2: Non-linear Synchronization

A new model has to be defined to support non-linear synchronization as in Figure 4.2. For this purpose, the `MasterSlaveContract` class has been designed and implemented.

4.3 Design and Implementation

4.3.1 Design Overview

With a goal of supporting non-linear synchronization, the `MasterSlaveContract` class is created to provide data structures for representing non-linear timing relations (as in Figure 4.2) between a master and its slaves. At each point in time along the master's time line, each slave's time and rate is set accordingly. In general, the slave's time and rate need to be set explicitly when the master time line reaches a point where the slave's rate changes (passing time t2, in Figure 4.2) or when the master's time jumps (at time t3, in Figure 4.2). The ScriptX `Callback` class can be used to call a specified function at certain times or events in a clock's life cycle. `TimeJumpCallback` is a subclass of `Callback` which calls a function whenever a clock's time jumps. Therefore, `TimeJumpCallback` objects can be used by the `MasterSlaveContract` class to update a slave's time and rate (getting the quantities by querying its timing relation data structure) whenever the master jumps to a new time. `TimeCallback` is a subclass of `Callback`. `TimeCallback` calls a function whenever the specified clock reaches a certain time. `TimeCallback` objects can be used to update a slave's time and rate whenever the master reaches a point in the script where a new segment when either one of the two quantities changes.

The `MasterMarker` class was defined to hold the timing relation between a master and a slave. A `MasterMarker` object is a "marker" which marks a segment of the master's time line when the slave has a different rate or when its time jumps.

The `MasterSlaveContract` class manages a list of `MasterMarker` objects for each slave. The `MasterSlaveContract` class creates the corresponding timed callbacks to be called when the master jumps to a new time, or when the master runs into a period marked by another `MasterMarker` object.

How are the timing relations between a master-slave pair being set up using the `MasterSlaveContract` class? There are three ways of doing so:

1. **By hardcoding in the script**: In the script, a list of `MasterMarker` objects can be created explicitly by hardcoding the timing relations between a master and a slave. The `MasterMarker` objects are added to a `MasterSlaveContract` object and then the timed-callbacks are initialized.
2. **By incorporating from a timing relation specification string**: An alternative way to establish a master-slave timing relation is by providing a specially formatted string in the script and incorporating that string into the `MasterSlaveContract` object.
3. **By incorporating from a timing relation specification file**: The third way is to provide a specially formatted file in which the timing specification is given.

The three ways listed above represent a continuum in increasing flexibility for setting up the timing relations of a master-slave pair using the `MasterSlaveContract` class. The increase in flexibility implies a more suitable means for specifying timing relations in a networked environment for multimedia applications.

The following sections discuss details of the `MasterMarker` class (examples of each of these three methods are provided in section 4.4) and the `MasterSlaveContract` class. Examples on how to use the `MasterSlaveContract` class are also given.

4.3.2 Details of the **`MasterMarker`** class

The `MasterMarker` class helps to partition a master player's time line into separate segments. Each segment specifies a different value of the slave's rate. A `MasterMarker` object holds four quantities: master's segment start time, master's segment end time, the initial slave's time, and the slave's rate during the segment.

Creating and Initializing a New Instance

The following script is an example of how to create a new `MasterMarker` object:

```
mark := new MasterMarker start:5 finish:10 \
        SlaveStartTime: 5 SlaveRate: 0
```

The new method of the MasterMarker class calls its init method and uses the same keyword arguments. The details of calling the init method are described below:

`init`

SYNOPSIS:

`init` *self* `[start:num] [finish:num] [SlaveStartTime:num] \`
`[SlaveRate:num] [string:str]`

Argument	Value
self	`MasterMarker` object
`start:`	`Number` object. Default value: 0
`finish:`	`Number` object. Default value: 0
`SlaveStartTime:`	`Number` object. Default value: 0
`SlaveRate:`	`Number` object. Default value: 0

Table 4.1: Arguments of `init` Method

Argument	Value
`string:`	`String` object. Format: `"start,finish,SlaveTime,SlaveRate"` Default value: `""` (empty `String`)

Table 4.1: Arguments of `init` Method

The `string:` keyword argument is used as an alternative way for creating a new `MasterMarker` object. Pass as an argument, a string formatted as specified in Table 4.1. This alternative way simplifies the incorporation of timing specifications from a file.

RETURN VALUE: A new `MasterMarker` object is returned.

Instance Variables

Instance Variable	Value
`Start`	Master's segment start time.
`Finish`	Master's segment end time.
`SlaveStartTime`	Slave's initial time at the segment start time.
`SlaveRate`	Slave's rate during the segment.

Table 4.2: Instance Variables for the `MasterMarker` Class

4.4 Details of the `MasterSlaveContract` class

The `MasterSlaveContract` class serves as a contract between the master and the slave. The contract enforces the non-linear synchronization between master and slave. For each slave, a `MasterSlaveContract` object essentially maintains a list of `Master-Marker` objects that partition the master's time line into segments. Each segment reflects a change in the slave's rate. The `MasterSlaveContract` object creates timed call-backs along the master's time line so that when the master jumps to a new time or when the master runs into a new segment, the slave's state (current time and rate) can be updated correctly by looking up the `MasterMarker` object corresponding to that segment. Default behavior for slaves without any `MasterMarker` object is linear synchronization with the master. Therefore, linear synchronization is a special case of non-linear synchronization.

Creating and Initializing a New Instance

The following script is an example of how to create a new `MasterSlaveContract` object:

```
contract := new MasterSlaveContract master: m slave: s
```

In the above example, both `m` and `s` are instances of the `Clock` subclass. `m` is set up as the master; `s` is the slave. The `MasterSlaveContract` `new` method calls its `init` method, and uses the same keyword arguments. The details of calling the init method are described below:

`init`

SYNOPSIS:

`init` *self* `[master:clock] [slave:clock] [initOffset:num]\`
    `[initRate:num]`

Argument	Value
self	`MasterSlaveContract` object
`master:`	`Clock` (or its subclass) object Default value: `undefined`
`slave:`	`Clock` (or its subclass) object Default value: `undefined`
`initOffset:`	`Number` object. Default value: `0`
`initRate:`	`Number` object. Default value: `1`

Table 4.3: Arguments to the `init` Method

RETURN VALUE: A new `MasterSlaveContract` object is returned. The values of `initOffset` and `initRate` specify the initial offset and initial rate for the slave. If a slave does not have any `MasterMarker` object attached to it, default behavior is linear synchronization based upon the initail offset (refer to section 4.1). Therefore, linear synchronization is a special case of non-linear synchronization.

Instance Variables

Instance Variable	Value
`master`	`Clock` (or its subclass) object, being the master.
`slaves`	`Array` object, holding all the slaves (`Clock` object, or its subclass)
`SlaveMarkersTable`	`HashTable` object. Each entry has a slave as key and an `Array` of `MasterMarker` objects as value.

Table 4.4: Instance Variables of the `MasterSlaveContract` Class

Instance Methods

`AddSlave`

SYNOPSIS:

`AddSlave` *self slave*

Argument	Value
self	`MasterSlaveContract` object
slave	`Clock` (or its subclass) object.

Table 4.5: Arguments to `AddSlave` Instance Method

`AddSlave` adds a new slave, *slave*, to the `MasterSlaveContract` object, *self*. *self*.`master` is set up as *slave*'s master clock. *self*.`slaves` is updated.

RETURN VALUE: The updated *self* is returned.

`DropSlave`

SYNOPSIS:

`DropSlave` *self slave*

Argument	Value
self	`MasterSlaveContract` object
slave	`Clock` (or its subclass) object.

Table 4.6: Arguments to `DropSlave` Instance Method

`DropSlave` removes *slave* from *self*. *self*.`slaves` is updated. *slave*'s master clock is set to `undefined`, and *slave*'s entry in *self*.`SlaveMarkersTable` is deleted. Timed callbacks on *self*.`master` that correspond to *slave* are removed.

RETURN VALUE: The updated *self* is returned.

`AddMMarker`

SYNOPSIS:

`AddMMarker` *self slave mmarker*

Argument	Value
self	`MasterSlaveContract` object
slave	`Clock` (or its subclass) object.
mmarker	`MasterMarker` object

Table 4.7: Arguments to `AddMMarker` Instance Method

`AddMMarker` adds *mmarker*, that belongs to *slave*, to *self*. *self*.`SlaveMarker-sTable` will be updated.

RETURN VALUE: The updated *self* is returned.

`ResetCallbacksForSlave`

SYNOPSIS:

`ResetCallbacksForSlave` *self slave*

Argument	Value
self	`MasterSlaveContract` object
slave	`Clock` (or its subclass) object.

Table 4.8: Arguments to `ResetCallbacksForSlave` Instance Method

`ResetCallbacksForSlave` resets all the timed callbacks that correspond to *slave* on *self*.`master`. The method is called either when all of the `MasterMarker` objects for slave have been added to *self* using `AddMMarker`, or when the list of `MasterMarker` objects for *slave* is modified.

RETURN VALUE: *self* is returned.

`ResetCallbacksForAll`

SYNOPSIS:

`ResetCallbacksForAll` *self*

Argument	Value
self	`MasterSlaveContract` object

Table 4.9: Argument for `ResetCallbacksForAll` Instance Method

`ResetCallbacksForAll` resets all the timed callbacks for all of the slaves in *self*.

RETURN VALUE: *self* is returned.

`IncSpecForSlave`

SYNOPSIS:

`IncSpecForSlave` *self slave* `[dir:d] [path:p] [string:s] \`

[SectionID:id]

Argument	Value
self	MasterSlaveContract object
slave	Clock (or its subclass) object
dir:	DirRep[a] object. Specify the directory to find the specification file. Default value: undefined
path:	String object. Specify the file name for the specification file. Default value: undefined
string:	String object. Format: "initoffset,initRate; s1,f1,sst1,sr1;s2,f2,sst2,sr2;..;.." Default value: "" (empty String)
SectionID:	String object. Default value: "[MSCSPEC]"

Table 4.10: Arguments for IncSpecForSlave Instance Method

a. A DirRep object represents a directory structure in the corresponding platform in ScriptX

IncSpecForSlave provides two ways for incorporating timing relations for a slave and setting up the corresponding timed callbacks with the master: by providing either a specification string or a specification file.

1. **Incorporating from a specification string**:

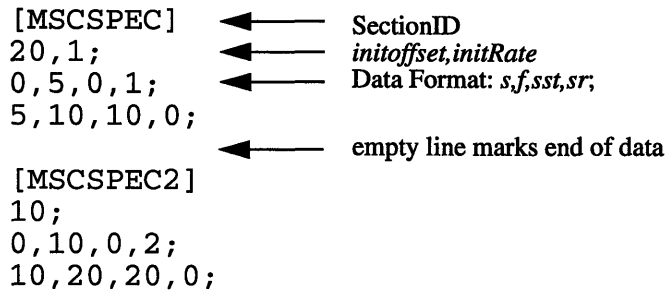
The specification string is of the format:

"initOffset,initRate;s1,f1,sst1,sr1;s2,f2,sst2,sr2;..;.."

where initOffset and initRate are the respective values of the slave's initial offset and initial rate. If only linear synchronization is required, the string need only specify the initial offset and initial rate values. After the ';' delimiter, the rest of the substring is a series of four-number units, delimited by ';'. Each four number unit represents the four values, respectively of: segment start time, segment end time, slave's time at segment start, and slave's rate during segment.

2. Incorporating from a specification file:

`IncSpecForSlave` looks for the specification file specified by `dir` and `path` keyword arguments. In the file, the value of `SectionID` (defaults to `"[MSCSPEC]"`) marks the beginning of the timing relation data in the file to be incorporated. The first empty line or EOF marks the end of the data. :



```
[MSCSPEC]
20,1;
0,5,0,1;
5,10,10,0;

[MSCSPEC2]
10;
0,10,0,2;
10,20,20,0;
```

Figure 4.3: Sample Timing Specification File

Different values of `SectionID` correspond to different sections of the file, where the data may be found. Therefore, multiple sets of data can be included in a single file. Figure 4.3 shows a sample timing specification file. *initoffset* and *initRate* are the initial offset and initial rate of the slave. Each data line is of the format: *"s,f,sst,sr;"*, where *s* is the segment start time, *f* is the segment end time, *sst* is the slave's start time at segment start, *sr* is the slave's rate during the segment.

RETURN VALUE: The updated *self* is returned.

Summary of the `MasterSlaveContract` class

Instance Variables	`master`
	`slaves`
	`SlaveMarkersTable`

Instance Methods	`AddSlave`
	`DropSlave`
	`AddMMarker`
	`ResetCallbacks-`
	`ForSlave`
	`ResetCallbacks-`
	`ForAll`
	`IncSpecForSlave`

Table 4.11: Summary of the `MasterSlaveContract` Class

4.5 Example usage of the `MasterSlaveContract` class

This section provides several examples on how to use the `MasterSlaveContract` class. The examples demonstrate the three different ways of setting up timing specifications for a slave:

1. By hardcoding in the script
2. By incorporating from a string
3. By incorporating from a fileEach of the examples attempt to set up timing specifications for a slave shown in Figure 4.4

•

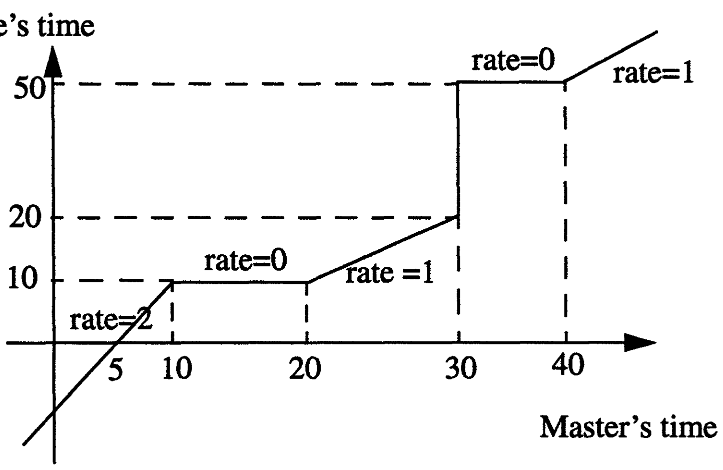


Figure 4.4: Timing Spec. of Slave for all three examples

Example 1

The following script demonstrates setting up the timing relations in Figure 5.4 by hardcoding in the script.

```
global m1 := new player
global s1 := new player
global contract1 := new MasterSlaveContract \
      master:m1 slave:s1 initOffset:5 initRate:2
global mark1 := new MasterMarker start:10 finish:20 \
     SlaveStartTime:10 SlaveRate:0
global mark2 := new MasterMarker start:20 finish:30 \
     SlaveStartTime:10 SlaveRate:1
global mark3 := new MasterMarker start:30 finish:40 \
     SlaveStartTime:50 SlaveRate:0
global mark4 := new MasterMarker start:50 finish:1000 \
     SlaveStartTime:50 SlaveRate:1
AddMMarker contract1 s1 mark1
AddMMarker contract1 s1 mark2
AddMMarker contract1 s1 mark3
AddMMarker contract1 s1 mark4
ResetCallbacksForSlave contract1 s1
```

Example 2

The following script demonstrates setting up the timing relations in Figure 5.4 by incorporating a string:

```
global specStr := "5,2;10,20,10,0;20,30,10,1;30,40,50,0;\
      50,1000,50,1"
global m2 := new Player
global s2 := new Player
global contract2 := new MasterSlaveContract master: m2\
     slave: s2
incSpecForSlave contract2 s2 string:specStr
```

Example 3

The following script demonstrates setting up the timing relations in Figure 5.4 by incorporating a file "ex3.ini", located at theScriptDir (ScriptX global variable for directory launching the running script), which have the content:

```
[example3]
5,2;
10,20,10,0;
20,30,10,1;
30,40,50,0;
50,1000,50,1;

[...]
```

Figure 4.5: Specification File for Example 3

```
global m3 := new Player
global s3 := new Player
global contract3 := new MasterSlaveContract master:m3\
       slave:s3
incSpecForSlave contract3 s3 dir:theScriptDir \
       path:"ex3.ini" SectionID:"[example3]"
```

Chapter 5

Text-based Media Abstraction

Text is an important medium in most multimedia applications. Richly formatted text moving actively on the screen is commonplace in lively multimedia presentations. For multimedia applications in a networked environment where bandwidth and local storage are concerns, text provides an efficient means of content delivery (because of its small size), and nonetheless an attractive medium if client-side processing of the text is possible. In ScriptX, text has associated attributes that specify its color, font, size, etc. Text formatting is done within a script, and the process requires a great deal of detailed attention that could have been avoided.

The `SAText` (*Static, Annotated Text*) class was developed to provide an alternative way to prepare and present formatted text in ScriptX. The `SAText` class creates formatted `Text` objects from an annotated source: a string or an ASCII text file. The contents are annotated in the source with formatting command strings The `SAText` class is intended to provide a simpler and more flexible way to prepare text-based media in ScriptX. It also allows formatted text to be distributed in a networked environment just like other media such as graphics, audio or video clips, which are separated from the scripts or applications.

The `SAText` class has been further extended to include the `DAText` (*Dynamic, Annotated Text*) class. A `DAText` object, like a video player, plays a stream of formatted text. Variable rates for playing `DAText` objects allows fastforward, rewind, and pause, just as in a video player. This chapter discusses the design and implementation of the `SAText` class and the `DAText` class. Examples in using the `SAText` and `DAText` classes will be given at the end of the chapter.

5.1 Existing Text Support in ScriptX[4]

The Text component in ScriptX provides facilities for the display, editing and formatting of text, including paragraph formatting. The Text component consists of the `Text` class, `TextPresenter` class and the `TextEdit` class. A `Text` object is a subclass of `String`. The `TextPresenter` and `TextEdit` classes hold `Text` objects as their "target", and are responsible for displaying (and editing, for `TextEdit`) the text. A `Text` object is more than a string in that the plain text string that it represents, can be formatted by setting its attributes such as size, font, color, to different values. The whole target string is indexed into cursor positions:

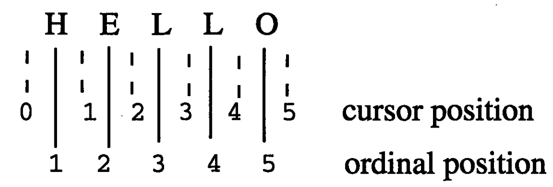


Figure 5.1: Cursor Position and Ordinal Position

To format a part of the string represented by a `Text` object, application developers specify the range of the cursor positions to format and then apply the instance methods `setAttr` or `setAttrFromTo` on the `Text` object. Formatting text this way with ScriptX is time-consuming, considering the trouble counting the cursor positions to change the attributes.

The current ScriptX features also allow the importing of external text files to create `Text` objects. Two text file formats are supported: *plain* ASCII and RTF (rich text format). Imported ascii text files are not translated into plain, ScriptX `Text` objects with default attributes. Formatting plain text in ScriptX involves applying `SetAttr` or `SetAttrFromTo`. An RTF file is a fully formatted document which is used in docu-

4. Most of the background materials in this section are derived from "The ScriptX Architecture and Components Guide".[14]

mentation like this thesis. RTF needs to be generated from an RTF editor, and is not generally used to prepare lively text-based media for a multimedia presentation.

5.2 The **SAText** class

5.2.1 Overview

SAText stands for *Static, Annotated Text.* The SAText class provides a simpler alternative for preparing static, formatted text-based media in ScriptX. The source to an SAText object is either a text string, or an ASCII text file that is annotated with special command strings. Without any annotation in the source, SAText objects are created using the default attributes of Text objects. The SAText class is a subclass of Text. Contents are formatted according to the command string annotations. The command strings are of the form:

@command.argument{*foobar*}

As the content being annotated, *foobar*'s attributes are set by the command string where the new SAText object is created. A typical source file or source string for creating an SAText object is as follows:

```
@leading.15{@size.12{Hello,
@brush.255:255:0{Colorful @size.24{World!}}}}
```

Figure 5.2: Sample Source File for Creating SAText Object

Creating the sample source file into a formatted text (SAText object) needs only on line of script:

```
mytext := new SAText dir:theScriptDir path:"sample.sat"
```

While using the existing ScriptX support will need the following lines of script:

```
mytext2 := "Hello, Colorful World!" as Text
setAttr mytext2 @leading 0 15
setAttr mytext2 @size 0 12
setAttr mytext2 @brush 6 (new Brush color: \
        new RGBcolor red:255 green:255 blue:0)
setAttr mytext2 @size 15 24
```

As shown in the sample file in Figure 5.2, command strings can be nested within one another. If one content string is annotated by two command strings of the same type, the innermost one takes effect. For instance, in the sample source file, `World!` is being annotated by both `@size.24{}` and `@size.12{}`. `@size.24{}` dominates because it is the innermost command string to `World!`. The set of command strings and the possible arguments recognized by the `SAText` class cover all of the attributes and attribute values supported by ScriptX for formatting a `Text` object. The set of recognized command strings are listed in the following table[5]:

Annotation Keyword	Arguments
`@brush`	*r:g:b* (the numbers for red,green,and blue component respectively) e.g. `@brush.255:255:255{}`
`@font`	*font* (name of a system font) e.g. `@font.Times Roman{}`
`@size`	*number* (a value in points) e.g. `@size.12{}`
`@weight`	*name* Possible values: `extralight,light,regular,medium,demi-Bold,bold,extraBold,heavy` e.g. `@weight.bold{}`

Table 5.1: Arguments to the Annotation Keywords in SAText Source

5. For more information about what each attribute does to the Text object in ScriptX, refer to p.770 of "The ScriptX Core Classes Reference"[15]

Annotation Keyword	Arguments
`@width`	*name* Possible values: `condensed`,`normal`,`expanded` e.g. `@width.condensed{}`
`@style`	*name* Possible values: “`roman`”,”`italic`”,”`oblique`” e.g. `@style.italic{}`
`@underline`	*number* Possible values: 0 or 1 e.g. `@underline.1{}`
`@leading`	*number* e.g. `@leading.5{}`
`@paraLeading`	*number* e.g. `@paraLeading.5{}`
`@firstLineLeading`	*number* e.g. `@firstLineLeading.5{}`
`@alignment`	*name* Possible values: `flush`,`flushleft`,`flush-ToEnd`,`flushRight`,`fill`,`center`,`tty` e.g. `@alignment.flush{}`
`@paraIndent`	*number* e.g. `@paraIndent.5{}`
`@indent`, `@indentLeft`	*number* e.g. `@indent.5{}`
`@indentFromEnd`, `@indentRight`	*number* e.g. `@indentFromEnd.5{}`

Table 5.1: Arguments to the Annotation Keywords in `SAText` Source

5.2.2 Details of the SAText Class

The `SAText` class is a subclass of `Text` class.[6] It is basically a `Text` object with its target string's attributes set according to the formatting annotations of the source, which can be a string or a file.

6. For more information about the instance variables and instance methods of SAText class inherited from the Text class, refer to “The ScriptX Core Classes Reference” p.772 [15]

Creating and Initializing a New Instance

The following script illustrates two ways to create a new instance of the `SAText` class:

1. Creating from a source file:

```
satext1 := new SAText dir:theScriptDir path:"example.sat"
```

2. Creating from a source string:

```
satext2 := new SAText string:"@size.12{
     @brush.0:255:0{Hello!}}"
```

The `new` method of `SAText` calls its `init` method and uses the same keyword arguments. The details of calling the `init` method are described below:

init

SYNOPSIS:

`init` *self* `[dir:d] [path:p] [string: str]`

Argument	Value
self	`SAText` object
`dir:`	`DirRep` object. Specifies the directory of source file. Default value: `undefined`
`path:`	`String` object. Specifies the file name of source file. Default value: `undefined`
`string:`	`String` object. Specifies the source string. Default value: `""` (empty `String`).

Table 5.2: Arguments to `init` Method

The init method can be applied in one of two ways:

1. importing from a source file by specifying the location using the `dir:` and `path:` arguments
2. importing from a source string using the `string:` argument.

5.3 The DAText class

5.3.1 Overview

DAText stands for *Dynamic, Annotated Text.* It is a dynamic version of both the SAText class, and the original Text class. DAText objects are like video players, but they render streams of formatted text strings instead of video frames. In the current implementation, the DAText class is a subclass of both the Player class and the TextPresenter class[7]. Subjugation to the Player class permits the DAText objects to exercise play, fastforward, and rewind functionality. DAText objects can also access other time-based facilities through the Clock class (which is the superclass of the Player class). The rate instance variable of a DAText object can be set to render the text at variable rates (positive value for forward rendering, negative value for reverse). Dominance by the TextPresenter class enables the DAText class to render text on the screen. DAText objects can be created the same way SAText objects are created: by using an annotated (or a plain) source string or by importing data from a source file. DAText objects can also be created using existing Text objects to produce moving text.

5.3.2 Details of the DAText class

Creating and initializing a New Instance

The following script illustrates how to create a new instance of the DAText class:

```
datext1 := new DAText dir:theScriptDir path:"example.dat"
```

The new DAText object can then be added to a visible space such as a window and rendered using the play instance method inherited from the Player class. The new method of DAText calls its init method and uses the same keyword arguments.

7. For more information about the instance variables and instance methods of the Player class and the TextPresenter class, refer to "The ScriptX Core Classes Reference" p.779 and p.575 [15]

init

SYNOPSIS:

init *self* [dir:d] [path:p] [string: str] [Text:T]

Argument	Value
self	DAText object
dir:	DirRep object. Specifies the location for the source file. Default vale: undefined
path:	String object. Specifies the file name of source file. Default value: undefined
string:	String object. Specifies the source string. Default value: "" (empty String)
Text:	Text object. Specifies source text. Default value: undefined

Table 5.3: Arguments for init Method

A DAText object can be created in one of three ways:

1. Importing from a source file:

The Dir: and path: arguments specify the location of the source file to open. The source file contains the annotation command strings defined in the SAText class. If the source file does not have any annotations, then the Text class default attributes are used to create the DAText object.

2. Importing from a source string:

The string: argument specifies an annotated (or plain) string as the source for creating a DAText object. If the string is not annotated, the Text class default attributes will be used in creating the DAText object.

3. Using an existing Text object:

The Text: argument specifies an existing Text object to be used for creating the DAText object. The attributes of the original Text object are preserved.

Instance Variables

The instance variables that the `DAText` class inherits from the `Player` class and the `TextPresenter` class are discussed here to illustrate the usage of the DAText class.

Instance Variables Inherited from the `Clock` (through the `Player`) Class:

`rate`

self.`rate` (read-write) `Number` object

`rate` is a read-writable instance variable of type `Number` object. It is used to control the rate at which the `DAText` object renders its moving text. It measures the number of characters per second that are rendered. A positive `rate` value indicates forward playback. A negative value indicates reverse playback. Applying the `play` method on `DAText` sets its rate to 1. If the `DAText` object needs to play at some rates other than 1, the `rate` instance variable has to be set explicitly.

`ticks`

self.`ticks` (read-write) `Number` object

`ticks` is a read-writable instance variable of type `Number` object. Since `rate` measures the number of characters that are rendered per second, the `ticks` instance variable indicates the number of characters that are already rendered. If `ticks` is less than 0, then no characters are being rendered. `ticks` can be set so that for the `DAText` object will jump to any position in the string.

Instance Variables Inherited from the `TextPresenter` Class:

`target`

self.`target` (read-only) `Text` object target is a read-only instance variable of type Text object. target specifies the Text object created by the DAText object. The attributes of *self.*target are either determined by the annotations in the source (file or string) or by the existing Text object used to create the DAText object.

Instance Methods

Some of the instance methods that the DAText class inherits from the Player class, for normal usage are listed in the following table:

Instance methods inherited from Player	Comments
play	sets rate to 1
fastforward	sets rate to 5
rewind	sets rate to -5
stop	sets rate to 0

Table 5.4: Instance Methods from the Player class

5.4 Example usage of the SAText and DAText classes

This section provides several examples that illustrate the usage of the SAText and DAText classes. The examples demonstrates:

1. Creating a SAText object from a source string and rendering it in a window.
2. Creating a DAText object from a source file and rendering it in a window, with varied rates.

Example 1

```
global win := new window
show win
global sat := new SAText string: \
      "@size.24{@brush.255:0:0{Hi!}}"
global textpresent := new TextPresenter boundary: \
        (new Rect x2:100 y2:50) target:sat
    append win textpresent
```

The above scripts creates and shows a new window, `win`, and a new `SAText` object, `sat`, representing the string "`Hi!`" in 24 point font in the color red. `sat` is then put into a `TextPresenter` object which is rendered in `win`.

Example 2

```
    global win := new window
    show win
    global dat := new DAText dir:theScriptDir \
         path: "example2.dat"
    append win dat
    play dat
    stop dat
    dat.ticks := 20
    dat.rate := 5
```

The above scripts create and display a new window, `win`. A new `DAText` object, `dat`, is created and is rendered in `win`. The scripts show how to play `dat` at the default rate (using `play`), 1, and how to stop (using `stop`), jump (setting `dat.ticks`), and play `dat` at a different rate (setting `dat.rate`).

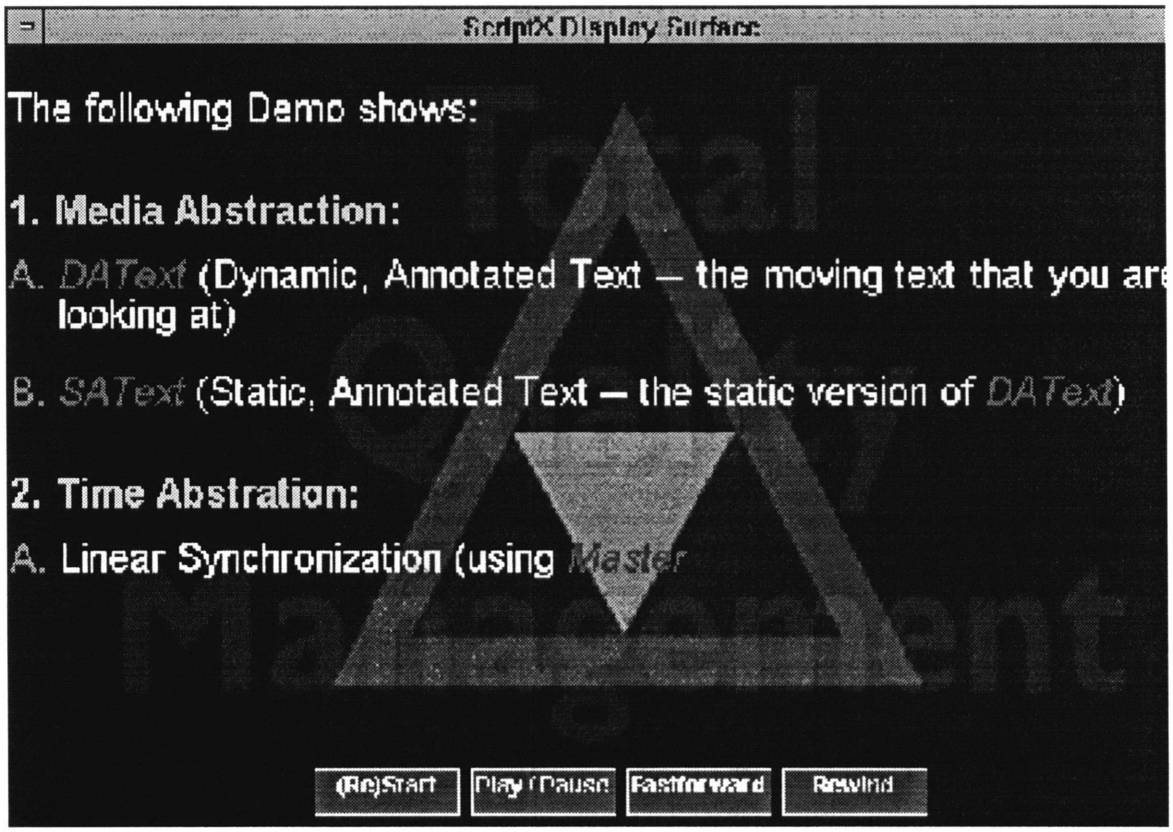
Chapter 6.

A Practical Example Using the Abstraction Models

A sample multimedia presentation was developed and implemented in ScriptX to demonstrate the integrated usage of all three abstraction models developed in this thesis. Specifically, the `BackgroundFetching`, the `MasterSlaveContract`, and the `SAText` and `DAText` classes were used. The sample course uses materials taken from the preview tape of the video course "A New American TQM: Revolutions in Management." [19]

Scene 1

The following figure shows the appearance of Scene 1:



The following Demo shows:

1. Media Abstraction:

A. DAText (Dynamic, Annotated Text – the moving text that you ar
looking at)

B. SAText (Static, Annotated Text – the static version of DAText)

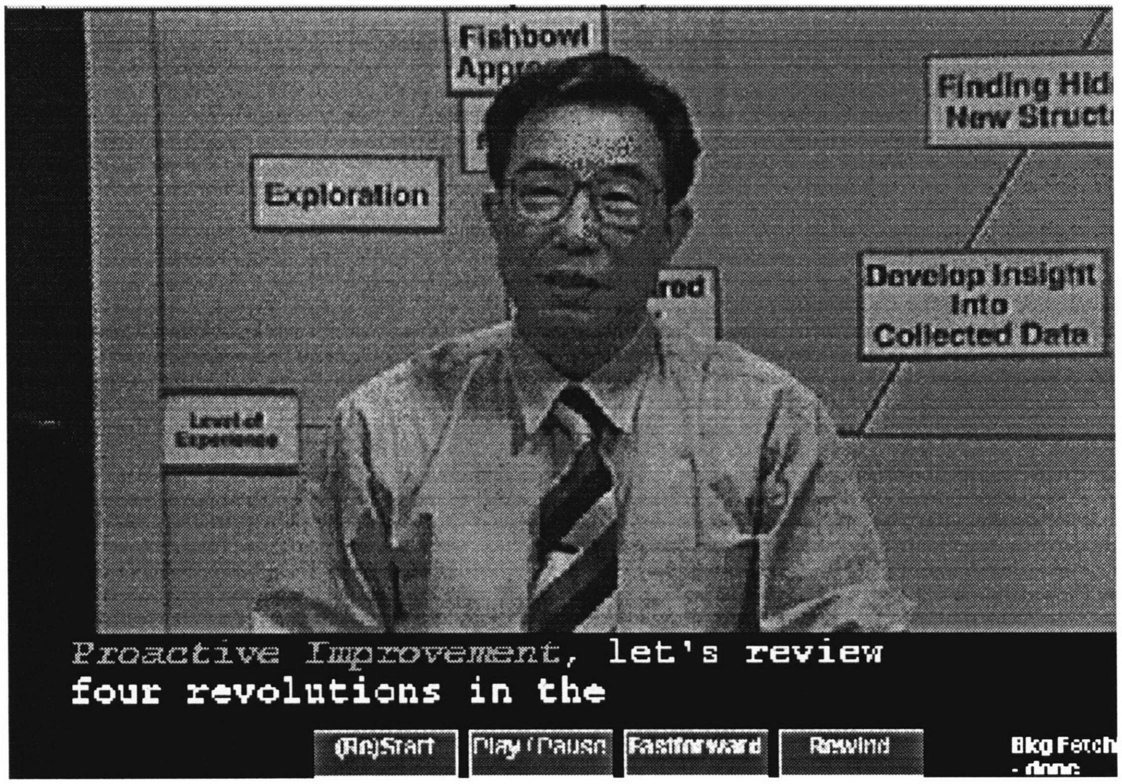
2. Time Abstration:

A. Linear Synchronization (using Master

Figure 6.1: The Appearance of Scene 1

Scene 1 is the introduction. A bitmap for the courseware is rendered in the background. A `DAText` object runs in the foreground at a constant rate throughout scene1. The `DAText` was created from an annotated source file included as Appendix A. At the bottom of the window are four push-buttons. The buttons exist throughout all scenes in the presentation and are used to control the entire presentation . The functions are: start/restart, play/pause, fastforward and rewind. The buttons were created using ScriptX's `PushButton` class. The class activates a function to detect mouse clicks. Each pushbutton function controls the master and, by extensions all of the slave objects. The start/restart button sets the master's time to 0. The play/pause button sets the master's rate to 1/0. The fastforward button sets the top player's time 5 seconds ahead. The rewind button sets the top player's time 5 seconds backward.

Scene 2



Proactive Improvement, let's review
four revolutions in the

Figure 6.2: The Appearance of Scene 2

Scene 2 and scene 3 are lecture materials. In scene 2, three pieces of media are rendered and synchronized spatially and temporally:

1) the lecturer's picture as a bitmap rendered in the background,

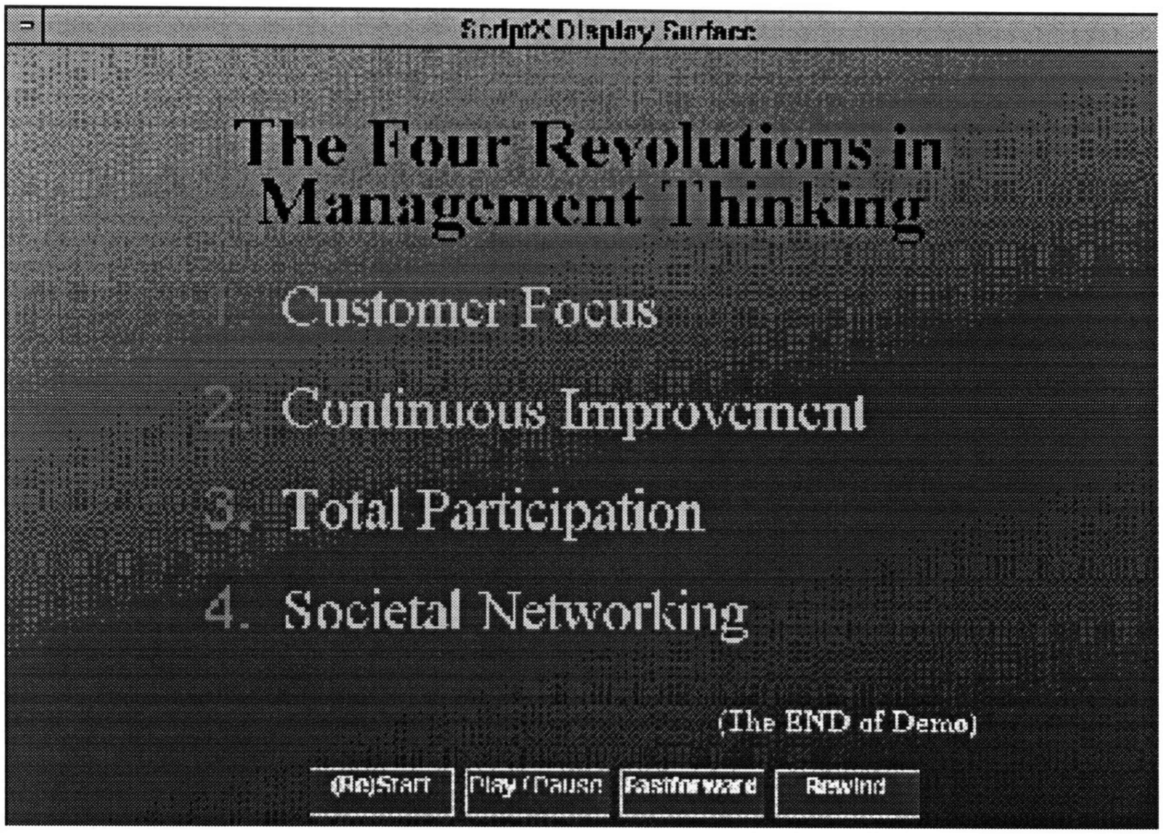
2) a DAText object serving as closed-caption text in the foreground, and

3) the lecturer's voice running continuously as an audio stream.

The closed-caption text is synchronized as a slave to the audio stream so that each word in the text stream is rendered when the audio stream utters that word. The synchronization is done using a MasterSlaveContract object created from a timing specification file shown in Appendix A.

During scene 2 playback, a BackgroundFetchingAgent object is instantiated to retrieve a file from over the network that will be used in scene 3. The audio playback apparently is not affected by the fetching, since it is automatically set to run at priority by the ScriptX system. If the DAText object playback in scene 2 is run with a thread with normal priority, it slows noticeably during the background fetch to support scene 3 because the fetch, run as a separate thread, is sharing CPU cycles. However, if the DAText object is being run as a thread with high priority, there is no noticeable delay.

Scene 3


ScriptX Display Surface

The Four Revolutions in Management Thinking

1. Customer Focus
2. Continuous Improvement
3. Total Participation
4. Societal Networking

(The END of Demo)

Figure 6.3: The Appearance of Scene 3

The bitmap file fetched during scene 2 is rendered in the background in scene 3. A `DAText` object runs in the foreground to provide the text for the slide. The lecturer's voice is rendered in an audio stream. The text is synchronized with the audio as in scene 2, using a `MasterSlaveContract` object. The audio serves as the master; the `DAText` object as the slave. The `MasterSlaveContract` object is created from a timing specification file included in Appendix A. Note that scene 2 and scene 3 use the same timing specification file. The `SectionID` instance variable of the `Master-SlaveContract` class indicates where in the file the timing data for each scene is located. For example, the `MasterSlaveContract` object in scene 3 uses "[scene3]" as value to the `SectionID` instance variable (see Appendix A.4).

Chapter 7

Conclusion and Future Work

7.1 Conclusion

With the abstraction models proposed in this thesis, the extended ScriptX platform has demonstrated to satisfy all of the desired capabilities as discussed in section 1.1 for supporting networked, interactive multimedia. The `BackgroundFetchingAgent` class in the *Data Flow Abstraction* provides ScriptX developers with a network API for performing background fetching operations. The example in chapter 6 using the `BackgroundFetchingAgent` API shows that data can be fetched silently from remote servers without disturbing the presentation running in the foreground provided that running threads that require attention have been set to high priority. The `MasterSlaveContract` class in the *Temporal Synchronization Abstraction* has leveraged the existing ScriptX support in timing to support non-linear synchronization enabling the orchestration of media playback where rates change arbitrarily over time. The `SAText` class in the *Text-based Media Abstraction* provides a means for creating formatted text media in ScriptX from annotated text files. The `DAText` class enables the convenient creation of text stream media which can be rendered dynamically. The example application in chapter 6 has demonstrated that all these abstraction models have been successfully implemented.

7.2 Future Work

For the *Data Flow Abstraction*, hostname resolution is a priority to enhance the existing functionalities of the `BackgroundFetchingAgent` API. At present, the *Data Flow Abstraction* is defined to support background prefetching. Several other important features in the area of data flow are worth added to enlarge the scope of the model. A streaming protocol and buffering to allow live media (e.g. a live video stream) to be played over the network on a ScriptX client are two such enhancements. Live media has advantages for efficient usage of bandwidth and local storage as with the prefetching model. Moreover, compression and decompression can be deployed in future research. Compressed data files can be stored on the servers and decompression by the ScriptX client as part of the retrieval process.

The *Temporal Synchronization Abstraction* currently handles media playback synchronization in a presentation. Protocols to synchronize network fetching with media playback can be developed.

The `SAText` class presently recognizes command strings that set the attributes of text in ScriptX. More innovative command strings can be implemented to create special media effects to the text content. For instance, `@blink.rate{`*foobar*`}` may cause *foobar* to blink with a rate `rate` in the presentation.

References

[1] "Latest Internet Host Survey Available: The Internet is Growing Faster than Ever." *Press Release, Internet Society, Reston VA, USA.*, 6th February, 1995. Available online at *http://www.nw.com/zone/WWW/isoc-pr-9501.txt*

[2] "The National Information Infrastructure: Agenda for Action.", SunSITE-based Government Documents. Available online at *http://sunsite.unc.edu/nii/NII-Table-of-Contents.html*

[3] "NCSA Mosaic Home Page". National Center for Supercomputing Applications, The University of Illinois at Urbana Chanpaign, 18th, March 1995. Available online at *http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/help-about.html*

[4] "Welcome to Netscape". Netscape Communications Corporation. Available online at *http://www.mcom.com/*

[5] "CGI Overview". Available online at *http://hoohoo.ncsa.uiuc.edu/cgi/*

[6] Dave Thompson. "Common Client Interface Protocol Specification". National Center for Supercomputing Applications, The University of Illinois at Urbana Chanpaign, February, 1995. Available online at *http://yahoo.ncsa.uiuc.edu/mosaic/cci.spec.html*

[7] Paul Rohr, "Software Development Interface", 2nd March, 1995. Available online at *http://www.spyglass.com:4040/newtechnology/integration/iapi.html.*

[8] "OLE Automation in Netscape.", Netscape Communications Corporation, 22nd March, 1995. Available online at *http://home.mcom.com/newsref/std/oleapi.html.*

[9] Boss B. "An API for WWW Applets". Draft with annotations for W3A version 1.1, 28th February, 1995. Available on line at *http://grid.let.rug.nl/~bert/W3A/W3A.html*

[10] "ScriptX Develop's Guide", ScriptX Technical Reference Series, Version 1.0, Kaleida Labs, Inc., 1994.

[11] Arthur Dumas, "Programming WinSock", SAMS Publishing, 1995.

[12] Mark Towfiq, "Frequently Asked Questions About Windows Socket Version 1.1", 6th September, 1994. Available online at *ftp://SunSite.UNC.EDU/pub/micro/pc-stuff/ms-windows/winsock/FAQ*

[13] "WATCOM C/C++^{32} User's Guide", 1st Edition, WATCOM International Corporation, Waterloo, Ontario, Canada, 1993.

[14] "ScriptX Architecture and Components Guide", ScriptX Technical Reference Series, Version 1.0, Kaleida Labs, Inc, 1994.

[15] “The ScriptX Core Classes Reference”, ScriptX Technical Reference Series, Version 1.0, Kaleida Labs, Inc, 1994.

[16] Nathaniel S. Borenstein, “Email with A Mind of Its Own: The Safe-Tcl Language for Enabled Mail.”, ULPAA 1994 Conference Proceedings.

[17] Stan. Letovsky, “ccitcl: Safe Tcl + CCI.” Available online at *http://gdbdoc.gdb.org/letovsky/tcl/ccitcl.html.*

[18] “The Java Virtual Machine Specification.”, Sun Microsystems, 15th March, 1995.

[19] Shoji Shiba, “A New American TQM: Revolutions in Management.”, MIT/CAES Video Productions.

[20] Stan. Baron; W. Robin Wilson, “MPEG Overview.”, SMPTE Journal v. 103 p. 391-394. June 1994.

Appendix A

Materials for Preparing the Example in Chapter 7

A.1 The `DAText` source file in Scene 1

The following source file is used to create the DAText object in scene 1.

```
@font.Courier New.{@size.20{@Brush.255:255:255{The following Demo shows:

@weight.bold{@Brush.250:250:70{1. Media Abstraction:}}
@leading.20{@indent.25{@Brush.50:255:50{A.}
@Brush.255:50:50{@style.italic{DAText}} (Dynamic, Annotated Text--the moving
text that you are looking at)
@Brush.50:255:50{B.} @Brush.255:50:50{@style.italic{SAText}} (Static,
Annotated Text -- the static version of
@Brush.255:50:50{@style.italic{DAText}})}}

@weight.bold{@Brush.250:250:70{2. Time Abstraction:}}
@leading.20{@indent.25{@Brush.50:255:50{A.} Linear Synchronization (using
@Brush.255:50:50{@style.italic{Master-Slave Offset}})
@Brush.50:255:50{B.} Nonlinear Syn. (using
@Brush.255:50:50{@style.italic{Master-Slave Contract}})}}
==> Be able to FF/Rewind/Jump to any position in the content.}}}
```

A.2 The `DAText` source file in Scene 2 for the Closed-caption Text

```
@paraleading.10{@Brush.255:255:255{@font.Courier
New{@alignment.flush{@leading.40{@size.20{@weight.bold{Today, I want to
speak of the @Brush.50:255:50{@style.italic{Proactive
Improvement}}.  But before going to the
@Brush.50:255:50{@style.italic{Proactive Improvement}}, let's review
four revolutions in the Management Thinking.   }}}}}}}
```

A.3 The `DAText` source file in Scene 3 for the Slide's Text

```
@post.{@leading.15{@size.36{@alignment.center{@brush.5:0:150{@font.Times New
Roman{@weight.bold{The Four Revolutions in}}}}}}
@size.36{@alignment.center{@brush.5:0:150{@font.Times New
Roman{@weight.bold{Management Thinking}}}}}}}@pause.5{}

@play.10{}@size.30{@alignment.flushleft{@brush.84:240:90{@size.30{1.}}
@font.Times New Roman{@brush.250:250:70{@post.{Customer}
@post.{Focus}}}}}@pause.5{}

@size.30{@alignment.flushleft{@brush.84:240:90{@size.30{2.}}  @font.Times
New Roman{@brush.250:250:70{Continuous Improvement}}}}@pause.5{}

@play.10{}@size.30{@alignment.flushleft{@brush.84:240:90{@size.30{3.}}
@font.Times New Roman{@brush.250:250:70{Total Participation}}}}@pause.5{}

@size.30{@alignment.flushleft{@brush.84:240:90{@size.30{4.}}  @font.Times
New Roman{@brush.250:250:70{Societal Networking}}}}@play.5{}
@size.16{@font.Times New Roman{@weight.bold{
@alignment.flushright{@brush.250:250:250{(The END of Demo)}}}}}
```

A.4 Timing Specification file for slaves in scene 2 and scene 3

The following file shows the timing specification file used by the `MasterSlaveContract` class in both scene 2 and scene 3 to synchronize the `DAText` objects (as the slaves) to the audio (as the master). "[scene2ccap]" is for scene 2; "[scene3]" for scene 3.

```
-- Filename: expspec.ini
--    the specification file for MasterSlaveContract in scene 2 and 3

-- format: [MSCSPEC]
           initialOffset;
           start,finish,ticks,rate;
           .....
[scene2ccap]
0,0;
0,5,0,0;
5,50,9,0;
50,100,18,0;
100,200,32,0;
200,300,41,0;
300,457,55,0;
457,565,68,0;
565,643,80,0;
643,701,92,0;
701,807,105,0;
807,1000,118,0;
1000,1168,124,0;
1168,1276,135,0;
1276,1327,142,0;
1327,1380,153,0;
1380,1500,163,0;

[scene3]
0,0;
0,390,2,0;
352,402,6,0;
402,452,10,0;
452,538,22,0;
538,648,48,0;
800,900,63,0;
900,1200,68,0;
1200,1350,87,0;
1350,1580,100,0;
1620,1750,112,0;
1750,2050,126,0;
2100,2200,141,0;
2200,2300,155,0;
2500,2500,156,3;
```

6967-26