

A Streaming Multimedia Extension for the X Window System

Helge Bahmann

November 27, 2002

This work analyzes the requirements of multimedia applications on the underlying graphics system. It further develops a semantics suitable for multimedia applications in networked graphics systems and presents a reference implementation of these concepts on the basis of the X Window System.

Contents

1	Preface	6
2	QuickTime	8
2.1	Components and the component manager	8
2.2	Image and sequence decompression	11
2.3	Timing	13
2.4	Movie toolbox	14
3	Anatomy of X11	16
3.1	Drawing model	18
3.2	X resources and XIDs	19
3.3	The X protocol	20
3.4	The Xlib client library	23
3.5	The X server	25
3.6	X extensions	26
3.7	XFree 86 extension modules	28
3.7.1	Module loader	28
3.7.2	Protocol extensions	29
4	Networked multimedia	31
4.1	Requirements	31
4.1.1	Network transparency	31
4.1.2	Real-time constraints	35
4.1.3	Abstraction level	35
4.1.4	Extensibility	36

4.1.5	Security considerations	36
4.2	Architecture	37
4.2.1	Image decompression	38
4.2.2	Timing	40
5	Prototype implementation	44
5.1	Xdv extension	45
5.1.1	Goals and design requirements	45
5.1.2	Implementation	46
5.1.3	Xdv client API	48
5.1.4	Server-side implementation	51
5.2	MPEG 1 codec	53
5.2.1	Structure of MPEG-1 video	54
5.2.2	Implementation	54
5.2.3	Client interface	55
5.3	Xtime extension	56
5.3.1	Implementation strategy	56
5.3.2	General implementation notes	57
5.3.3	Schedulers and clocks	57
5.3.4	Xtime client API	58
5.3.5	Server-side implementation	60
6	Sample Applications	62
6.1	A simple MPEG1 player application	62
6.1.1	Initialization	62
6.1.2	Setting up the player and the mainloop	63
6.1.3	Transmitting frames	64
6.1.4	Frame playback	66
6.1.5	Analysis of the sample player	67
6.2	Improved MPEG1 player	67
6.2.1	Backbuffers	71
6.2.2	Picture decoding and playback	71
6.2.3	Transmitting frames	72
6.2.4	Setting up the player and the mainloop	74

7 Discussion	75
7.1 Prototype implementation	75
7.1.1 Xdv extension	75
7.1.2 Xtime extension	80
7.2 Experiences gained through the prototype	82
7.3 Related works	82
7.4 Summary	83
7.5 Further work	84
Bibliography	84
A Xdv client API	87
B Xtime client API	97

1 Preface

The initial motivation for this work resulted from an analysis of the data flow between existing software video players (or more generally speaking any application dealing with compressed digital video) and the underlying graphics system (specifically X11). Usually the decoding is performed in the context of the player process, while the graphics system is largely ignorant of the semantics associated with the compressed video frames and is used only for the final blit operation to the visible area of the screen (or more generally to make the resulting pictures visible to the user, which may also be accomplished by using overlays).

The strategy outlined above works reasonably well if the communication channel between player and graphics system operates with relatively low and reliable latency and offers sufficiently high bandwidth to transport the decoded frames. These requirements are usually met if both, the decoder and the graphics system, run on the same physical machine and use fast inter-process communication mechanisms such as shared memory for data transport.

This approach however fails if the requirements outlined in the previous paragraph cannot be met, which is the case when decoder and player have to communicate over a slow network link (compared to shared memory and ipc mechanisms). Trying to apply the same strategy to this scenario will cause significant load on the network and still yield very unsatisfactory results, caused by both the limited capacity and unpredictable latency of today's networks.

The obvious question is how this situation can be improved. Commonly network graphics systems use compression to reduce the amount of data transferred between the application and the display system, thus reducing network traffic. This however solves only the first part of the problem, the large amount of data required to describe a sequence of images, but fails to address the latency issue, or even makes the latency problem worse as compression inherently introduces delay. Moreover it will result in a great waste of resources as the pictures are first decompressed, then again compressed (generally in a far less space-efficient fashion than the

original representation) and finally decompressed again. Obviously it would instead be desirable to perform the decompression only once, after the transport step.

This paper explores ways to extend the semantics of networked graphics systems (specifically the X Window System) to provide a basis suitable for multimedia applications. Although it would be possible to build a “networked multimedia system” to complement the graphics system side-by-side, it is the authors believe that a firm coupling between the two is required anyways, and that the extension is both natural and potentially beneficial to a very broad range of applications.

The main goal is to demonstrate the feasibility of this approach, extending a networked graphics system by image decompression and timing functionality to provide streaming media services. Further it tries to evaluate the suitability of the X Window System for this extension, and what architectural obstacles have to be overcome. Due to limited time the results presented here had to be restricted to video services only; audio has been left out almost completely, although the impact of adding audio has been taken into consideration from the beginning throughout the whole architecture.

The paper is structured as follows: In the first chapter following this introduction a brief overview of the architecture of an existing, non-networked multimedia system (QuickTime) will be given. The next chapter takes a look at the X Window System, to provide the basis for understanding the reference implementation and to serve as a reference for network graphics systems. Next the requirements and an architecture for a networked multimedia system will be derived and explained. In the following chapter the reference implementation itself will be discussed in great detail, showing the impact on and interaction with the core X architecture. In chapter six some code samples will demonstrate how the previously developed extensions can be used in practice for networked multimedia applications, and specifically how they help in solving the initial motivating problem for this paper: building a networked video player. The paper closes with a discussion to what extent the stated goals could be achieved, which areas remain problematic, and how the approach taken here compares to other implementations.

2 QuickTime

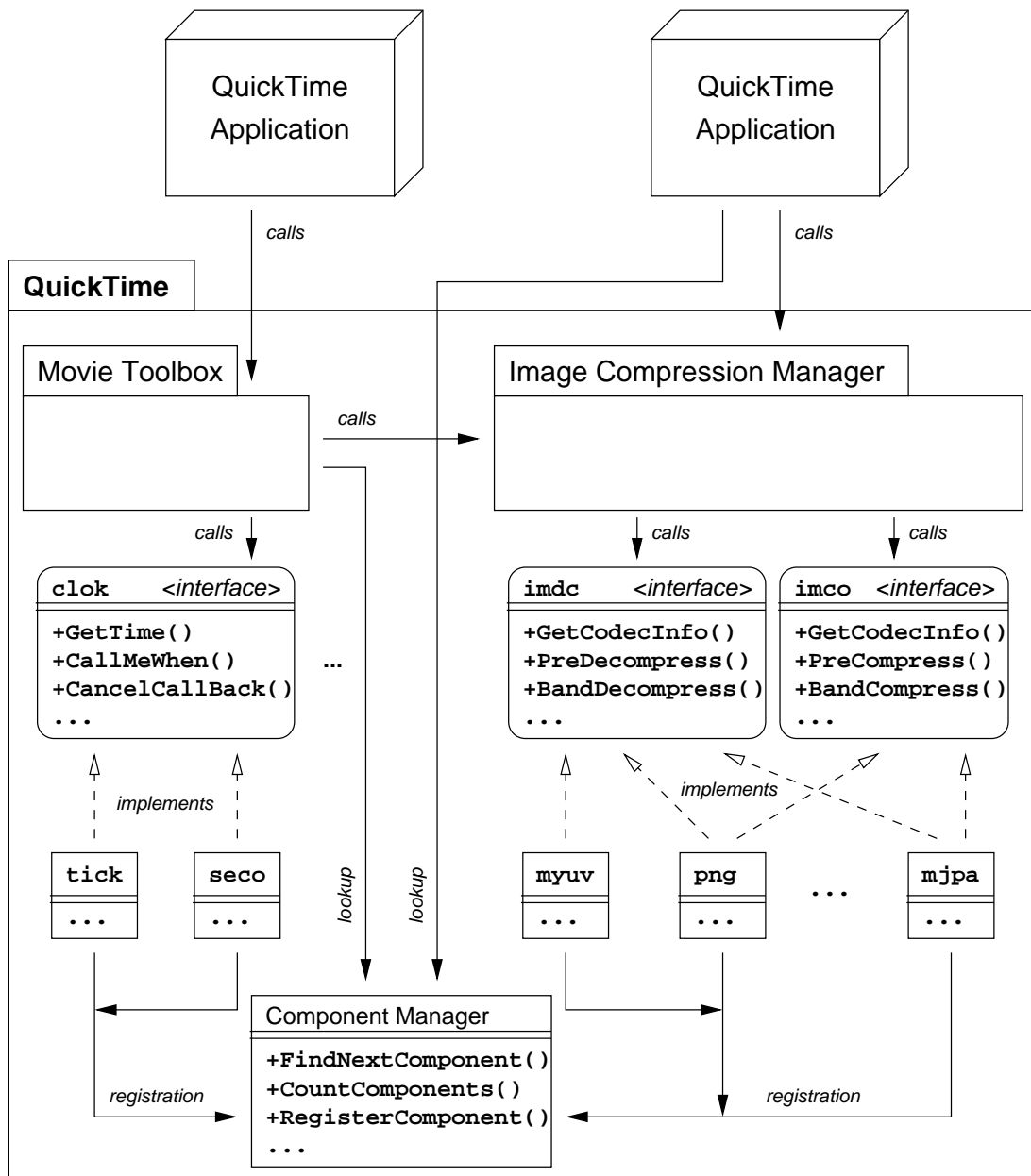
This chapter will take a short tour through the architecture of QuickTime. QuickTime was announced by Apple Computers Inc. in 1991 and released later in the same year. It was designed to provide a framework and toolbox for dealing with all kinds of time-based data (hence its name), and over the time it has evolved into one of the most comprehensive multimedia frameworks available, offering a broad range of services ranging from compression and decompression of still images and video clips, to video capturing and editing functions. Not the whole QuickTime architecture can (and for understanding the following chapters need) be covered in this thesis, further information can be obtained from the QuickTime documentation in [1] and [2].

The purpose of this chapter is to show the design considerations which were made in existing multimedia systems. These considerations will serve as a guideline for the development of a networked multimedia system in chapter 4.

This paper is concerned only with a small subset of the services provided by QuickTime: playback of video streams. Consequently the discussion in this chapter is limited to the low-level mechanisms provided by QuickTime that are relevant to video playback. Other services offered by QuickTime, such as video recording and editing, will therefore not be discussed.

2.1 Components and the component manager

QuickTime features a highly modular architecture to provide its various services to multimedia applications (see figure 2.1). The functionality provided by the services can be accessed by client applications through various *managers*, which are a set of functions and structures dealing with one specific aspect of multimedia programming; i.e. the *Image Compression Manager* provides functions for compressing and decompressing images.



Interaction of QuickTime applications with the basic building blocks that form the QuickTime multimedia architecture

Figure 2.1: UML diagram of the QuickTime component architecture

The central part of QuickTime is the *Component Manager*, which acts as a repository for registered QuickTime *components*. Components in the sense of QuickTime are pieces of software providing a defined interface and whose functionality can be transparently used by the client application. The purpose of components is to obviate the need for applications to know about every possible type of image, audio or video format, or the specifics of hardware devices, e.g. for video acquisition. For example, QuickTime provides multiple image decompressor components that share a common interface, but each component is specialized to handle one particular type of image format.

Components are identified by three four-letter-codes: the component type, subtype and manufacturer code. The *component type* identifies the kind of services the component provides. Examples include 'imdc' for image decompressor components (responsible for reconstruction of images from a compressed representation), 'clock' for clock components (providing a source of time information), 'vdig' for video digitizer components. Additionally the component type identifies the set of API functions that a particular component can support, e.g. all 'imdc' components provide the `BandDecompress` function.

The component *subtype* further specifies the implementation provided by the component. For image decompressor components this is usually the compression algorithm or format used, e.g. 'png' deals with still images in the Portable Network Graphics¹ format, 'mjpa' deals with Motion JPEG A pictures, and 'myuv' with MPEG-1² video images. Finally the *manufacturer code* identifies the vendor of a particular component. Notably some components are dual-natured in that they provide multiple types of services – e.g. the 'png' component provides functionality to both compress and decompress of images in the PNG format.

Technically seen the interface to the components is provided through a single dispatcher function. The first argument supplied to the dispatcher by the caller is a request code, referred to as *code selector* in QuickTime, which identifies the specific type of service requested by the caller; the interpretation of the remaining arguments then depends on the service type requested. The code selectors are specific to each component type, e.g. the selectors provided by 'imdc' and 'imco' components are listed in table 2.1.

The purpose of the *Component Manager* is to act as a “mediator” between applications and components. It allows components to register themselves and advertise their services to applications through the `RegisterComponent` function, and it allows clients to look for

¹see [7]

²see [3]

```

enum {
    kImageCodecGetCodecInfoSelect          =0x00, /*CDGetCodecInfo*/
    kImageCodecGetCompressionTimeSelect    =0x01, /*CDGetCompressionTime*/
    kImageCodecGetMaxCompressionSizeSelect =0x02, /*CDGetMaxCompressionSize*/
    kImageCodecPreCompressSelect           =0x03, /*PreCompress*/
    kImageCodecBandCompressSelect          =0x04, /*BandCompress*/
    kImageCodecPreDecompressSelect         =0x05, /*PreDecompress*/
    kImageCodecBandDecompressSelect        =0x06, /*BandDecompress*/
    kImageCodecCDSequenceBusySelect        =0x07, /*CDSequenceBusy*/
    kImageCodecGetCompressedImageSizeSelect=0x08, /*CDGetCompressedImageSize*/
    kImageCodecGetSimilaritySelect         =0x09, /*CDGetSimilarity*/
    kImageCodecTrimImageSelect             =0x0A /*CDTrimImage*/
};

```

Table 2.1: Request codes for imdc components

components qualified by application-defined constraints (e.g. by component type 'imdc', or by component type 'imdc' and subtype 'png ').

While applications can directly call the component dispatcher functions to request services from a particular component, they should generally make use of the various manager APIs provided by QuickTime instead. For example, the *Image Decompression Manager*, which will be covered in the following section, wraps the services provided by 'imdc' and 'imco' components in a set of convenience functions.

2.2 Image and sequence decompression

Image³ compression and decompression in QuickTime is performed by 'imco' and 'imdc' components, respectively. They provide compressors and decompressors for various still and video image encodings, and make dealing with different formats largely transparent to the calling application.

³The terms *image* and *picture* are commonly used in the context of still images, while *frame* usually refers to the individual pictures that form a video sequence; the distinction between these terms is blurry and they are used interchangeably within this paper, but the term that is most commonly used in a specific context will be preferred

The API provided by image compressor and decompressor components is straight forward. The `kImageCodecPreCompressSelect` and `kImageCodePreDecompressSelect` code selectors are used to perform preparatory steps necessary before compression or decompression can begin. This includes setting up parameters, such as desired quality or special options, and allocation of temporary buffers. The passed parameters are checked whether the selected options can be supported by the underlying implementation.

Image decompression itself is performed by calling the dispatcher function with the `kImageBandDecompressSelect` code selector. In this call the caller has to supply a band⁴ of compressed image data and specify a target where the reconstructed image band should be written to. This function must be called iteratively until the full image has been reconstructed. Image compression is performed in a completely analogous fashion.

The image compression manager provides convenience functions to hide the details of the calls into the (de)compressor component. For still images, the `CompressImage` and `DecompressImage` functions perform all steps necessary for the compression or reconstruction of an individual image, respectively. For sequences of images, the image compression manager provides the `CompressSequenceBegin`, `DecompressSequenceBegin` to setup a context suitable for dealing with the image sequence, while the `CompressSequenceFrame` and `DecompressSequenceFrame` functions perform the desired operation on an individual picture of the sequence.

Image decompressor components specifically targeted at compressed video images need to distinguish between key-frames and non-key-frames. Key-frames are pictures that can be decoded independently from the context they are in, i.e. without access to other pictures in the sequence; non-key-frames are encoded relative to other pictures in the video sequence, employing the temporal redundancy between images to achieve better compression.

The simple model of the `imdc` components however does not offer a way to make decoding dependencies implied by key-frames and non-key-frames explicit. Instead this dependency is implicitly determined by the order the pictures are handed over to the decompressor component. When applications want to decompress a contiguous sequence of pictures, this additional complexity remains hidden inside the Image Compression Manager. However it is the task of the application to provide the required frames in the correct order, if it wishes to extract a single picture from a sequence.

⁴a horizontal strip of the image

```

enum {
    kClockGetTimeSelect          = 0x1, /* ClockGetTime */
    kClockNewCallBackSelect      = 0x2, /* ClockNewCallback */
    kClockDisposeCallBackSelect = 0x3, /* ClockDisposeCallback */
    kClockCallMeWhenSelect       = 0x4, /* ClockCallMeWhen */
    kClockCancelCallBackSelect   = 0x5, /* ClockCancelCallback */
    kClockRateChangedSelect      = 0x6, /* ClockRateChanged */
    kClockTimeChangedSelect      = 0x7, /* ClockTimeChanged */
    kClockSetTimeBaseSelect      = 0x8, /* ClockSetTimeBase */
    kClockStartStopChangedSelect = 0x9, /* ClockStartStopChanged */
    kClockGetRateSelect          = 0xA  /* ClockGetRate */
};

```

Table 2.2: Request codes for `clock` components

With respect to the MPEG video decoder discussed later, bi-directionally predicted frames (B-frames) are of special interest here, as they are encoded relative to a temporally preceding and a temporally succeeding frame. Decoding of B-frames introduces another complication as the order in which pictures have to be decoded does no longer correspond to the temporal ordering of their presentation. The author could not definitely determine where in the QuickTime architecture this additional complexity is handled. The lack of expressiveness of the interface provided by both the `imdc` components and the Image Compression Manager make it fair to assume that this functionality is provided in higher layers of the QuickTime architecture.

2.3 Timing

In QuickTime time information is provided through clock components (component type `clock`). They provide an abstraction of various kinds of time sources, including the system tick timer, or timers derived from audio DAC and ADC devices.

The time measured by a clock component does not necessarily relate to UTC time: e.g. if an audio stream sampled at $44.1kHz$ is used as a clock source, its native time scale would be 44100 ticks per second, however a small deviation from this nominal rate is perfectly legal. QuickTime accounts for this deviation by making sure that all parts of a presentation are synchronized to the same clock source, and thus synchronized to each other.

Clock components provide two basic services: Firstly, they allow to get the current time; this is accomplished using the `ClockGetTime` API function, which maps to the `kClockTimeGetTimeSelect` code selector (cf. table 2.2) of the corresponding component. The function returns the time scale (number of ticks per second) in which time is measured as well as the timer value itself.

The second important service provided by clock components is to schedule callback functions; this is accomplished using the `ClockNewCallback` and `ClockCallMeWhen` API functions, which again map to the corresponding code selectors of the component. They allow to setup periodic or one-shot timer callbacks which are to be called at a predefined rate or a predefined point in time, respectively.

QuickTime makes use of periodic callback functions to drive multimedia presentations. For example, to show a video clip, a periodic callback function would be setup, scheduled at a rate equal to the frame rate of the video. The callback function would then in turn call the image compression manager to drive decoding and display of the individual video frames.

Using the 'clock' abstraction layer for timing services provides a simple and precise mechanism for synchronization of video and audio components in a presentation; as audio implicitly carries very fine-grained timing information (the sampling rate of the audio stream), it is very well-suited to provide time information for the whole presentation, and makes synchronization between audio and video easy.

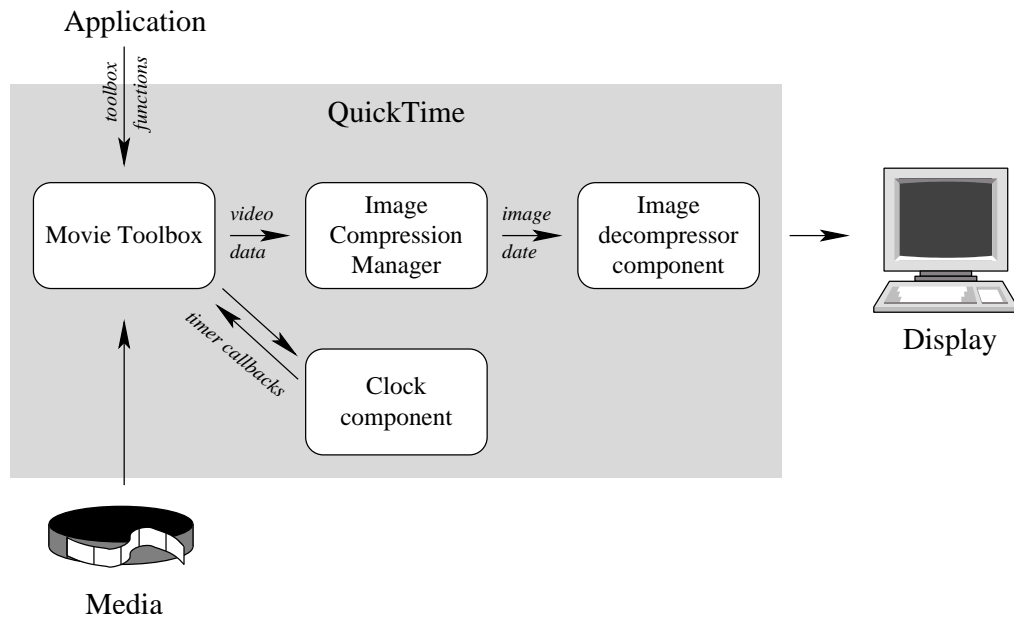
When encountering a presentation containing both video and audio, the QuickTime movie toolbox, discussed in the next section, will therefore choose the audio stream to provide the clock source for the whole presentation⁵. If no audio track is present, the system timer is used instead.

2.4 Movie toolbox

While the Image Compression Manager deals with (sequences of) images, the *Movie Toolbox* provides an even higher-level interface. As the name suggests, its main purpose is to deal with movies, which are presentations usually containing video, audio and sometimes also text media.

The services provided by the Movie Toolbox do not only cover playback of video clips, but also include functions for movie creation and editing. The toolbox contains a number of very

⁵unless this choice is explicitly overridden by the user



The movie toolbox provides a high-level interface to client applications; it hides the details of joining together the individual QuickTime components required in a presentation

Figure 2.2: Interaction of decompressor and clock components in the Movie Toolbox

high-level functions, that are capable of performing editing operations on a whole presentation in a single step.

The Movie Toolbox makes use of the other building blocks of the QuickTime architecture; e.g. when playing back a presentation containing a stream of compressed video images, the toolbox will make use of the Image Compression Manager to decompress the frames. Likewise, for timing and synchronization services it uses the interface provided by the 'clock' components. The interaction of the components discussed before inside the Movie Toolbox is depicted in figure 2.2 (audio is omitted from the diagram, except for the fact that the audio device may provide the clock component).

The API provided by the Movie Toolbox hides the complexity of the underlying components, applications wishing to play back a multimedia presentation will usually turn to the high-level functions in the Movie Toolbox instead of controlling each of the individual components themselves. However the main strength of the QuickTime architecture lies in the ability to hook into this complex processing pipeline at every stage through well-defined interfaces.

3 Anatomy of X11

The purpose of this chapter is to give a short introduction to the *X Window System*, also commonly referred to as *X11*. The architectural model of X11 will be discussed as well as the provided service. For a better understanding of the software implementation covered in chapter 5 a fair amount of technical detail is included.

Development of the X window system begun in 1984 at the MIT and saw an evolution of the protocol and core system until 1986 with the release of the version 11, hence the name X11. The basic architecture has remained almost unchanged since then, though a large number of extensions have been developed. Today it is the graphics and windowing system that is most commonly used in Unix environments, but implementations are available for nearly every platform.

X11 is architected as a client-server system and essentially consists of three parts: *X clients*, which are the applications containing the processing and control logic; the *X server*, which acts as the interface to both the graphics hardware and input devices, and contains the drawing logic; both communicate with each other using the *X protocol* (cf figure 3.1).

The client part is usually implemented in several layers: the low-level *Xlib* which mainly serves as a thin wrapper library to generate X protocol requests¹; on the next layer toolkit and widget libraries provide the application with high-level constructs such as buttons, menus and scrollbars, and perform dispatching of events to these widgets²; and on the next higher layer the actual application logic.

¹Of course the programmer is not required to use Xlib, but there is only one alternative implementation of the client side X protocol known to the author

²Traditionally there has been a clear distinction between toolkit (like *Xt*, the X intrinsics toolkit library) and widget library (like *Xm*, the X Motif library), but some toolkits blur this distinction and combine both into one layer

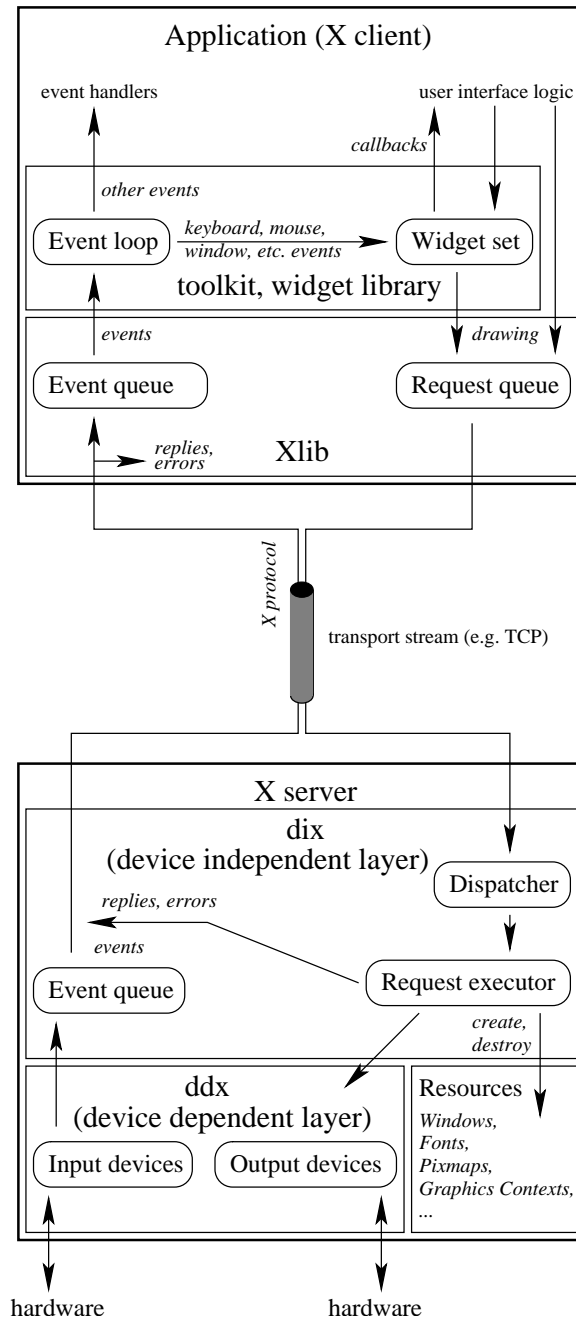
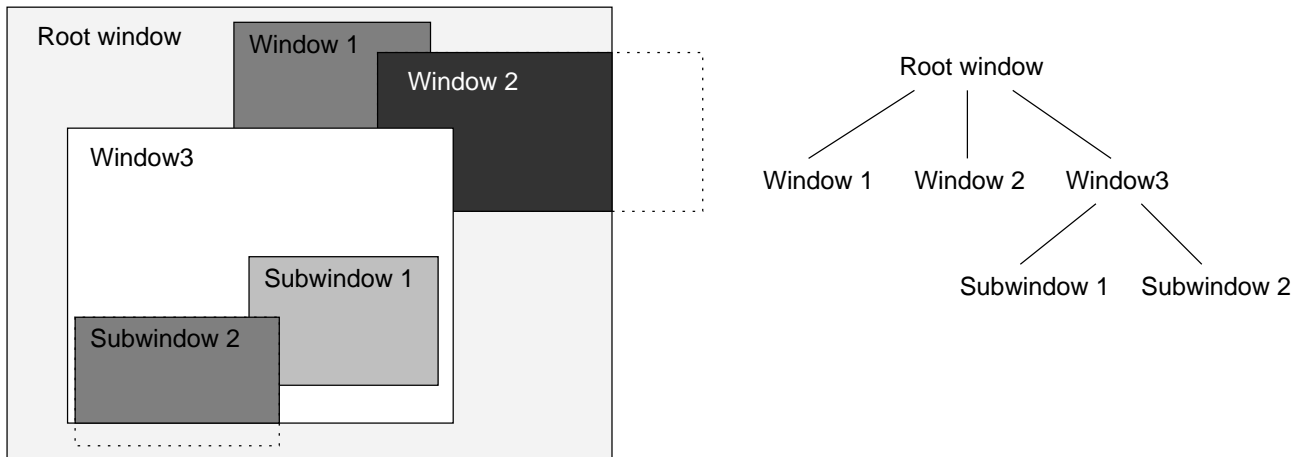


Figure 3.1: Architectural overview of the X window system



Windows at the same hierarchy level in the hierarchy tree on the right are stacked from left to right in the screen layout on the left.

Figure 3.2: Window hierarchy and screen layout

The server is architected in multiple layers as well and will be covered in more detail in section 3.5.

3.1 Drawing model

The main services provided by the X server are drawing operations, management of input devices, and inter-client communication. Input devices and inter-client communication are of little interest for the following, so discussion of these topics is omitted, but the drawing model of X11 is of interest and will be examined in detail.

Drawing in X11 is performed into either visible on-screen areas (called *Windows*) or invisible off-screen areas (called *Pixmap*s); as both windows and pixmaps can be the target of drawing operations, they are collectively referred to as *Drawables* in the X terminology. Drawables represent rectangular arrays of pixels, each of which can be individually addressed and set to a specific color.

Windows in X11 are organized into a hierarchy, with one special window, the *root window* which represents the whole screen, on top of the hierarchy (cf. figure 3.2). Each window determines a rectangular area within its parent window, so the direct descendants of the root window describe rectangles on the screen. Windows at the same hierarchy level may overlap,

and in this case visibility is determined according to the stacking order of the windows, e.g. in the example *window 3* is stacked on top of *window 2*, covering parts of the window. The dimension of a window may exceed the boundaries of its parent (indicated in the figure by the dotted boxes), in which case its visible area is clipped to the boundaries of its parent. When drawing directly into a window the client application need not take care whether the portion of the window in question is visible or covered, appropriate clipping is performed by the graphics system automatically.

The drawing services provided by the core X window system can be categorized in

1. *Geometric primitives*. This includes lines, polygons, circles, arcs and boxes
2. *Text*. This includes font management and rendering of text strings
3. *Still images*. This includes copying rectangular areas from one Drawable into another, and transfer of pre-rendered graphics between client application and server

Of these, 1 and 2 are most commonly used by applications to draw a graphical user interface. For video playback applications the only choice is to do the rendering of the images themselves and then make use of the services listed under 3.

The drawing model of X11 is mostly stateless, that is, all state information required to perform a drawing operation (e.g. color, or font) has to be explicitly given. To avoid unnecessary transfer of a large amount of (usually static) drawing parameters for every individual operation, context objects storing a set of drawing parameters are created and referenced in every drawing operation. These context objects are called *graphics contexts*, or short GCs.

3.2 X resources and XIDs

Client and server commonly have to exchange information related to resources allocated in the server; this is the case for example when the client wishes to initiate a drawing operation, to identify the target surface the operation is to be performed on; or if the server wishes to report an event that a particular window has been closed, it needs a way to identify the affected window to the client.

To refer to these kind of resources client and server use XIDs, which are essentially 32-bit numbers uniquely identifying a resource. Each resource has an associated type (which in turn

may be a specialization of a more abstract resource class) describing the operations it can be used in, e.g. a window may be the target of a drawing operation, but a font may not; all resources irregardless of type share a single XID space, so no two different resources existing at the same time can have the same XID. The most important resource types are **Windows**, **Pixmap**s and **GC**s as discussed in the previous section.

The XID values are allocated in the client when creation of a resource is requested. This saves one round-trip-time compared to server-side allocation as the client may immediately issue requests affecting the newly created resource without waiting for the creation to complete. This is important to deliver good performance even for network links with considerable latency, but the downside is that recovering from (the rare case of) allocation failures in the server is difficult at best – in practice the client will almost always simply terminate.

The XID space is also shared among multiple clients, so clients can access resources created by other X clients without restriction³. For this reason the most significant bits of every XID are used to identify the client creating this resource, so every client has a “private portion” in the XID space which no other client can allocate XIDs from.

Resources are usually destroyed when the X server detects that the connection to the client which created the resource is shut down; it is in some rare cases however desirable and possible to retain resources past termination of the client.

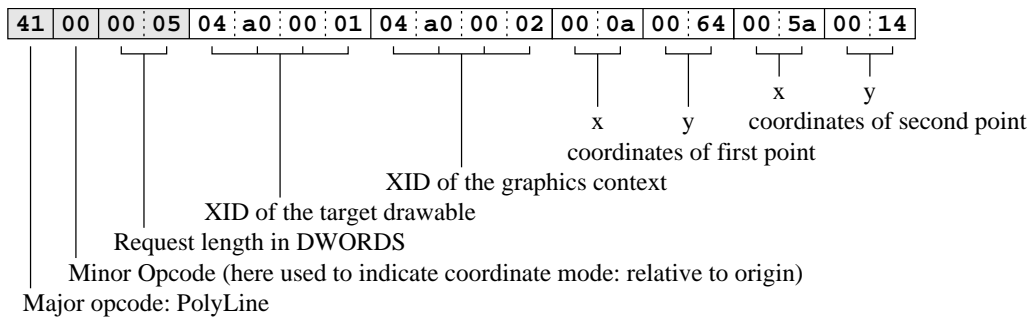
3.3 The X protocol

The X protocol is the remote procedure call marshaling protocol used for communication between the X client and the X server. It relies on a stream-oriented, reliable transport protocol; commonly TCP or (in the case of X client and X server being on the same machine) Unix domain sockets are used.

To save computational effort for the very common case of application and X server running on the same machine, the byte order used when transferring 16- or 32-bit values is that of the client; in case X client and X server are using different native byte order the server has to perform the required byte-swapping.

Each end communicates to the other via short, variable-length messages. Client messages sent to the server are **X Requests**, while server messages sent to the client are either **X Replies**,

³certain security extensions may however limit this access



Drawing a line between the points (10, 100) and (90, 20). Note that the number of points comprising the polyline is implicitly determined by the request length. Shaded area indicates fields common to all requests.

Figure 3.3: An exemplary X request

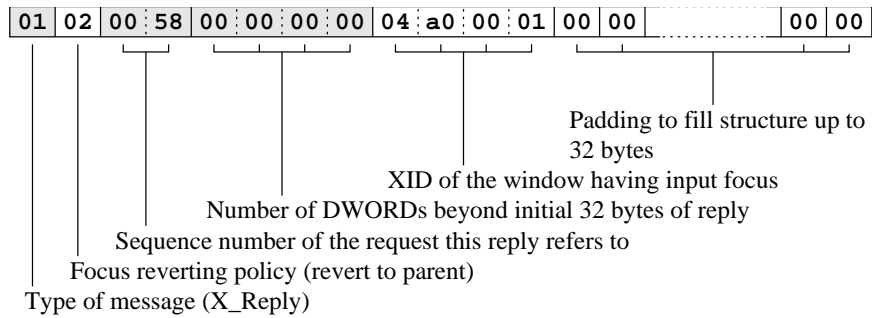
X Errors or X Events. The message types and its function will be presented in short form below.

Operations are initiated by the client sending an **X Request**, containing the (major) opcode of the requested operation, in some cases a minor opcode to further identify the operation, the total length of the request, and the specific parameters required for the operation. (The core protocol does not use a minor opcode to further distinguish requests, they are used by protocol extensions only.) Both client and server maintain a sequence number for every request; this sequence number is not sent along in the request, instead it is implicitly maintained by increasing a counter in both the client and the server for every request sent or processed.

All messages sent from the server to the client start with a byte identifying the type of message, a value of 0 indicating **X Error**, a value of 1 indicating **X Reply** and any other value indicating an **X Event** to follow (and at the same time determining the type of event).

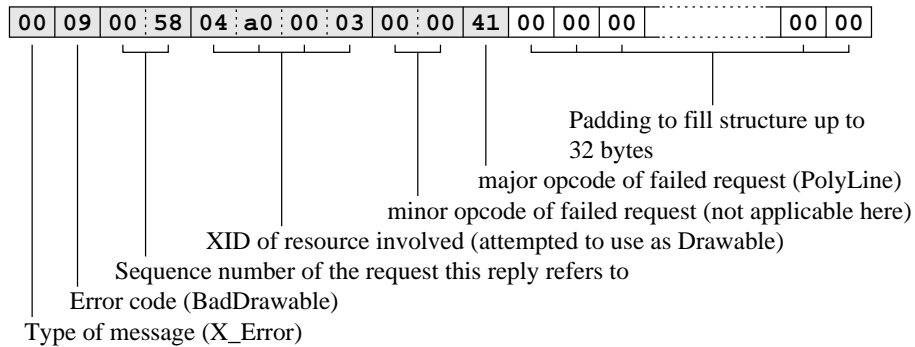
X Reply messages are only generated by the server in direct response to certain X requests, and they are used to send requested information back to the client. Each **X Reply** message contains the 16-bit sequence number of the request it refers to; it is used by the client to match the reply to the request that generated it. Replies are at least 32 bytes in size, but contain a field indicating the additional number of DWORDs that form this reply, resulting in a theoretical maximum length of $4(2^{32} - 1) + 32$ bytes in a reply.

In case an operation fails, the server sends an **X Error** to the client, indicating the request that failed (identified through its sequence number), opcode of the operation and reason for



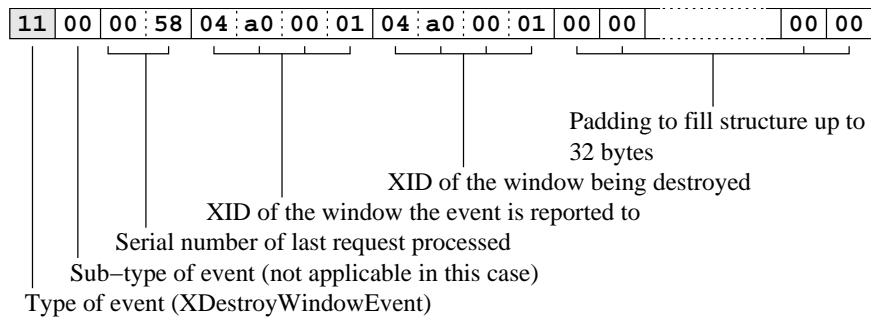
Layout of a reply to a `GetInputFocus` request; it indicates the window which currently has keyboard focus and where the focus reverts to if this window is no longer visible. Shaded area indicates fields common to all replies.

Figure 3.4: An exemplary X reply



The error resulting from an attempt to use an invalid (not allocated) resource as target for a drawing operation. Shaded area indicates fields common to all errors.

Figure 3.5: An exemplary X error



Structure of a DestroyWindowEvent; this event can be reported both to the window affected or to its parent window, hence the distinction between the two in this structure. Shaded area indicates fields common to all events.

Figure 3.6: An exemplary X event

the failure. Every X **Error** message is fixed at 32 bytes in size.

To asynchronously notify the client of events that occurred inside the X server or attached peripherals (e.g. key press or mouse movement) the X server can send an X **event** to the client. Events are fixed in size at 32 bytes and start with a byte describing the type of the event, followed by another byte further specifying a sub-type if applicable and a 16-bit integer containing the sequence number of the last X request processed from this client. The rest of the event structure is used to provide detailed information to the client and is specific to each event type.

As can be seen from this description the protocol itself is largely asynchronous in nature, requiring only few synchronization points where the client has to wait for a reply from the server⁴. This property is almost mandatory to provide satisfactory performance even in the case of relatively high communication latency between client and server.

3.4 The Xlib client library

Most X client applications use Xlib to interact with the X server. For this purpose Xlib provides a number of functions for generating the X protocol requests and dispatching of the

⁴It should however be noted that the most commonly used client-side protocol library, Xlib, does not utilize the full asynchronicity of the protocol; in fact it does not offer any interface to collect X **replies**, and every library function requesting data from the server will therefore synchronously wait for the reply.

replies generated by the server. For example, the request shown in figure 3.3 on page 21 was generated by

```
XDrawLine(display, window, gc, 10, 100, 90, 20);
```

All `Xlib` functions are structured similarly, taking an opaque display object as first argument, to identify the connection to the server. The first step in generating the protocol elements is to translate the passed objects (in this case `window` and `gc`) into the resource values assigned to them during creation. After that an appropriately sized buffer is allocated and the X request is composed of the parameter values passed by the caller.

The protocol requests are not sent directly to the server; instead, to reduce the number of required system calls, `Xlib` keeps a buffer for every open X server connection, into which all requests are written. As a result, simply calling `Xlib` drawing functions does not suffice to produce a visible result on the screen, but the client application has to ensure that the buffer is flushed and the data is transmitted to the server. Flushing occurs under the following conditions:

1. it is explicitly requested by the client through the use of the `XFlush` API function
2. the buffer is too full to hold an additional request when the client calls an X API function
3. the client uses an `Xlib` function that implicitly waits for a message from the X server
4. after generating each protocol request, if `Xlib` is requested to act in synchronous mode; this is often useful for debugging as it ensures that the display state matches the state of the client application

To support multi-threaded clients, `Xlib` keeps a lock per X server connection; this lock is used to serialize access to the request buffer.

Some operations expect a response carrying data from the server; for example calling the `XGetInputFocus` function will generate a `GetInputFocus` request and cause the server to generate an according reply (cf. figure 3.4). Although the protocol allows for this reply to be delivered asynchronously, the `XGetInputFocus` function will generate the protocol request, flush the request queue and will not return until this reply has been received. Though this greatly simplifies the programming model, it does not fully utilize the capabilities of the protocol.

The only server notifications that are treated completely asynchronously in the library are **X Events**; when received from the server, these will be queued, and the application can dispatch of these events one after another by using the **XNextEvent** and related functions.

3.5 The X server

An overview of the basic architecture of the X server is already included in figure 3.1 on page 17. As the figure shows, the server is built in two main layers: the device-independent layer (*dix*) and the device-dependent layer (*ddx*).

The tasks of the device-independent layer are interaction with the X clients (receiving and generating protocol elements, queueing), scheduling and resource management. The device-dependent layer interfaces to the hardware and performs all actions requested by the *dix* layer, e.g. execution of drawing requests and collection of events from input devices.

The following paragraphs will illustrate how *dix* layer of the X server interacts with the client and how requests are processed, to provide the basis for understanding the prototype implementation.

Resource management

One central task of the *dix* layer is to manage the X resources created in the server. To allow fast access to any resource, references to the resources are kept in a hash table, indexed by the **XID** assigned to the resource when it was created; further speed-up is achieved by caching the most recently requested resources.

Resources can be created on behalf of the client by calling the **AddResource** function. The caller has to supply the **XID** the client wishes to associate to the newly-created resource, the type of the resource being created, and an opaque pointer to the resource object. Resources are accessed by using the **SecurityLookup...** class of functions, which take an **XID** and an expected resource type as arguments and return the pointer to the corresponding resource object. The type is matched against the type that was specified when the resource was allocated, to provide type-safety when accessing resource objects.

Request dispatching

While a multi-threaded dispatcher model, e.g. using one thread per client or a pool of threads, would be possible, most X servers (including the XFree86 server used in this project) are written single-threaded and event-driven.

When a request is received, it is first read from the operating system and written into the corresponding queue; the server keeps a queue for each connected client as part of the `ClientRec` structure that contains all state information about a particular client. If multiple clients have requests pending, the scheduler then decides which of them to serve next, usually this selection is performed in a simple round-robin fashion.

The next request is then pulled from the queue and fed into the main dispatcher function, which will extract the (major) opcode from the request packet and use it as an index into the “`requestVector`”; the `requestVector` is a table the elements of which point to functions that will perform the action corresponding to the opcode, e.g. the 65th element points to the `ProcPolyLine` function which will interpret the request data as a sequence of points to be connected by straight lines.

The first action performed by the function found in the `requestVector` is to perform byte-swapping of the values contained in the request structure if required, and then translate the passed `XID` values into the corresponding resource objects; e.g. the `ProcPolyLine` function would have to lookup the `Drawable` and graphics context objects referenced by the client.

Finally the dispatcher function will usually call into the `ddx` layer to perform drawing, allocate resources requested by the client, or generate reply packets to transfer requested data back to the client.

3.6 X extensions

Since the core X11 protocol only reserves the first 127 (out of the possible 255) request opcodes and just a small amount of the available values for event types and error codes, the remaining can be used for extensions. Usually an extension defines its own set of requests, event types and error codes. Instead of reserving fixed numbers for each extension with the associated disadvantages of static assignment, a flexible mechanism to dynamically map protocol opcodes, error codes and event types to available extensions is used.

BIG-REQUESTS	allows protocol requests larger than 2^{18} octets
DOUBLE-BUFFER	enables windows to have a back buffer
DPMS	power management features
GLX	OpenGL embedded in the X protocol stream
MIT-SHM	allows sharing memory segments with the server
RENDER	Porter-Duff image compositing; see [14]
SECURITY	manipulating access control to the X server
SHAPE	allow windows to have non-rectangular shapes
XIE	support for various image formats; see [13]
XInputExtension	support for various input devices
XVideo	support for video overlays

Table 3.1: Selection of common X extensions

Unlike core protocol elements, which are solely identified by static numbers, extensions are primarily identified through a unique character string (cf. table 3.1). Each extension is assigned a request opcode (generally the extension wishes to further multiplex this opcode into sub requests, hence this opcode is referred to as the *major* opcode, while the codes identifying the sub requests are referred to as the *minor* opcodes), a range of values for event types and a range of values for error codes.

The client can get a complete listing of the available extensions through the `ListExtensions` request, and obtain the major opcode, event type and error code values assigned by the server to a particular extension through the `QueryExtension` request. Commonly this is implicitly performed by the client library upon connecting to the server, largely hiding to the programmer the fact that some functions translate into core protocol requests, while others rely on extensions.

Some extensions are considered standard and are supported by virtually any X server (e.g. `MIT-SHM`), and in most client applications have become dependent on these extensions being present. In fact by now the amount of code dedicated to extensions by far exceeds the code for the core X server.

3.7 XFree 86 extension modules

Traditionally, the X server has been a monolithic program custom-tailored to the intended hardware platform. The X server produced by the XFree86 project⁵ since version 4.0 features a very modular architecture. It provides the ability to load modules at server start up time, which extend the server functionality by providing drivers for graphic adapters or other hardware, or by implementing protocol extensions.

3.7.1 Module loader

Although most operating systems provide facilities for dynamically loading libraries at run-time, the XFree86 server does not make use of this, but instead provides its own loader mechanism based on the ELF object format. The rationale behind this is to provide modules which are independent from the underlying operating system.

Part of this operating system abstraction is also the symbol resolution: Only symbols that are explicitly exported from the server core and symbols provided by already loaded modules are available to other modules. Thus not all symbols normally provided by the operating system environment are available to modules, and not all symbols internally used by the X server.

Extension modules come in the form of ELF objects; for most Unix systems this is the default object format, so no special steps are necessary to turn the object files generated by the C compiler and linker into a format suitable for the XFree86 loader.

After loading a module first the symbol references are resolved. Then the loader looks up a specially named symbol, which is the name of the object file, followed by `ModuleData`. This symbol must point to a structure describing this module; it contains among meta-information about the module (type, version information, vendor information) a function pointer to the setup routine for the module. Control is passed on to this setup function by the loader, and all further actions in initializing this module are performed by this function. It may, depending on the purpose of the module, provide and register hardware drivers, register X server extensions, or otherwise hook into the functionality of the X server.

⁵see [17]

3.7.2 Protocol extensions

Owing to the large number of extensions that have been written for the X Window System, the X server provides a solid API for extension modules. The individual steps that have to be performed in the module initialization function to register a new protocol extension are discussed below.

Extension registration

Registration is performed through the `AddExtension` API function provided by the server; it should be called during the module initialization and needs to be provided with the following information: the name of the module (used by the client to find the extension, cf. section 3.6); the number of error and event codes to reserve (to deliver extension-specific error and event messages to the client, cf. section 3.3); the dispatcher function⁶ that will process client requests directed at this extension (the dispatcher function will be invoked whenever the server receives a protocol request carrying the major request opcode assigned to this extension, cf. sections 3.3 and 3.5).

Only one major opcode will be reserved for an extensions, therefore the registered dispatcher function will generally perform multiplexing and further differentiate requests by the minor opcode field.

For each event defined by the extension, a function to byte-swap the contents of an event message needs to be provided. This is required to be able to send events to clients which have a different native byte order.

Resources

Extensions that provide new server-side objects which clients need to interact with, should define new resource types for these objects to avoid duplicating functionality already provided by the server core, and to provide a uniform interface to client applications. For this purpose, the `CreateNewResourceType` function can be used by extension modules. The module has to provide a clean-up function for each resource type, so that automatic deallocation of resources on client disconnect can work.

⁶to be precise, two dispatcher functions need to be provided; one that receives requests from clients which have the same byte order as the server, and one that receives requests from clients with different byte order and that has to byte-swap the values first

The value returned by the `CreateNewResourceType` can be used as an argument to the `SecurityLookup...` class of functions mentioned in section 3.5; this way, the X server can guarantee type-safety for all allocated resources.

4 Requirements and Architecture of a networked multimedia system

4.1 Requirements

This section will detail the requirements and issues to take into consideration for a networked multimedia system. As the prototype discussed later will be based on the X window system, some remarks specific to this target environment will also be made, though most of the arguments apply equally to other networked graphics systems.

Taking over the terminology of the X window system, *server* will in the following always refer to the display system, while *client* will refer to the application using the display to provide users with a graphical presentation.

4.1.1 Network transparency

The introduction of network transparency into the system has fundamental architectural consequences. In addition to the general issues involved in communicating multimedia content across (best-effort) networks¹, matters are further complicated as this work is aimed at providing a fairly fine-granular semantic. As the QuickTime model is heavily based on callback functions, it cannot be directly mapped onto a networked environment. Decisions have to be made, which part of the tasks have to be run on which end of the network: at the client applications side, or at the side of the graphics display.

The following paragraphs will detail the specific problems in this context and motivate the distribution of tasks.

¹see [10] for a fairly comprehensive overview

Communication mechanism	Measured round-trip time
Local procedure call (function pointer)	0.008 μ s
System call (<code>getpid</code>)	0.212 μ s
Remote procedure call (X protocol), client and server on same machine	67.366 μ s
Remote procedure call (X protocol), local area network	152.521 μ s
Remote procedure call (X protocol), across campus network	17185.557 μ s

The measured system (local tests and X Server) is an AMD Athlon, 1.3GHz, Linux 2.4.18, XFree86 4.1; the X client for the last two tests was hosted on an Intel Pentium III, 900MHz system running the same software configuration, although it turned out that neither CPU speed nor software configuration had any significant impact on the timings

Table 4.1: Communication latencies

Communication latency

An important issue to take into consideration for all kinds of distributed systems is the communication latency involved in the RPC mechanisms. Table 4.1 gives a rough overview of the latencies involved using various communication channels.

As the table shows the latency is non-negligible for LAN connections already and poses a serious problem for long-distance connections, or when the network is experiencing heavy contention by multiple users.

Thus a synchronous programming model, where each request requires an immediate response before the application can proceed, would result in very poor utilization of the available resources as both communication partners tend to spend significantly more time waiting than processing.

For networked operation an asynchronous programming model is therefore mandatory, but it turns out that asynchronous operation generally also greatly benefits the non-networked case because requests can be batched and the overhead for inter-process communication (context switching, adverse effects on the processor cache) can be mitigated over a larger number of requests. As shall be seen later, asynchronicity was an important design principle for the X

window system.

It should however be noted that asynchronicity introduces a significant amount of complexity into the application, and dealing with the arising problems of deferred error reporting and cascaded errors is difficult at best².

Jitter

Variances in the delay between sending a message across a network and receiving it is generally referred to as *jitter* or *delay jitter*; this means that if two messages are sent with a known delay, the difference between the arrival times of the messages is still rather unpredictable.

Naturally this conflicts with the desired precise timing of presentation elements, and it follows that the timing cannot be driven by the client, but instead the server has to be able to autonomously provide *timing* for the presentation; another consequence is that the server needs to provide *buffering* for data representing the presentation elements.

Related to this is the issue of synchronizing different presentation streams, most notably video and audio, and although audio is outside the scope of this work, the possibility of including audio support into the architecture should be left open. Using client-side synchronization would make it difficult to achieve synchronization quality acceptable to the human perception system³, unless operated in highly-controlled network environments.

Architecture neutrality

Computer are often very heterogenous, consisting of machines with very different architectures, in both hardware and software. Linking these together in a distributed system puts a number of portability requirements on the system components as well as protocol conventions.

Portability includes the ability to run a software in different hardware and operating system environments. The X window system provides a good example and a good basis for development in this respect as much effort has been put into portability, and in fact it is the only graphics system that is available on virtually any platform.

The network protocol used for communication between the components has to account for the fact that different architectures use different native representations for numerals: least

²The default action of X client applications is to terminate on receiving an `X_Error` from the X server

³cf. [5] for bounds on the perceptability of synchronization errors

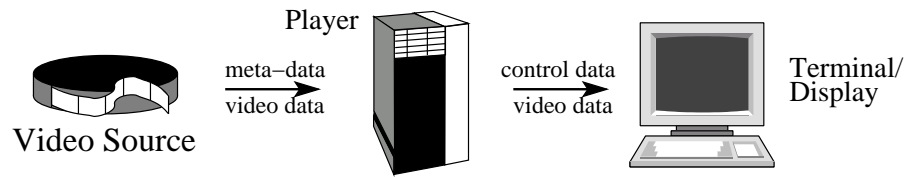


Figure 4.1: Client application supplying video data

significant byte first (LSB, also known as *little endian*), or most significant byte first (MSB, also known as *big endian*). The convention used in most network protocols is to store numerals as MSB, which is therefore also called *network byte order*. The X protocol has a different convention, explained in section 3.3, which will be used for consistency reasons, as the prototype implementation is based on the X window system.

Data transport

The protocols underlying most networked graphics systems do not only supply graphics, but also convey user actions such as key presses. Moreover the state of the graphics system often depends on the correct execution order of graphics operations, and for this reason, a reliable, order-preserving data transport mechanism is used underneath; in the case of the X protocol this is usually TCP.

The characteristics of bulk real-time data such as video is however very different and make reliable streams such as TCP a less than well-suited transport protocol: for one, video data is comparatively resilient against network errors such as packet loss or reordering⁴; moreover TCPs strategy for recovering from network errors does not play too well with the combination of large data rates and real-time requirements typical for streaming media.

Therefore there is potential benefit in separating the data flows and allow differentiation at the network level to provide service guarantees most appropriate for the different characteristics of flows.

There are other reasons why one would wish a separate transport path for the video data, e.g. in the case when a third party different from both client and display server is acting as video source (see figure 4.2), as otherwise all data would have to pass through the client (as in figure 4.1); the latter is not only less efficient, it also introduces additional latency that may be undesirable for certain applications.

⁴for this purpose, special encapsulations have been devised; e.g. cf. [8] for MPEG payload data

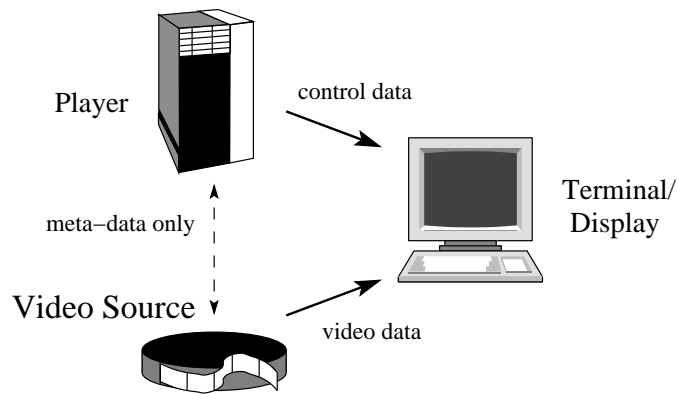


Figure 4.2: Video source separate from client application

4.1.2 Real-time constraints

Multimedia is closely related to real-time processing, but unlike typical hard real-time applications failure to meet a given deadline is not fatal but only lowers the quality of the users visual experience. Effort should be put into honoring the given timing constraints, but the failure case cannot be excluded and must be prepared for.

Timing constraints in video applications typically specify an interval during which a picture should be visible to the user. Due to the nature of the human perception system a small amount of fuzziness does not significantly reduce visual quality, but systematic skew where one picture that is displayed too late will delay all subsequent frames (possibly even cumulative effects) has to be avoided.

The most common solution in multimedia is to accept the possibility of missed deadlines and to simply drop frames when they cannot be displayed in time, and proceed with the next one to catch up with the intended picture rate. However care must be taken if composition of pictures depends on previous pictures.

4.1.3 Abstraction level

One of the most difficult aspects in designing an interface is the level of abstraction provided. While all of the above requirements could be met by providing a “black-box” API – that is, video streams are considered opaque objects for which the client and the graphics system merely set up a screen area for displaying the video, but the presentation would run completely

autonomously – this approach has a number of fundamental weaknesses. Basically no client interaction besides starting and stopping the presentation is possible, and some of the more interesting applications that combine video images with other graphics primitives are impossible to implement.

It is therefore a stated goal to provide a semantic that allows influencing the presentation at a very low-level, similar in spirit to QuickTime, where clients can take over complete control of the presentation at every stage of the processing. The key point of this work will be to provide a model that can unite these two seemingly contradictory requirements and provide for both, largely autonomous processing independent from the client to allow a smooth presentation, and still retain full client control at every step.

4.1.4 Extensibility

During discussion of the QuickTime architecture a number of compression formats for still images and video frames have already been mentioned. Although most of the formats in use today are based on a small common set of compression techniques, the details of the implementations differ widely, resulting in a large number of different and incompatible formats for compressed video images.

Conversion between the different compression formats is generally very compute-intensive and time-consuming, as stated in the preface it is therefore highly desirable to avoid reencoding the data. This in turn requires that the display server has to provide support for a wide range of compression formats, and to simplify the process of adding new formats the architecture should be modular in a way so that it can be extended in this fashion easily.

4.1.5 Security considerations

Though development of a security framework is outside the scope of this work, the issues introduced by networked multimedia should at least be mentioned. Although the following discussion is focussed on the X window system, most of it applies to any networked graphics system.

The X server is sensitive to security in two ways: 1. It is the users sole way of communicating with his applications, intercepting this communication would enable anyone to impersonate him and gain all of his privileges. 2. The X server usually has elevated privileges on the system it

is running on, mainly for efficiency reasons to allow direct communication with the underlying hardware devices.

The first problem is solved by authenticating access to the X server itself and providing confidentiality for the communication between X server and X clients e.g. through encryption at the transport layer. This is obviously unaffected by modifications at the protocol level, but data transport paths outside of the X protocol cannot immediately benefit from this protection, so the mechanisms need to be extended to cover these transport paths as well. It should be noted that separate transport mechanisms also offer potential advantages in that different confidentiality policies and resulting encryption strengths can be applied to the separate streams, allowing to save computational effort by using weaker ciphers for less sensitive data.

The second problem is far more difficult to address. The key issue is the requirement for low overhead to save precious computing resource for actual processing and to present to the user a system reacting with almost no perceptible delay. As with every complex piece of software the risk of introducing an exploitable hole into a sensitive system through a programming error increases with every line of code; video codecs are prone to this simply due their complexity and need to optimize heavily, and frequently are not provided with source code available, making security auditing impossible. One possible solution would be to perform the decoding step in a specially protected environment to minimize the impact of programming errors; this however incurs a serious performance penalty, so care must be taken balancing these contradictory requirements.

4.2 Architecture

Taking the requirements outlined in the previous section into consideration, an architecture for a networked multimedia system can now be formulated.

The main design challenge is to unite the contradictory requirements of full network transparency, detailed client control and smooth presentation. The preceding paragraphs show that this cannot be achieved trivially. As was pointed out in section 4.1.1, time-critical decisions have to be made by the display server autonomously. For example during a video presentation the timing when an individual picture has to be displayed on the screen needs to be independent of the network to ensure highest possible accuracy.

The approach taken is to provide two separate kinds of services inside the display server:

1. (Video) image decompression services
2. Presentation timing services

The image decompression services will extend the semantics of the graphics systems by providing a new primitive operation, namely generating pictures from a compressed video stream. Timing services will allow arbitrary drawing operations to be scheduled for execution at a later point in time.

This split-up resembles the distinct services provided by clock and image decompressor components in the QuickTime architecture (see second chapter), though there are a number of differences especially with respect to timing.

4.2.1 Image decompression

Purpose

The concept of image decompressor components can be transferred from the QuickTime architecture without too many changes. It is desirable to have a similar plugin-architecture for different components dealing with different compression formats. As in QuickTime the main service provided by each of these components is to decode images from a compressed representation into a representation suitable for display or further processing.

However a very different approach needs to be taken towards passing the compressed image data; while QuickTime combines data transport and processing into one step (by passing the image data as procedural argument), section 4.1.1 shows that a strict separation of these concepts is more appropriate for a networked system.

Separation of transport and processing also mandates that decoding dependencies of pictures are treated differently. In QuickTime they are implicit and they have to be taken care of by the application using the decompression services (i.e. the dependency is implicitly determined by the order in which pictures are decompressed, and the application has to take care of decompressing required pictures in the correct order itself). Instead these dependencies need to be made explicit, and this information must be conveyed at the same time as the actual image data is transmitted in order to be able to support application scenarios as shown in figure 4.2 on page 35, as it is unfeasible for the player application to obtain knowledge of the decoding dependencies in this case.

Semantics

The decompression services will be provided by server side “video stream objects”. The semantics provided by these objects is as follows.

- (1) **Frame data.** The video stream data is transferred to the server as a sequence of individual frames. The raw compressed data for each frame is transmitted separately, decoding dependencies of each frame are communicated to the server at the same time as the data is submitted
- (2) **Sequence of pictures.** The pictures are arranged in a sequence by associating a sequence number with each picture at the time its data is transferred to the display server
- (3) **Picture access.** The client application may request an individual picture, identified by its sequence number, to be decoded into a visible area of the screen or an off-screen area for further processing
- (4) **Strictly monotonic access.** Pictures may only be accessed sequentially by the client, i.e. no picture will be accessed twice and a picture may not be accessed after a subsequent picture (i.e. picture with a higher sequence number) has been accessed
- (5) **Picture skipping.** Not all pictures have to be accessed by the client, pictures may be skipped
- (6) **Decoding dependencies.** The decompressor takes care of decoding all required pictures in the correct order for each picture requested by the client

First it should be noted that the timing information usually associated with the pictures is dropped completely. This is intentional to provide the client with the flexibility to display the same stream with different timings, such as slow motion, fast forwarding, or even picture by picture. The playback timing information will be provided through the timing mechanism discussed later.

The order in which pictures can be retrieved from the stream is defined by the sequence numbers, however the interface does neither expose nor define which “logical” order the pictures are in; this is completely at the discretion of the supplier of the data; so the client application wishing to display pictures out of a sequence needs to be aware of the ordering through other

means – either if the X client itself is supplying the data representing the stream, then it will know the ordering as it has to send this information along to the server anyway; on the other hand, if an external source is providing the data, player and data source will usually have to agree on numbering the pictures in display order. For all purposes of client access, the video stream acts as an opaque “picture generator” and is ignorant of the logical ordering associated with the pictures.

The video stream decoder has to be prepared to decode pictures in a different order as it has to emit them⁵. This behavior is required to allow the X client to display pictures from a video source to the data of which it does not have any access, and thus cannot know the correct decoding sequence.

On the other hand if the X client itself is submitting the video data to the decoder, it may use the available information about the decoding dependencies; it may wish to have the decoder emit the pictures in decoding order and use this property to optimize the decoding process (an example how this can be achieved is given in section 6.2).

Restriction (4) is required to enable the decoder to operate with bounded amount of memory, as otherwise it would not have a clear way to dispose of data that will never be used again. About the sequence numbers it should be noted that they are not required to be consecutive, i.e. the numbering may have “gaps” – from the perspective of the server-side extension any numbering will do, as far as it is consistent with (4). The semantic is intentionally liberal in this respect to allow using a numbering that is directly derived from stream meta-information such as presentation timestamps.

Lastly there is a need for the ability to skip pictures (5) – for one the X client may not want to display all pictures, e.g. during fast forward; for another it is required in case the deadline for displaying a picture has been missed, e.g. because the decoder can not keep up with the intended frame rate.

4.2.2 Timing

Purpose

The services provided by QuickTime clock components cannot be transferred directly, as the QuickTime model is heavily based on callback functions, however the conceptual modularity of

⁵Within reasonable limits; if arbitrary orderings are allowed, it is possible to construct sequences that require an unbounded amount of storage to keep intermediate results that are needed to decode subsequent pictures.

QuickTime to allow using different sources of time information should be preserved to provide synchronization capabilities similar to QuickTime.

The basic idea is to submit drawing operations, but have them not executed immediately but at a later, pre-defined point in time. As was pointed out in section 4.1.2 however a strong guarantee cannot (and for multimedia need not) be made. Instead the notion of a “validity interval” is introduced, during which the drawing must be performed, allowing the specific set of drawing operations to be silently skipped if the deadline cannot be met. This allows to implement the common approach taken in video playback applications, to drop individual frames if the decoder cannot keep up with the desired frame rate.

Thus the programming model for a timed presentation is more abstract than the model provided by QuickTime: In QuickTime, the decision *which* operations to perform at a specific point in time can be postponed to when this point has been reached; in the networked case at hand however, this decision has to be made beforehand, requiring a programming model that does considerably more planning.

Semantics

The semantic of the server-side drawing scheduler will be as follows:

- (1) **Execution interval.** Operations are submitted to the scheduler indicating a time interval during which they should be performed; the operations *should* be performed at the beginning of the interval and *may* be performed at any point within the interval; however the operations will not be performed if the end point of the interval has been passed
- (2) **Transactional semantic.** Groups of related operations may be submitted together; the server guarantees transactional semantics for this group – either all of the operations are executed, or none is
- (3) **Dependencies.** The client application may express that a group of operations depends on successful execution of a previous group of operations; in that case the display server will not execute the dependent group of operations if the referenced group was not executed, either because the timing constraints could not be fulfilled or because a dependency was not met

- (4) **Execution order.** Operations timed at an interval beginning at a later point in time will not be executed before operations timed at an interval beginning at an earlier point in time
- (5) **Clock source.** Time information for a scheduler is provided through a selectable clock source; different clock sources are not guaranteed to be synchronized with respect to each other

The main idea of submitting drawing operations together with a validity interval is covered in (1) and (2). Providing transactional semantic is necessary in practice because some partially completed operations could result in resources being left in an undefined state.

It should be noted that (1) poses a hard real-time problem, and in addition it requires that the time required to complete a given operation can be calculated in advance: as stated, it demands that all operations are *completed* within the designated interval. While this behavior is certainly desirable, it is difficult to achieve in practice (especially since none of the underlying system components were specifically designed for hard real-time operation), so that for the purposes of the prototype implementation (1) will be replaced by:

- (1') **Execution interval.** Operations are submitted to the scheduler indicating a time interval during which they should be performed; execution of the operations *should* be started at the beginning of the interval and they *may* be started at any point within the interval; however the operations will not be performed if the end point of the interval has been passed before their execution could begin

This relaxation of the timing constraints can be justified under the (reasonable) assumption that the time required to complete a graphics operations is relatively short.

Applications may in some cases wish to use previously rendered frames, or parts thereof, to construct subsequent frames. It is not generally possible to express this in terms of validity intervals, hence the need to make this dependency explicit to the X server, as outlined in (3). Requirement (4) is necessary to remove ambiguities in case multiple operations with overlapping validity intervals are submitted. As shall be seen later, both mechanism can be used to intelligently control frame skipping for video stream formats with predicted frames.

The ability to choose the source of time information for a particular scheduler borrows from the concept of QuickTime 'clock' components. Even though audio support is not covered, this

hook is included here to demonstrate how introducing audio affects the architecture, and what interfaces are required from audio streams.

5 Prototype implementation

The reference implementation provides an extension of the X window system by streaming media services; it consists of three X server extension modules, which are accompanied by client-side libraries that provide access to the extension mechanisms. The implementation consists of roughly 12000 lines of C code, including a few sample programs.

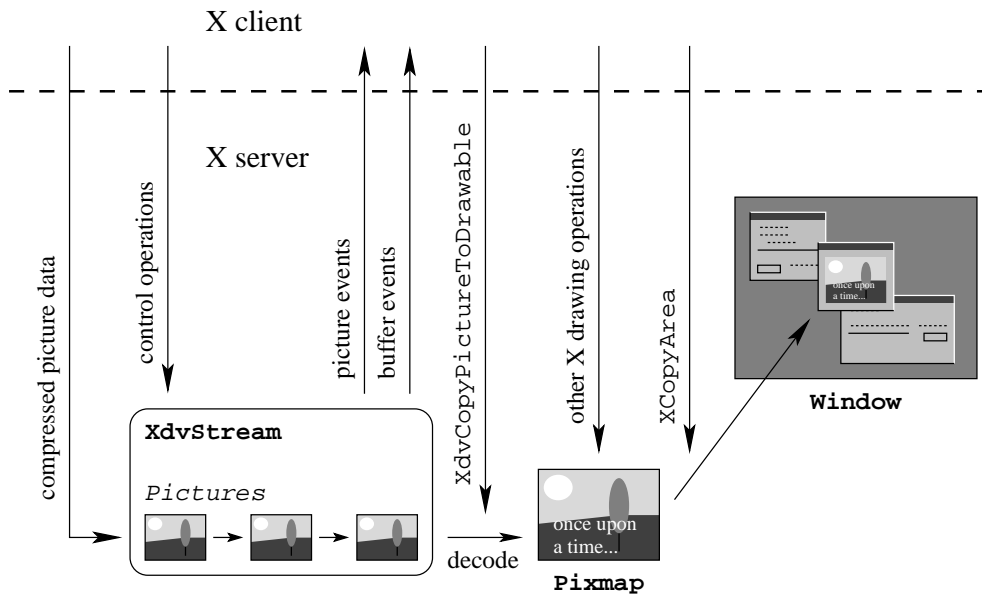
The first extension module, `Xdv`, provides a framework for video decompression services; in a rough sense, it is the equivalent of the QuickTime image decompression manager discussed in chapter 2.2. The second module, `codec_mpeg`, provides an MPEG1 decompressor module for the `Xdv` framework; it is provided as a separate module to demonstrate the extensibility and modularity of the prototype implementation. These modules will be covered in sections 5.1 and 5.2, respectively.

The last extension module, `Xtime`, provides timing services; it is roughly equivalent to the QuickTime clock component type described in chapter 2.3. It will be covered in section 5.3.

The extensions were written as loadable modules for the XFree86 server, as described in section 3.7; they were developed on a Linux-x86 system for version 4.2 of the X server, but are known to also work with version 4.1 and are likely to work with other versions as well. Due to the nature of the XFree86 loader design, it is to be expected that they also work with other operating systems on the same hardware platform.

The reasons for the decision to implement the functionality without touching the core code of the X server are twofold: First, the extension modules can be plugged into any x86-based Unix system running the XFree86 server, allowing anyone to easily test and validate the implementation. Second, development was slightly easier on self-contained modules, as it obviated the need for time-consuming recompiles of the whole server.

The requirements outlined in section 4.1 were considered throughout the whole design and



Intended usage of the Xdv extension and interaction with the core X protocol. Pictures obtained from a video stream can be decoded into a Drawable, which may either be a window (where the picture is immediately visible to the user) or an off-screen pixmap where the client may apply further operations to the image before display (a subtitle is rendered over the image in the example shown).

Figure 5.1: Client view of the Xdv extension

implementation process. The purpose of the implementation is to illustrate these requirements as well as the usefulness of the architecture presented in section 4.2.

5.1 Design and implementation of the Xdv extension

This section will describe the video image decompression component of the streaming media extensions, Xdv.

5.1.1 Goals and design requirements

This extension provides facilities for decoding a compressed video stream. It receives data from a video data source (e.g. the client application itself), and decodes client-selected pictures into a client-specified target (cf. figure 5.1).

The following requirements were considered in addition to those formulated in section 4.1:

1. Make use of specialized hardware
2. Support for multiple codecs

Architecturally the Xdv extension as presented here is prepared to meet all of the above design requirements; however not all of these could be implemented due to time constraints:

1. Modern graphics cards can display overlay images in non-RGB color spaces (i.e. $Y C_R C_B$), avoiding the need to perform color space conversion; the implementation is easily extended to support non-RGB output formats, but this capability is not included here due to lacking hardware to properly this feature.
2. The implementation is kept modular to allow the use of multiple video codecs, but only one codec (MPEG-1) has actually been implemented so far

It should also be noted that Data transport and decoding are kept separate throughout the whole implementation; although no other transport path has been implemented than to embed the compressed video data into the X protocol stream, other transport paths (like RTP¹ or shared memory) are architecturally easy to integrate

5.1.2 Implementation

Overview

The Xdv extension represents video streams as X resources as described in chapter 3.2. This new resource type, `XdvStream`, provides the semantics outlined in section 4.2.1.

Individual pictures are not directly represented through X resources, they can only be accessed through the corresponding stream resource. They are identified through a 32-bit integer which will often be referred to as `XdvPictureID`; these numbers are the sequence numbers discussed in 4.2.1.

The server-side part of the implementation does make only one assumption about these numbers: Monotonically increasing numbers (modulo 2^{32}) denote the intended access pattern

¹cf. RFC1889 and [6]

and enable the server to discard all pictures “before” the one accessed last. (More precisely the numbers should be thought of as forming a ring of 2^{32} elements, and the relations “before” and “after” are defined in terms of the direction of the shortest connecting path along the ring). Specifically the server does not make any assumption about the IDs being contiguous, or taking particular values at all.

The `XdvPictureIDs` can directly be derived from stream-specific meta-information about the individual pictures, like presentation or decoding timestamps (extended or clamped to 32-bit integers), or simply consecutive cardinals, as long as they represent the intended access sequence.

Alongside with each `XdvPicture` goes the meta-information about picture dependencies for non-key-frames; for this purpose, up to two referenced pictures can be indicated for each picture. They indicate to the scheduler in which order pictures have to be decoded. Particular decoder modules can also use these references and interpret the referenced pictures in ways meaningful to the particular encoding format (e.g. forward reference, backward reference pictures in MPEG-1 video streams).

Client extension

The client extension consists of a header file (`Xdvlib.h`), containing prototypes and definitions, and corresponding library (`libXdv.a`) which must be linked into programs wishing to make use of this extension. The functions available through this library are explained below.

Server extension

The server-side extension consists of two layers:

- codec-independent layer (from now on called “Xdv layer”)
- codec-dependent layer (from now on called “codec layer”)

The purpose of the Xdv layer is to provide a framework usable by many different video codecs; its tasks include:

- client request dispatching

- buffer management
- resolution of picture dependencies and decoder scheduling
- management of multiple codecs

The general design concept has been to push as much functionality as possible into the codec-independent layer, to increase code reuse for the various codecs. The tasks of the codec layer are:

- decode pictures
- transform pictures into requested format
- transfer picture to requested destination

It may appear that copying the decoded pictures into the destination and conversion into the target format belong into the codec-independent layer, and in fact a number of convenience functions usable by codecs are provided there. However a specific codec may want to off-load the decoding into dedicated hardware and at the same time use specialized conversion and blitting functions of the hardware; to give codec implementations the flexibility to do so, the decision which mechanism to use is left for the lower layers.

The Xdv layer is implemented as a single loadable X server module `libxdv.so`. Additional loadable modules can provide specific codecs, and can be loaded independently from each other.

Protocol extension

The client and server extensions are accompanied by an according extension to the X protocol by additional requests and replies; since their structure is mostly evident from the corresponding API functions, they are not discussed here, as it would provide only very little interesting information.

5.1.3 Xdv client API

This section will describe how the client API functions supplied by the Xdv extension should be used. Detailed information on and declaration of the functions mentioned here can be found in Appendix A

Extension initialization

Most X extension modules export a function that enables the client to query for existence and capabilities of a particular extension, and Xdv adheres to this convention. To make sure that the Xdv extension is supported by the X server in question, every application should call `XdvQueryExtension`.

Creating and freeing video stream resources

`XdvStream` resources are represented in the client simply through their resource id. They are allocated and destroyed through the `XdvCreateStream` and `XdvFreeStream` functions and act as objects through which all of the Xdv functionality is made available.

Stream properties

Stream instances support reading and writing a number of named properties. Properties are generally specific to every codec, though some properties common to all types of codecs exist, including `XDV_WIDTH` and `XDV_HEIGHT` describing the width and height in pixels of the images generated by the stream.

Properties are used to set control and meta-variables of a stream; this includes setting options that modify behavior of the codec (like `XDV_INTRA_QUANTIZER_MATRIX` for the MPEG-decoder supplied as part of this implementation), parameters describing the supplied data stream (like the aforementioned width and height properties), or influence buffering parameters (through `XDV_BUFFER_MAX` and `XDV_PICTURES_MAX`).

Stream properties are represented very much like window properties in the core X11 protocol and server². They are accessed through atoms, and their values are arrays of 8-, 16- or 32-bit integers. Unlike window properties however the size of the array and type of elements is fixed through the meaning of the property; for example `XDV_WIDTH` is represented through a 1-element array of 16-bit values, i.e. a single 16-bit value; on the other hand `XDV_INTRAQUANTIZER_MATRIX` is represented through a 64-element array of 8-bit integers.

The functions used by clients to set and get stream properties are `XdvSetStreamProperty` and `XdvGetStreamProperty`.

²see [12] for the mechanism commonly used in X

Event notification

Stream objects can notify the client of various status changes and events. The two events used by this extension are `XdvThresholdNotify` and `XdvPictureNotify`.

`XdvThresholdNotify` is used to inform the client about the buffering status of a particular video stream. Two buffering quantities may be monitored through these events: Total number of bytes allocated for the compressed video data, and total number of pictures that have not yet been decoded. Clients can request notification whenever one of these quantities passes defined threshold values, or when data is lost due to buffer overruns. The intention is to allow clients “soft” synchronization with the video stream to help it prevent buffer under- or overruns by using low and high water-mark thresholds.

`XdvPictureNotify` is used to deliver notification about individual pictures; conditions under which this event is raised include allocation of new pictures, or pictures that could not be decoded because no data was available. The intention is to provide feedback to the client when video data is supplied through a different source; it is also intended to allow the client an assessment of the quality of the video presented if data is transmitted through a lossy channel.

Extracting pictures from the stream

The API is designed to allow streams to generate multiple output formats. Video is usually not encoded in the RGB color space, and it is sometimes desirable to keep the decoded data in its native color space. The current implementation does not provide functions for generating output in non-RGB color spaces (e.g. YUV) and appropriate X resources (e.g. `XvImage`); this part of the implementation was omitted due to lacking hardware and software to properly test this capability.

Before pictures are generated, the output format has to be fixed, as the graphics hardware may be able to simultaneously display images of different depths and color formats³. This allows the implementation to take necessary preparatory steps to optimize the decoding for the requested format.

Clients can set up a stream resource to generate pictures as X Drawables using the `XdvSetOutputDrawableRGB` function, and extract individual pictures using the

³the implementation currently only has support for TrueColor and DirectColor visuals (these visuals directly encode the red, green and blue components in each pixel value); PseudoColor (pixel values are interpreted as indices into a color palette) is unsupported

`XdvCopyPictureToDrawable` function.

Embedding frame data in the X protocol stream

While the transport of frame data has been completely separated from the decoding process, the only transport path available at the moment is by embedding the data into the X protocol.

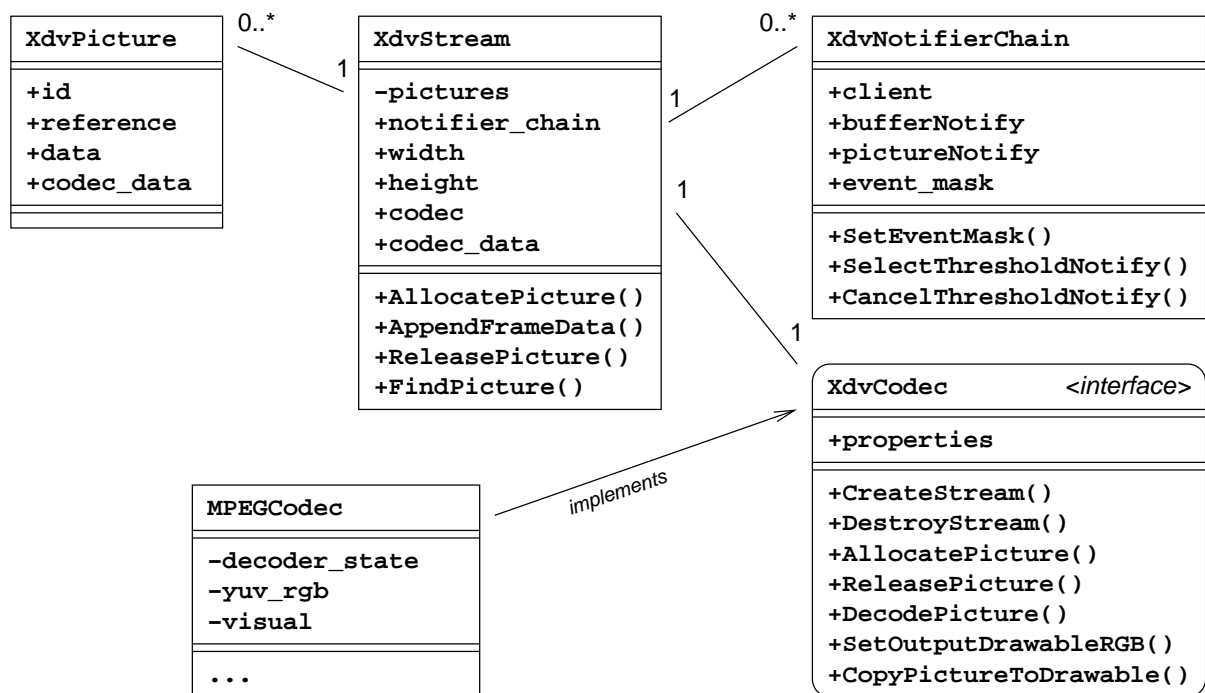
Supplying frame data through the X protocol requires two functions: `XdvCreateFrame`, which allocates a new frame and supplies meta-information about the frame (referenced frames); and `XdvFrameData` which transmits the raw data for a previously allocated frame.

The main data structures this extension is concerned with are `XdvStream` and `XdvCodec`. The interrelationship of all important data structures defined in the extension is shown in diagram 5.2. A short description of the presented data structures follows, further information can be found in the corresponding C header files of the sample implementation.

5.1.4 Server-side implementation

Data structures

- `XdvStream` represents a video stream resource as it is visible from the client application; it contains all information relating to the stream and its presentation, most notably stream meta-data, decoder state information and buffer management information.
- `XdvCodec` describes a particular video codec; among other things it acts as a virtual method table for the `XdvStream` instances if it looked at from an object-oriented point of view. It contains pointers to all codec-specific operations that can be performed on an `XdvStream` (like the actual decoder function) as well as methods for setting/getting codec-specific properties of the stream.
- `XdvPicture` represents a single picture of a video stream; it contains all data and meta-data required to reconstruct this picture, including pointers to forward- and backward-reference frames.
- `XdvNotifierChain` contains the notification state of a particular client. It keeps track of which events the client is interested in; its existence as a stand-alone data structure is more of a technical origin, to allow automatic deallocation of notifier chains if the corresponding client closes its connection to the X server.



The central data structure in the Xdv extension is `XdvStream`, representing a video stream.

Figure 5.2: UML diagram of the main Xdv data structures

All of the data structures except for `XdvCodec` are managed by the Xdv layer, keeping track of their allocation and deallocation. `XdvCodec` is provided and filled in solely by the particular codec implementation.

The `XdvStream` and `XdvPicture` structures both contain a pointer `codec_data` which the codec implementation may use for its own purposes to attach codec-specific data to picture and stream objects. Uses for this include decoder state information, codec-specific properties, pointers to copies of decoded pictures (for reference frames); all of these purposes are demonstrated in the sample MPEG codec implementation.

The Xdv layer does not interpret the `codec_data` pointer in any ways; it is therefore the responsibility of the codec implementation to allocate or deallocate data structures referenced by `codec_data`.

Interface to decoder modules

Every codec module provides a number of API functions through function pointers in the `XdvCodec` structure; these functions are invoked when the Xdv layer needs the decoder to perform operations such as decoding a picture from the compressed video stream. Codecs register themselves with the Xdv framework by calling the `XdvAddVideoCodec` function.

The functions a codec module must provide are:

- **CreateStream, DestroyStream**: called from the Xdv layer when the stream is created, or destroyed, respectively
- **AllocatePicture**: called from the Xdv layer before a picture is to be decompressed; it advises the decoder to allocate temporary space to decode and hold the indicated picture
- **ReleasePicture**: called when a picture is no longer needed, e.g. its last referencing picture has been decoded; the decoder should free the previously allocated buffer space for this picture and reuse it for subsequent pictures
- **DecodePicture**: called when a picture must be decoded, but not yet displayed; this is the case this picture is used as a backward reference frame for other pictures
- **SetOutputDrawableRGB**: instructs the decoder to take preparatory steps to generate pictures in the indicated output format; this may include setting up conversion routines to convert the picture to the visual format indicated to this function (Xdv also includes a number of helper functions for color space conversion).
- **CopyPictureToDrawable**: instructs the decoder to display the indicated picture, decoding it along the way if it has not been decoded previously; the Xdv layer does *not* necessarily previously call **DecodePicture**, as the decoder may want to provide a combined decode-and-display operation that is more efficient than performing each step separately

5.2 MPEG 1 codec

The implementation of the Xdv extension is complemented with a sample MPEG 1 codec module. The purpose of this module is to show how the infrastructure provided by Xdv can be used to plug in decoder modules for different video formats, and to verify the viability of the whole approach by providing a system which is fully functional and can be demonstrated.

5.2.1 Structure of MPEG-1 video

This section will take a cursory glance at the structure and ideas behind MPEG 1 video; necessarily this presentation is incomplete and only covers what is necessary to understand the specifics of the codec implementation. For a complete reference see [4].

The images forming an MPEG-1 video stream are usually referred to as *frames* in this context; each of these frames is encoded in one of the following three possible ways:

- **I frames** (intra frames) are coded independently and can be decoded out of the context of the video stream; they act as “synchronization points” in the stream
- **P frames** (predicted frames) are coded relative to the temporally preceding I or P frame; they make use of the temporal redundancy in a video sequence by coding only the differences to the referenced picture
- **B frames** (bi-directionally predicted frames) are coded relative to the temporally preceding I or P frame and relative to the temporally succeeding I or P frame, making even better use of the temporal redundancy

Reference frames are implicitly determined in an MPEG video stream by the ordering of the frames inside the bitstream (cf. figures 6.1 and 6.2 on page 68).

The details how pictures are coded are of little interest for the following discussion; one thing that should be noted however is that MPEG pictures are not coded in *RGB*, but instead in the *YCbCr* (often also wrongly referred to as *YUV*) color space (see e.g. [11]); this representation is advantageous for compression as it allows treating the different color components according to their importance to the human perception system. On RGB display systems this makes it necessary to perform a color space transformation before display.

As can be seen from this short description, the picture decoding requirements for MPEG 1 are fairly involved already, and provide a good test bed for the XdV framework. Particularly the decoding dependencies of the different frame types provide a good way to demonstrate the motivation behind the design of the extension.

5.2.2 Implementation

The MPEG1 decoder implementation was borrowed from `libmpeg2`[18], but had to be modified in a number of ways to be suitable for integration into the X server. Considerable portions

of code could be left out from the library, as this functionality is provided in the Xdv layer already: determining reference pictures, decoder scheduling and color space transformation.

Reference pictures are determined through the generic Xdv mechanisms as described in section 5.1.2. In particular this requires the client submitting the video data to the server to extract this information out of the video stream.

The Xdv layer also takes care of scheduling the decoder routines in the appropriate order; that is, a client requesting a B frame to be decoded will cause the Xdv layer to decode all necessary frames in the correct order to obtain the requested picture.

The implementation consists of the adapted `libmpeg2`, and a small wrapper of just around 500 lines of C code to interface with the Xdv layer, filling in the function stubs discussed in section 5.1.4. This serves to show that plugging in new video codecs is a fairly easy and straight-forward task.

5.2.3 Client interface

The MPEG1 sample codec module expects to be supplied with individual frames for decoding; the frame data to be submitted has to include the picture header and all slices comprising the picture; GOP and sequence headers must not be sent along with the frame data⁴.

Decoding dependencies of B- and P-frames are implicitly determined in an MPEG1 video stream, but they need to be explicitly conveyed as frame meta-data; thus the application submitting the frame data has to perform some amount of parsing of the MPEG1 bit-stream.

The information contained in the sequence header (width and height of the images in the stream, quantizer matrices) are supplied as stream properties (see section 5.1.3). The names of these properties are `XDV_WIDTH`, `XDV_HEIGHT`, `XDV_INTRA_QUANTIZER_MATRIX` and `XDV_NONINTRA_QUANTIZER_MATRIX`; width and height are indicated as 16-bit integers, while the matrices must be given as arrays of 64 8-bit integer values in the same format as the corresponding matrices contained in the sequence header.

The sample programs discussed in chapter 6 will show how the MPEG decoder module is intended to be used in practice.

⁴see [4] for the definition of these terms and the full semantic of the MPEG1 bit-stream

5.3 Design and implementation of the Xtime extension

This section will describe the timing and synchronization component of the streaming media extension, `Xtime`.

5.3.1 Implementation strategy

The purpose of this extension is to enable X clients to communicate timing and synchronization requirements to the X server. It is desirable to have a generic facility that is applicable to all drawing operations supported by the X server, not specific to video.

The basic idea behind the implementation is to have the client submit X requests to the server which are not executed immediately, but instead are queued for execution at a predefined later point in time. Basically there are two options for accomplishing this goal:

1. modify the X protocol to tag every drawing operation with the destined time for execution
2. provide special X protocol requests to mark other requests for later execution

Of these options the first has the benefit of being more elegant from a design point of view, but would be very invasive into the X architecture and would break compatibility with existing X implementations. Therefore the second approach has been chosen for this implementation. Again there are two choices how to mark protocol requests for later execution:

1. send the scheduled requests as ordinary protocol elements and prepend (or brace) them with special marker protocol elements indicating the time of execution
2. introduce a protocol requests carrying other protocol requests “piggy-backed”, together with timing information

The first option has the advantage that ordinary `Xlib` functions can be used to generate the protocol elements on the client side; however this option would drastically change the X protocol semantics, as it makes the protocol stateful. The second option can be implemented without changing the protocol semantic as each request is still self-contained, but as a downside generating the “requests inside requests” is more difficult on the client-side, and the standard `Xlib` functions can only be used for this purpose with a number of difficulties (discussed below). In this implementation the second option has been chosen (cf. also figure 5.4), as it is conceptually cleaner.

5.3.2 General implementation notes

As outlined in the previous paragraphs, the basic idea of scheduling is to have ordinary X protocol requests executed at a later point in time. Only considering the protocol, the problematic part with this approach is the sequence numbering ordinarily implied with each request, and matters are further complicated by the fact that some requests may result in an appropriate reply being generated.

The question to be answered is how sequence numbering should be applied to the scheduled requests, and the only correct answer is to apply numbering as the requests are submitted. The simplest approach to numbering in this case is to include the piggy-backed requests in the X protocol sequence numbering, but two problems arise with this approach:

1. request numbering does not match the number of requests that are transmitted over the wire (because scheduled requests are piggy-backed)
2. since execution order under this scheme may be different from submission order, replies and errors received by the client may have non monotonically increasing sequence numbers

The first of these problems does not have much significance to the client and the server in practice as long as client and server agree on the numbering (which the current implementation guarantees). However the second problem breaks `Xlib`'s monotonicity assumption and seriously confuses the client libraries.

The bottom line is that it is not really possible to use protocol requests that potentially generate a reply in a scheduled operation. Unfortunately it is quite impossible to remove this "defect" without invasive modifications into both the server core and the client libraries, while one of the goals for this sample implementation has been to be performed with little or no intrusion into the core code.

5.3.3 Schedulers and clocks

Schedulers themselves have no notion of time. To provide `XtimeSchedulers` with time information, an abstract clock interface, analogous to the one discussed in section 2.3 is provided; it allows to use different time sources for schedulers. Currently only one time source is available (the system clock), but this abstraction is included here as a place to later hook in audio. It is up to the client application to choose which source is to be used for a particular scheduler.

5.3.4 Xtime client API

This section will describe how the API functions supplied by the `Xtime` extension should be used by applications. Detailed information on and declaration of the functions mentioned here can be found in Appendix B.

Extension initialization

Most X extension modules export a function that enables the client to query for existence and capabilities of a particular extension, and `Xtime` adheres to this convention. To make sure that the `Xtime` extension is supported by the X server in question, every application should call `XtimeQueryExtension`.

Creating and destroying schedulers

Throughout the client library, schedulers are represented through their resource id. They are allocated and destroyed by the client through the `XtimeCreateScheduler` and `XtimeDestroyScheduler` functions.

Scheduling requests

Requests are submitted to scheduler via either the `XtimeSchedule` function, or by the `XtimeBegin`, `XtimeEnd` pair of functions. `XtimeSchedule` expects as one of its parameters a buffer containing the X protocol requests to be performed; generation of these protocol requests is the client application's task.

A more convenient API is offered through the `XtimeBegin` and `XtimeEnd` functions; these functions employ the buffering policy of `Xlib` to generate the piggy-backed requests which the client application has to generate itself if it is using `XtimeSchedule` (see section 3.4 and description of the X protocol requests in section 3.3 as reference). The workings of these function is depicted in figure 5.3; `XtimeBegin` inserts an (incomplete) request into the `Xlib` buffer, keeping a reference to the request; calling `XtimeEnd` later will modify the size of this partial request to include all X requests submitted until this point.

The difficulty in this approach lies in controlling the flushing behavior of `Xlib`, which obviously must not happen between the calls to `XtimeBegin` and `XtimeEnd`. For this reason,

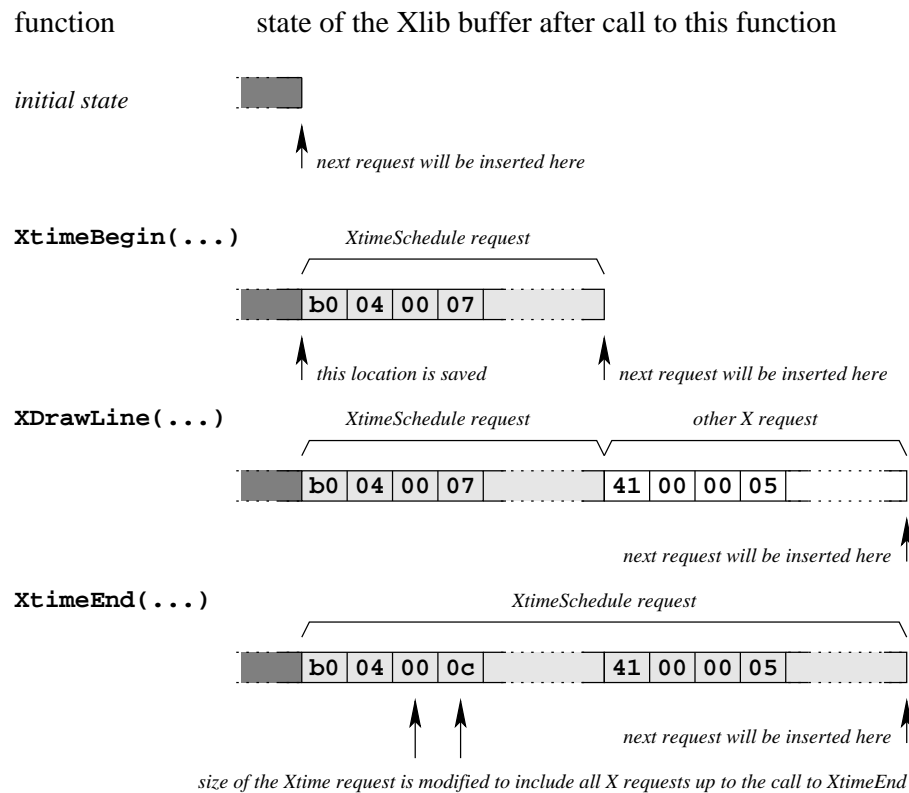
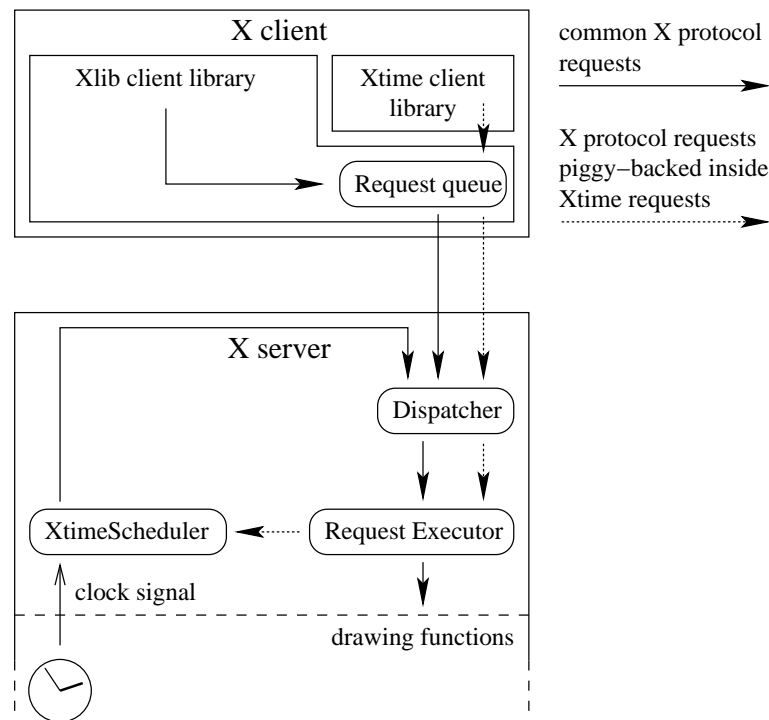


Figure 5.3: Embedding X protocol requests by using `XtimeBegin` and `XtimeEnd`

X functions which either implicitly or explicitly flush the buffer *must not* be called between `XtimeBegin` and `XtimeEnd`. To prevent implicit flushing when the buffer becomes too full, `XtimeBegin` is passed an argument to reserve a predefined amount of space in the buffer, and which must be sufficiently large to keep all requests. Lastly, synchronous operation has to be temporarily deactivated around these calls. To prevent multi-threaded clients from interfering with the buffer while a request is constructed this way, the lock protecting this buffer is held across `XtimeBegin` and `XtimeEnd`.

If the above constraints are honored, the functions provide a convenient interface to scheduling requests; while this approach may at first look more like a clever hack, it can in fact be turned into a clean implementation, although this would require a number of core changes in the workings of the Xlib client library.



Scheduled requests are sent piggy-backed inside Xtime requests and reinjected into the dispatcher at a later point in time

Figure 5.4: Data flow and dispatching of scheduled X requests

Choosing a clock source

To provide a scheduler with a source of time information, it has to be assigned a clock source, which is done using the API function `XtimeSetClockSource`. As one of its arguments it expects the clock to be used; since currently there is no other clock available besides the system clock, its value is ignored at the moment.

5.3.5 Server-side implementation

The functionality of the `Xtime` extension is provided by a loadable X server module. Implementation of the `XtimeScheduler` resource is straight-forward: A new resource type is allocated in the module initializer, appropriate dispatcher functions facilitate creation and manipulation of the resource.

Clock abstraction

Clocks are represented inside the server through the `XtimeClockSource`; this structure provides a number of functions that are used by the `XtimeSchedulers` to provide time information: `CreateTimer` and `DestroyTimer` are used to register and unregister timer callback functions; `GetClockResolution` delivers the resolution (ticks per second) used by this clock, and `GetClockValue` returns the current tick value of the clock.

The implementation of the only available clock at the moment maps these calls onto timer- and timer-callback functions provided by the X server.

Execution of scheduled requests

To notify the `Xtime` schedulers at appropriate points in time, timer callback functions are registered with an `XtimeClockSource`. In these functions the implementation looks at the next pending group of requests, and if they have not yet expired and dependencies are met, executes each request in turn. The path scheduled requests take through the system is illustrated in figure 5.4.

This mechanism bypasses both the X server scheduler as well as the main dispatcher loop, which has a number of consequences as discussed in section 7.1.2. However this turned out to be the only fashion in which this functionality could be provided, as the XFree86 server does not provide any hooks into the main dispatcher loop and the scheduler.

6 Sample Applications

This chapter will show how the APIs offered by the `Xdv` and `Xtime` extensions can be used to express the demands of common multimedia applications. Specifically, two MPEG player applications will be discussed: A very simple player showing how one can very easily make use of the provided extensions, and a more sophisticated player illustrating the motivation behind some of the more advanced functionality.

6.1 A simple MPEG1 player application

The full source code of the simple player presented in the following sections can be found on the accompanying media in the `mpeg_play.c` file. It is able to play back MPEG1 video streams; MPEG system streams have to be demultiplexed before they can be fed into this player.

The structure of this player will be discussed in detail, putting emphasis on showing how the functionality provided by `Xdv` and `Xtime` is used herein. Refer to appendix A and appendix B for declaration of and more information about the functions used.

6.1.1 Initialization

The first step `mpeg_play` will do is to open the indicated video stream (for simplicity all of the video data is mapped into the player's memory at this point) and extract some basic parameters of the video. Then the application continues with the usual X initialization, creating a window of appropriate size for the video and checking for presence of the required `Xdv` and `Xtime` extensions. After that, the scheduler and video stream resources are set up in `setup_player` before entering the main playback loop.

```

Display *dpy;
int end_of_stream=0;

void check_extensions(void)
{
    if (XdvQueryExtension(dpy, ...) != Success) abort();
    if (XtimeQueryExtension(dpy, ...) != Success) abort();
}

int main(int argc, char **argv)
{
    if (argc < 2) abort(); /* need filename as first argument */
    open_mpeg_video_stream(argv[1]);
    open_display_and_create_window();
    check_extensions();
    setup_player();
    while (!end_of_stream) mainloop();
    close_down();
}

```

6.1.2 Setting up the player and the mainloop

First, the required `stream` and `scheduler` resources are created, and some options pertaining to the stream to be played are set; width and height of the video images were obtained from the MPEG sequence header inside the `open_mpeg_video_stream` function and are communicated to the decoder as properties.

Then, the server buffers are filled with at least 6 video pictures (`send_frames`), and playback for these pictures is scheduled (`schedule_playback`); both functions are discussed in detail below. Finally the scheduler is kicked off and playback can begin; the time-scale used for the scheduler is set to the frame rate of the video (again previously obtained from the MPEG sequence header), so the integral numbers 0, 1, 2, ... denote the points in time when the corresponding frames should be displayed.

```

XID stream, scheduler;
int sent_frames=0, played_frames=0;
int frame_rate;

void setup_player(void)
{

```

```
stream=XdvCreateStream(dpy, Atom_MPEG1);
XdvSetStreamProperty(dpy, stream, Atom_XDV_WIDTH, 16, 1, &video_width);
XdvSetStreamProperty(dpy, stream, Atom_XDV_HEIGHT, 16, 1, &video_height);

XdvSetOutputDrawableRGB(dpy, stream,
    window, XVisualIDFromVisual(DefaultVisual(dpy, 0)));

scheduler=XtimeCreateScheduler(dpy);

while(!end_of_stream && (sent_frames<6)) send_frames();
for(played_frames=0; played_frames<sent_frames; played_frames++)
    schedule_playback(played_frames);

XtimeSetClockSource(dpy, scheduler, XTIME_DEFAULT_CLOCK, 0, frame_rate);
}
```

In the mainloop the player application has to feed more data to the X server and schedule playback of the submitted pictures periodically. A simple delay function is used to make sure frames are submitted at roughly the same speed as they are consumed by the server – this does not have to be exact as the server is performing all necessary steps to decompress and display the images autonomously. If there are any X events pending (e.g. window resize notifications), they are handled as well.

```
void mainloop(void)
{
    int n;
    while (!end_of_stream) {
        n=send_frames();
        while(played_frames<sent_frames)
            schedule_playback(played_frames++);

        while(XPending(display)) handle_x_events();

        delay_frames(n);
    }
}
```

6.1.3 Transmitting frames

Two utility functions take over the task of parsing the MPEG stream; `find_next_picture` looks for the boundaries of the next frame in the MPEG stream, advancing the stream pointer to

the end of the next frame; it also checks for the end of the stream and sets the `end_of_stream` marker flag. `picture_type` identifies the type of the frame as an I-, P- or B-frame.

```
char *streampos;

void find_next_picture(char **start, char **end);

#define I_FRAME    1
#define P_FRAME    2
#define B_FRAME    3

int picture_type(char *data);
```

The next utility function, `send_single_frame` sends an individual frame to the server. It uses the frame numbers stored in the array `reference_frames` as forward and backward references, if the frame is a P- or B-frame.

```
int reference_frames[2]={-1, -1};

void send_single_frame(int frameid, char *start, char *end)
{
    int type=picture_type(start);
    switch(type) {
        case I_FRAME:
            XdvCreateFrame(dpy, stream, frameid, 0, 0, NULL);
            break;
        case P_FRAME:
            XdvCreateFrame(dpy, stream, frameid, 1, 1, &reference_frames);
            break;
        case B_FRAME:
            XdvCreateFrame(dpy, stream, frameid, 1|2, 2, &reference_frames);
            break;
    }
    XdvFrameData(dpy, stream, frameid, end-start, start);
}
```

The `send_frames` function is the main function for transmitting frame data to the server; it looks at the frames found in the stream to determine their type and performs forward scanning into the stream to determine the display order and decoding dependencies of pictures; the relationship between the different orderings and the implied dependencies are illustrated in

figures 6.1 and 6.2 on page 68. It will then send the next I- or P-frame and any preceding (in terms of display order) B-frames.

```
int sent_frames=0;

int send_frames(void)
{
    int type, n, B_frames=-1;
    char *frame_start, *frame_end, *last_streampos;
    char *B_frame_start[25], *B_frame_end[25];

    find_next_picture(&frame_start, &frame_end);
    if (end_of_stream) return 0;

    type=picture_type(frame_start);

    do {
        B_frames++;
        last_streampos=streampos;
        find_picture(&B_frame_start[B_frames], &B_frame_end[B_frames]);
        if (end_of_stream) break;
        n=picture_type(B_frame_start[B_frames]);
    } while (n == B_frame);

    /* the stream pointer is now one past the next I or P frame; restore
    it, so we start with this frame next turn */
    streampos=last_streampos;

    reference_frame[1]=sent_frames+B_frames;
    send_single_frame(sent_frames+B_frames, frame_start, frame_end);
    for(n=0; n<B_frames; n++)
        send_single_frame(sent_frames+n, B_frame_start[n], B_frame_end[n]);
    reference_frame[0]=reference_frame[1];

    sent_frames+=B_frames+1;
    return B_frames+1;
}
```

6.1.4 Frame playback

Frame playback is achieved by scheduling decoding and display of the video images at appropriate points in time; note that the time scale for the scheduler was set to equal the frame rate

of the video, so the interval `[frameid, frameid+1)` denotes the interval in time when the frame numbered by `frameid` should be visible.

```
void schedule_playback(int frameid)
{
    XtimeGroup group;
    group=XtimeBegin(dpy, scheduler, frameid, 0, frameid, frameid+1, 0, 1024);
    XdvCopyPictureToDrawable(dpy, stream, frameid,
        gc, window,
        0, 0, video_width, video_height,
        0, 0, window_width, window_height);
    XtimEnd(group);
}
```

6.1.5 Analysis of the sample player

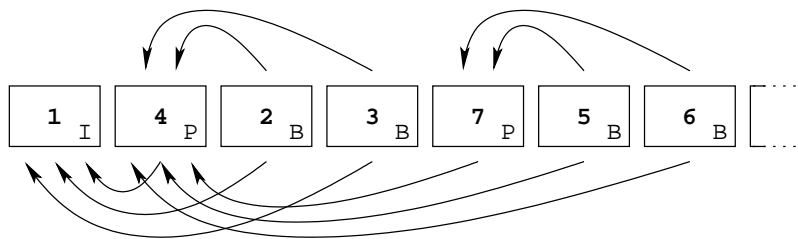
The code samples demonstrate how a simple video player can be constructed utilizing the `Xdv` and `Xtime` extensions; especially it shows how the data transport and display stage are decoupled – in fact, the player can be trivially split up into two separate programs, one supplying the data and the other controlling the playback, by splitting up the main loop accordingly.

However in the case at hand the player is not utilizing all information available to optimize the decoding process. As figure 6.3 on page 68 illustrates, the decoding work is not evenly distributed among the timeslots, resulting in load spikes when the first in a sequence of B frames has to be displayed. While the situation at hand could be improved using some heuristics on part of the `Xdv` extension, it can also be made explicit through the player.

A smarter way to schedule playback and decoding is depicted in figure 6.4 on page 69, and how to achieve this using the presented infrastructure shall be the topic of the next section.

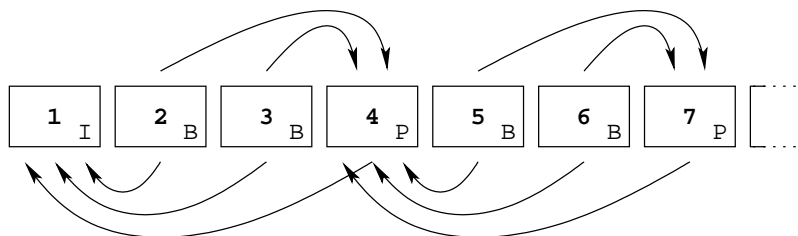
6.2 Improved MPEG1 player

One of the shortcomings of the player presented in the previous section has already been mentioned, but there are others. This section will present a player that improves on the previous example in the following ways:



Numerals denote the intended display order; the ordering of pictures in the bitstream also determines the order in which pictures have to be decoded. Arrows indicate the implied decoding dependencies.

Figure 6.1: Bitstream ordering of pictures in a typical MPEG1 video stream



The same stream as above, but pictures reordered to display order

Figure 6.2: Display order

Timeslot	1	2	3	4	5	6	7
Decoded pictures	1 _I	4 _P 2 _B	3 _B		7 _P 5 _B	6 _B	
Displayed picture	1 _I	2 _B	3 _B	4 _P	5 _B	6 _B	7 _P

The same stream as above when decoded using the simple MPEG1 player

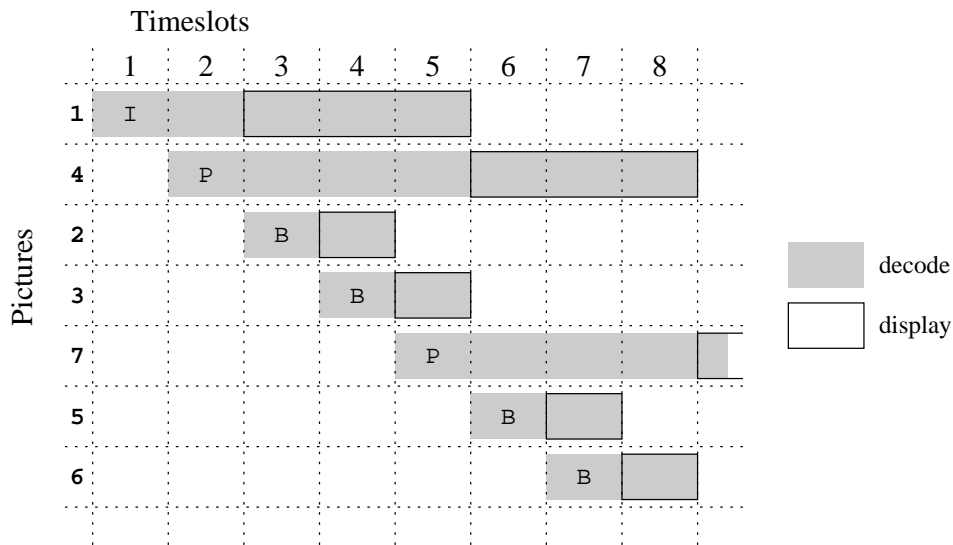
Figure 6.3: Decoding behavior of the first sample player

Timeslot	1	2	3	4	5	6	7
Decoded pictures	1 _I	4 _P	2 _B	3 _B	7 _P	5 _B	6 _B
Displayed picture			1 _I	2 _B	3 _B	4 _P	5 _B

Figure 6.4: Desirable decoder and display scheduling

- (1) **Distribute decoder work more evenly.** Decoding and playback should be scheduled in a way such that during every timeslot exactly one frame is decoded and one frame is displayed scheme that achieves this goal, i.e. like is shown in figure 6.4.
- (2) **Separate decoding from display.** The presentation timing is not as exact as it could be because the decode operation is coupled with the blit operation onto the screen; as a result, the blit operation is inherently subject to jitter when the complexity of the decompressing process varies between images.
- (3) **Frame skipping.** When the machine running the display server cannot keep up, frames have to be dropped; since the `Xtime` extension is agnostic to the structure of the video format, it has no preference for frames to be skipped; as a result a P or I frame may be “dropped” (i.e. it is not displayed), but it has to be decoded nevertheless because it is referenced by following B or P frames. One generally would therefore prefer to display the late P or I frame and drop the following B frame instead.

A solution to all of the problems above is achieved by providing more semantic information about the video stream to the `Xtime` functions. (1) requires that pictures are requested from the client in decoding order. (1) and (2) additionally require that decoding is performed into a backbuffer pixmap instead of directly onto the screen; this is achieved by scheduling two operations for every frame: The blit operation, to be executed during the interval $[t_{frame}, t_{frame+1})$, and the decoding operation, scheduled at some earlier (though possibly overlapping) interval in time, e.g. $[t_{frame-1}, t_{frame+1})$. (3) can then be addressed by extending the expiry time for the decoding and display of I and P frames to overlap with the decompression and presentation of the dependent B frames.



Validity intervals for decompression and presentation of the frames from the sample MPEG stream; shaded boxes denote the validity interval for the decompression operation; solid rectangles denote the validity interval for the blit operation. Note that the validity intervals overlap.

Figure 6.5: Smart decoder and display scheduling

The resulting decoding and display scheduling strategy taking into account these considerations is depicted in figure 6.5. The following properties of this strategy are evident from the graphics and show that the scheduling strategy is “optimal” in terms of resource utilization and graceful handling of resource over-utilization:

- If no deadlines are missed, no frames are skipped and all operations can be completed near the beginning of their validity interval, then exactly one decode and one blit operation is performed per timeslot (startup not taken into account), i.e. it will result in the optimal scheduling as shown in figure 6.4
- If deadlines are missed, B frames are skipped preferentially, while I and P frames may be displayed late; e.g. due to the overlapping validity intervals, picture 4 might be displayed late in timeslot 7, though picture 5 was skipped.

The following sections will demonstrate how the first player has to be modified it to incorporate this strategy.

6.2.1 Backbuffers

Decoding cannot be performed directly onto the screen but needs to be done into backbuffers in this case, and it can be shown that at most four backbuffers are needed. Accordingly, four pixmaps are created during player initialisation, and for each backbuffer the point in time when it becomes available for reuse is kept track of:

```
Pixmap backbuffer[4];
int backbuffer_available[4]={0, 0, 0, 0};

void open_display_and_create_window(void)
{
    ...
    for(n=0; n<4; n++)
        backbuffer[n]=XCreatePixmap(dpy, window, window_width,
            window_height, DefaultDepth(dpy, screen));
    ...
}
```

6.2.2 Picture decoding and playback

Since decoding and playback are now to be done in distinct timeslots and the presentation time of a single frame may comprise more than one timeslot, the `schedule_playback` needs more arguments to describe these points in time.

First it determines which of the four backbuffers is available during the time period in question, and marks this buffer unavailable until the end of this interval. It then schedules two independent set of operations: The first composes the image to be displayed into the selected backbuffer; this consists of decoding the video image and drawing an informational message in the upper left corner. The second performs the blit operation into the visible window area; obviously this blit operation must only be performed if the previous drawing operations were executed successfully, so the `XTIME_DEPENDS` flag is set and the dependency on the first set of operations is communicated to the server.

```
void schedule_playback(int frameid, int start_playback, int end_playback)
{
    XtimeGroup group;
    char s[80];
    int n, len;
```

```

len=snprintf(s, 80, "Frame number %d", start_playback-2);

for(n=0; n<4; n++) {
    if (backbuffer_available[n]<=frameid) break;
}
backbuffer_available[n]=end_playback;

group=XtimeBegin(dpy, scheduler, frameid*2, 0,
    frameid, end_playback, 0, 1024);
XdcvCopyPictureToDrawable(dpy, stream, frameid,
    gc, backbuffer[n],
    0, 0, video_width, video_height,
    0, 0, window_width, window_height);
XDrawString(dpy, backbuffer[n], gc, 16, 16, s, len);
XtimeEnd(dpy, group);

group=XtimeBegin(dpy, scheduler, frameid*2+1, XTIME_DEPENDS,
    start_playback, end_playback, frameid*2, 1024);
XCopyArea(dpy, backbuffer[n], window, gc, 0, 0, window_width, window_height,
    0, 0);
XtimeEnd(dpy, group);
}

```

6.2.3 Transmitting frames

In this context it makes sense to couple transmission of the frame data and decoder/playback scheduling more closely than in the first player. Since the forward scanning into the stream is more complex, a queue of pictures to be processed is introduced. The `send_frames` function, performing a similar task will then parse into the stream, queue the parsed frames and when a “sufficient” number of frames have been identified to allow determination of the display order and validity intervals according to figure 6.5, these frames are transmitted and scheduled for playback in the function `send_queued_frames` (discussed later).

```

struct {int type; char *start, *end;} picture_queue[25];
int picture_queue_length=0, queued_i_p_frames=0;

void queue_picture(int type, char *start, char *end)
{
    picture_queue[picture_queue_length].type=type;

```

```

picture_queue[picture_queue_length].start=start;
picture_queue[picture_queue_length].end=end;
picture_queue_length++;
}

void send_frames(void)
{
    int type;
    char *frame_start, *frame_end;

    while(!end_of_stream) {
        find_next_picture(&frame_start, &frame_end);
        if (end_of_stream) break;
        type=picture_type(frame_start);
        queue_picture(type, frame_start, frame_end);
        if ((type==I_FRAME) || (type==P_FRAME)) {
            queued_i_p_frames++;
            if (queued_i_p_frames==3) {
                send_queued_frames();
                return;
            }
        }
    }

    while(queued_i_p_frames) send_queued_frames();
}

```

The `send_queued_frames` function detaches the first I or P frame from the queue together with any number of immediately following B frames. The decoding and display times of these pictures can then be determined observing that

- the decoding time for each frame starts at the time slot corresponding to the position of the frame in the bitstream
- B frames are displayed exactly one timeslot after they have been decoded
- the I or P frame at the beginning of the queue is to be displayed after the following B frames, and its presentation interval spans the subsequent set of B frames (their decoding depends on the I or P frame as a backward reference).
- the function is called when exactly three I or P frames have been queued (or the end of stream has been reached).

These properties are easy to verify from the source code, and it is as easy to see that they lead to the scheme illustrated in figure 6.5.

```

void send_queued_frames(void)
{
    int n, bframes=1;
    while((bframes<picture_queue_length) && (picture_queue[bframes].type==B_FRAME))
        bframes++;
    reference_frames[1]=sent_frames;

    assert((picture_queue[0].type==I_FRAME) || (picture_queue[0].type==P_FRAME));

    send_single_frame(sent_frames, picture_queue[0].start, picture_queue[0].end);
    schedule_playback(sent_frames,
        sent_frames+bframes+1, sent_frames+picture_queue_length);
    sent_frames++;
    for(n=1; n<bframes; n++) {
        send_single_frame(sent_frames, picture_queue[n].start, picture_queue[n].end);
        schedule_playback(sent_frames, sent_frames+1, sent_frames+2);
        sent_frames++;
    }

    for(n=bframes; n<picture_queue_length; n++)
        picture_queue[n-bframes]=picture_queue[n];

    picture_queue_length-=bframes;

    queued_i_p_frames--;
    reference_frames[0]=reference_frames[1];
}

```

6.2.4 Setting up the player and the mainloop

The rest of the player can is mostly identical to the first example; the mainloop and the player setup routine are in fact slightly simpler than in the first example as the `send_frames` function also implicitly schedules displaying the pictures. For this reason their source code is not repeated here.

7 Discussion

This chapter reports about the lessons learned during design and implementation and will discuss to what extent the goals stated in the preface have been achieved.

7.1 Prototype implementation

The experience gained through the implementation and testing process shows that the X Window System in general and the XFree86 implementation in particular exhibit a truly remarkable extensibility. Essentially the X_{dv} extension blends into the existing infrastructure without any hassle; X_{time}, while troublesome in certain ways, was still possible to be implemented as a loadable module and without touching core server code *despite* the fact that it invalidates a number of the assumptions underlying X and is quite invasive into core components of the system.

Yet a number of shortcomings were uncovered that would need to be addressed in the future development to make XFree86 a more suitable platform for this approach to multimedia applications; the following sections list some of the obstacles encountered and provides suggestions how they could be removed.

7.1.1 X_{dv} extension

The problems around the X_{dv} extension are mostly caused by the large amount of processing required to decompress the video frames.

Efficiency

While care has been taken to keep the implementation efficient, the system is considered only a prototype not much time has been spent optimizing it either. Instrumentation revealed that more than 90% of CPU time is spent inside the decoder and color space transformation functions, so no gains from optimization in the generic `Xdv` layer are to be expected. Performance is therefore identical as if the decoding had been performed outside of the X server, although for reasons outlined below the system *appeared* to perform worse from the subjective experience of the user.

The decoder functions themselves turned out to be reasonably efficient already: On a test system (AMD Athlon 1.3GHz) it was possible to play back five simultaneous MPEG1 streams at 1.150 MBit/second at CIF resolution (352×288) without any frame dropping at about 80% CPU load; or alternatively one MPEG1 stream scaled to full screen resolution (1280×960) in software at around 60% CPU load. In both cases however the user interface became noticeably jerky, e.g. moving windows using the mouse caused the windows to “jump” along the movement path instead of moving smoothly.

Asynchronous decompression

The single-threaded event-driven dispatching model of the XFree86 server results in all graphics operations being strictly serialized at the dispatching level already. As a result the implementation is not particularly well-suited for overlapped processing between operations that are performed on the host CPU and operations that are performed by the graphics processing unit.

In fact during testing of the `Xdv` extension it became apparent that while the host CPU was busy performing the decompression of the video images, the graphics processing unit was mostly idle. Depending on the speed of the machine and the number of streams being decoded simultaneously the consequences were very perceptible to the user, especially when comparing the behavior of the `Xdv` extension to other approaches which perform the decompression outside the X server: The user interface noticeably lost responsiveness, resulting in a less pleasant user experience. In fact, although the machine did not run materially slower, it was nevertheless perceived as performing significantly worse.

The obvious, but unfortunately quite difficult answer to this problem would be to change the dispatching model of the XFree86 server, e.g. by threading the server.

A different option that is implementable with moderate intrusion into the core XFree86 architecture would be to split up the decoding process into small chunks and pause the decoder (and requesting client) between two chunks if there is other work pending. While this idea is in principle implementable using the currently available infrastructure (using `QueueWorkProc`, `ClientSleep` and `ClientSignal` functions), the mechanisms were not really designed for computationally expensive tasks and do not interact well with the X scheduler – for instance, the time spent inside the queued procedures is not accounted to the requesting client.

Asynchronous data transport

Related to the previous problem is a second issue concerning data transport. Due to the nature of the XFree86 server it is unable to receive incoming requests from the network during request execution; if requests are computationally expensive (as in the `Xdv` extension), the server may not be able to receive data as fast as the client would like to send it. Especially when comparatively large amounts of data have to be transferred from the client (like the frame data for `XdvStreams`), the net effect is that the queue of buffered frame data is drained faster than it can be refilled.

During experiments this effect was visible on slower systems (Pentium 133MHz), and it manifested in the server running out of data during playback of compute-intensive videos.

While all of the options mentioned in the previous section could improve the situation, it again appears that reworking the dispatching model of the X server is the only viable option.

Code complexity

It is a generally established design principle to keep software as simple as possible and focussed on one specific task. The rationale behind this are stability and maintainability as well as security considerations. Pushing a significant and complex amount of functionality such as decoding compressed video streams into the X server obviously violates this principle, so the question remains whether the benefits of this approach outweigh the drawbacks.

The alternative to full integration into the X server would be to accompany the X server with a separate “media server”, conceptually separating simple graphics operations and media streams. While this approach is conceptually appealing from a software design point of view, it is facing a number of technical difficulties:

- To provide support for hardware-accelerated video decompression it is necessary to access the same hardware resources as the X server does, likely requiring locking between the two components
- Related to the previous problem is the issue of timing and synchronization; it requires the “media server” to interact with the X scheduler
- The semantical strength of `Xdv`, to be able to perform compositing and other graphics operations on the images before displaying them, would be very hard to achieve
- Using a separate media server and possibly a separate communication protocol introduces new security issues for authenticating and securing the privacy of the communication; using the X protocol existing solutions can be leveraged¹

While all of the problems can undoubtedly be solved putting enough effort into the implementation, it appears fair to assume that the complexity of this approach would not be lower at best; rather it can be expected that the resulting system would be significantly more complex and difficult to maintain than an integrated solution.

Relationship to other X extensions

Over the years a large set of X extensions have been developed, and some of them provide functionality that is in ways related to the services provided by `Xdv`: of these, `XVideo`, `Xv_MC` and `XIE` are most closely related.

`XVideo` provides a “black-box” interface to video sources (called “ports” in the `XVideo` extension); the API allows little more than specifying a screen area where the video should be visible to the user. `XVideo` ports also provide an abstraction of various hardware acceleration capabilities (scaling, overlays, color space conversion), so the obvious solution to make use of these features from `Xdv` is to bind an `XVideo` port to an `XdvStream` in a fashion similar to the `XdvSetOutputDrawableRGB` function.

`Xv_MC` is an extension of the `XVideo` extension and provides access to hardware acceleration features helpful for MPEG decoding (IDCT and motion compensation). While this functionality could well be used by an MPEG decoder, the assumptions behind `Xv_MC` are that X client

¹for example `ssh`, cf. [15]

and X server are running on the same machine and it is thus unsuitable for networked operation. It does only provide a hardware abstraction layer, but no hardware emulation layer for these functions, requiring the client to fall back to performing relevant parts of the decoding process itself. Moreover it assumes that shared memory transport can be used for the data, and consequently does nothing to keep the data in a compressed state for the communication between client and server (Huffman decoding has to be performed by the client). As a result, it is of little use for distributed multimedia applications, and even in the case of local clients the decompression process can be improved when all decoding steps are kept tightly coupled².

Related in a completely different way is **XIE**³, the X Image Extension; it deals with transmitting images between client and server in various popular formats, e.g. JPEG, and simple image manipulation operations. This extension is deprecated however as the network traffic required for transmission of still images was not considered a major impact, and most of the image manipulation functionality has been superseded by other extensions (most notably **Xrender**).

Interlaced video

The API is designed to cover a wide range of video encoding formats. However it is not conceivable whether it is very suitable for dealing with half-picture encodings (such as MPEG2 can support).

There is certainly no problem in adapting the semantics of the **Xdv** functions and replacing each occurrence of “picture” with “half-picture”, e.g. have **XdvCopyPictureToDrawable** only copy odd/even lines into the output **Drawable**; however this significantly complicates post-processing of the emitted frames (e.g. output scaling has to be done *after* deinterlacing the picture).

Optionally, deinterlacing could be performed inside the **Xdv** extension; this allows to retain the current full-picture semantic, but of course means a loss of semantic and control with respect to the API provided to client applications.

Since neither option is completely satisfactory it has been decided to leave this issue open for further experimentation. It should be noted that there appears to be a general trend of avoiding interlaced encodings in digital video applications, possibly making the lack of support in **Xdv** appear less fatal.

²e.g. Huffman decoding and IDCT, cf. [9]

³cf. [13]

7.1.2 Xtime extension

Request scheduling

Implementation of the Xtime extension has been the most difficult part. While the solution is workable as a demonstrator, it is somewhat unsatisfactory, as the way scheduled requests bypass the main dispatcher loop and the X request scheduler has a number of consequences:

- Since the scheduler is agnostic to these requests, it is unable to account them properly to the client that generated the requests; this can lead to unfair scheduling behavior and starvation problems (as mentioned in the previous section)
- A few X requests (font-related) require the client to be put to sleep, i.e. the request is not completely finished when the corresponding dispatcher function returns, but Xtime is unable to determine this client state; while these functions are unsuitable for use in scheduled operations, the execution scheme used by Xtime should nevertheless be able to treat them correctly

All of these problems stem from the fact that the server does not provide any hooks into the scheduling and request dispatching mechanism to modules. While certainly a number of functions inside the X server exist that allow manipulation of the client state and injection of requests into the request queue, their symbols are not explicitly exported through the XFree86 loader (see section 3.7). In fact the list of symbols exported is only concise and well thought-out when it comes to providing support for hardware drivers – other symbol exports appear to have been added on an “on-demand” basis for standard X extensions; in the case at hand the loader mechanism itself turned out to be an obstacle.

While adding symbols to the export table of the X server would not have been particularly difficult, this approach was not taken – on the one hand, for the reasons mentioned in the beginning of chapter 5; on the other hand because a much cleaner solution would be to rework and modularize the execution scheduler itself.

Client API

The API provided by XtimeBegin and XtimeEnd is convenient, though the current implementation is slightly fragile as care must be taken by the user not to trigger an implicit flush of the

request queue. Better control of the flushing mechanism would be necessary to provide a more robust interface.

The changes required in `Xlib` to accomplish this are be fairly small and self-contained; it can be achieved by providing a hook into the `_XFlush` function internally used by `Xlib`, which enables `Xtime` to move the pending data “out of the way” instead of sending it to the server.

Implementation of these changes is not included here as the goal was to develop a “drop-in” solution.

Relationship to other X extensions

One extension that is quite closely related to `Xtime` is `XSync`. It provides the client with an API to suspend processing of its request queue inside the X server until a pre-defined point in time has reached. While this interface is suitable for simple synchronization and timing tasks, it lacks the expressivity required for sophisticated multimedia applications and has a number of other problems.

Using `XSync` for the timing of a presentation still does require the application to submit a large enough number of requests in advance to keep the server queue filled and compensate for network jitter. However the processing delay intentionally introduced by the client using the `XSync` functions affects all requests sent from this particular client. This side-effect is that all other drawing operations, like repainting the widgets which the graphical user interface is based on, are subject to the same buffering delay as the multimedia presentation, which is at the very best highly undesirable.

This effect is even made worse by the fact that the client-side X library, `Xlib`, generally waits synchronously when requesting a reply carrying data from the server – of course replies are subject to the same delay, and each “blocking” request to the server will cause the request queue to be drained.

Furthermore `XSync` only allows client applications to indicate the point in time when an operation should **start**, but it is unable to set a deadline. This means that all requests will be executed even if the display server cannot keep up with the frame rate intended by the client, and processing delays at each step may accumulate. As a result the client must detect skew in the presentation timing itself and try to reduce the frame rate accordingly – which has considerable difficulties, both due to the processing delays introduced by buffering in the request queue as well as delay induced by the network.

As these considerations show that the “abuse” of the request queue as jitter compensation buffer is not feasible in practice. The `XSync` extension is therefore not really usable for multimedia programming, except for perhaps the most simplest possible applications, and except for cases where the round-trip time between client and server is low and controlled – in which case there is strictly speaking no need for server-driven timing.

The API provided by `Xtime` avoids these problems by allowing the client to communicate the intended semantic much more explicitly, and allows better asynchronicity by cleanly separating the execution of scheduled requests from the client request queues.

7.2 Experiences gained through the prototype

Apart from the issues uncovered during the implementation, other useful experience could be gained from the prototype.

First, the sample programs in chapter 6 show that the X server extensions provide a fairly easy interface to distributed multimedia presentation. Most of the complexity in these programs is concerned with the parsing and processing of the MPEG video bitstream.

Second, it could be demonstrated that the initial goal of providing a network transparent multimedia system can in fact be achieved.

Third, the deep semantic integration into networked graphics systems has proven to be both feasible and beneficial. Especially the possibility of performing complex compositing operations on the images of a video stream (partially utilized in the second sample program) shows the potential of the approach.

Lastly, the X window system, on which the prototype implementation is based, has – despite some shortcomings – proven to be a suitable foundation for further work in this direction.

7.3 Related works

Surprisingly few other attempts have been made to provide network-transparent multimedia systems. One of the earliest is the `XMovie`⁴ system, dating back to 1991.

⁴cf. [16]

Comparing the approach outlined in this paper to other approaches some significant differences in the interpretation of the problem and resulting solution become apparent. Taking `XMovie` as an example, it is basically characterized by the following properties:

- video streams are regarded as a “continuous presentation” instead of “a sequence of individual pictures”
- video streams are very opaque objects; basically the only influence clients can take is to start or stop the presentation
- access to individual pictures does not fit very well into this model, or is entirely impossible
- playback timing and scheduling is not under control of the client; it is determined by the stream itself, and executed within the server
- video is not well integrated with other graphics primitives
- lack of synchronization semantic makes compositing operations difficult; even simple operations such as drawing subtitles over a video image are almost impossible to achieve

In contrast the `Xdv/Xtime` approach offers a very rich semantic. The interpretation of video streams as “sequences of pictures” provides the clients with a much higher degree of control over the presentation, including support for complex compositing operations. Yet it still achieves the benefits of client-independent timing of the presentation through the forward-scheduling mechanism provided by `Xtime`.

7.4 Summary

The prototype implementation shows that the goal of providing a network-transparent multimedia system is in fact feasible. Especially it is possible to provide similarly fine-grained control into the system as non-networked multimedia systems such as `QuickTime`.

It further shows that `X11` and especially the `XFree86` server, while providing a solid basis and good framework, have some rough edges that do not interact well with the stated goals. None of these obstacles are unsurmountable however, and the preceding sections already contain a number of useful suggestions.

7.5 Further work

Some areas that lend themselves very well for future work have been covered already; they mostly deal with changes to the X architecture to remove the obstacles encountered during design and implementation of the `Xdv` and `Xtime` extension.

Integration with other common X extensions such as `XVideo`, to provide overlay support, or `RENDER` (see [14]), to provide support for a more advanced rendering back end, would greatly improve the services already offered by the presented extensions.

A different option includes examination of other networked graphics systems, if they can provide support for networked multimedia as well, or better than the X window system.

The next logical step in extending the overall architecture would be integration of audio support; several networked audio systems are already available today, and it would be interesting if they could be incorporated into the architecture.

On the other end of the network, only very little work has been done on the client side; providing a comprehensive framework and higher-level API that makes use of the basic building blocks presented here could also be a useful starting point.

Bibliography

- [1] *Apple Computers, Inc.:* Inside Macintosh: QuickTime; Addison Wesley 1993
- [2] *Apple Computers, Inc.:* Inside Macintosh: QuickTime Components; Addison Wesley 1993
- [3] ISO/IEC 11172: MPEG-1: Coding of moving pictures and associated audio for digital storage at up to about 1,6 Mbit/s; ISO, 1996
- [4] *Mitchell, J. L., Pennebaker, W. B., Fogg, C. E., LeGall, D. J.:* MPEG Video Compression Standard; Kluwer Academic Publishers, October 1996
- [5] *Steinmetz, R., Engler, C.:* Human Perception of Media Synchronization; IBM Technical Report no 439310, 1993
- [6] *Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V. et al.:* RFC1889: RTP: A Transport Protocol for Real-Time Applications; IETF, January 1996
- [7] *Boutell, T. et al.:* RFC2083: PNG (Portable Network Graphics) Specification; IETF, March 1997
- [8] *Hoffman, D., Fernando, G.:* RFC2250: RTP payload format for MPEG1/MPEG2 video; IETF, January 1998
- [9] *Froitzheim, K., Wolf, H.:* A Knowledge-Based approach to JPEG Acceleration; Rodriguez, Safranek, Delp: Proceedings of the IS&T/SPIE's Symposium on Electronic Imaging: Technical Conference 2419 - Digital Video Compression, San Jose, 1995
- [10] *Froitzheim, K.:* Multimedia Kommunikation; dpunkt Verlag, Heidelberg, 1997

-
- [11] *Chapman, N., Chapman, J.*: Digital Multimedia; John Wiley and Sons Ltd., November 2000
 - [12] *Scheifler, R., Gettys, J.*: The X window system; ACM Transactions on Graphics Vol: 5 (2), 1986
 - [13] *Shelly, R.*: X Image Extension Overview; AGE Logic, Inc., 1994
 - [14] *Packard, Keith*: Design and Implementation of the X Rendering Extension; Usenix Technical Conference 2001
 - [15] *Flegel, U.*: The Interaction between SSH and X11 - thoughts on the Security of the Secure Shell; Technical report, Braunschweig, September 1997
 - [16] *Lamparter, B., Effelsberg, W.*: X-MOVIE: Transmission and Presentation of Digital Movies under X; Network and Operating System Support for Digital Audio and Video, 2nd International Workshop, Heidelberg, November 1991
 - [17] The XFree86 project; <http://www.xfree86.org>
 - [18] *Holtzman, A., Lespinase, M. and others*: libmpeg2 – an MPEG1 and MPEG2 decoder library; <http://libmpeg2.sourceforge.net>

A Xdv client API

Extension initialization

```
int XdvQueryExtension(  
    Display *dpy, unsigned int *version, unsigned int *revision,  
    unsigned int *req_base, unsigned int *ev_base, unsigned int *err_base);
```

Description.

This function checks for availability of the Xdv extension in the X server designated by the `dpy` parameter; it returns the `version` and `revision` number of the extension, as well as the base request opcode, base event number and base error number assigned to this extension into the locations indicated by the parameters.

Diagnostics.

The function returns 0 on success (that is, the extension has been found), or `XdvBadExtension` if the extension is unavailable.

Creating and freeing video stream resources

```
XdvStreamID XdvCreateStream(  
    Display *dpy,  
    Atom codec)
```

Description.

Allocates a new video stream resource, using `codec` to decode the compressed video stream.

Diagnostics.

The function returns the XID assigned to the stream resource

```
int XdvFreeStream(  
    Display *dpy,  
    XdvStreamID streamid)
```

Description.

Destroys a video stream resource previously created with `XdvCreateStream`.

Diagnostics.

The function may raise a `XdvBadStream` error if `streamid` does not refer to a valid video stream resource.

Stream properties

```
int XdvSetStreamProperty(  
    Display *dpy, XdvStreamID streamid, Atom name,  
    unsigned int format, unsigned int nelements,  
    void *data)
```

Description.

Sets the stream property denoted by **name** to the value indicated by **format**, **nelements** and **data**; **format** must be either 8, 16 or 32, indicating that the passed values are 8-, 16- or 32-bit integers; **nelements** is the number of integers passed, while **data** should point to the array of integers

Diagnostics.

The function may raise a **XdvBadStream** error if **streamid** does not refer to a valid video stream resource. It may raise a **BadAtom** error if the atom does not refer to a writeable property of the stream. It may raise a **BadValue** error if the format and length of the data passed is invalid for the property being accessed.

```
int XdvGetStreamProperty(  
    Display *dpy, XdvStreamID streamid, Atom name,  
    unsigned int nbytes, void *data,  
    unsigned int *nelements_return, unsigned int *format_return)
```

Description.

Reads the stream property denoted by **name** into the memory location denoted by **data**, writing at most **nbytes** into this location. The format of the data is written to **format_return**, which upon successful return of the function will contain either 8, 16 or 32 to indicate that the data should be interpreted as 8-, 16- or 32-bit integers; the number of integers contained in the property is returned in **nelements_return**, which may indicate that the property contains more data than the function was able to store in **data**.

Diagnostics.

The function may raise a **XdvBadStream** error if **streamid** does not refer to a valid video stream resource. It may raise a **BadAtom** error if the atom does not refer to a writeable property of the stream.

Common stream properties

The following properties should be available from all kinds of streams:

- `XDV_WIDTH`, `XDV_HEIGHT`: The natural width and height of the images generated from this stream; these properties must be set before initializing output for this stream, for some types of streams they may not be settable at all.
- `XDV_PICTURES_MAX`, `XDV_BUFFER_MAX`: The maximum number of pictures and bytes the stream will need to keep in its buffers; if more pictures or bytes of data are submitted, they are discarded. Setting these properties to a value lower than the current number of pictures or bytes of data however does not cause any data already in the buffers to be discarded.

Event notification

```
int XdvSetEventMask(  
    Display *dpy,  
    XdvStreamID stream,  
    unsigned int mask);
```

Description.

Allows the client to indicate to the server what kind of notifications it is interested in; the `mask` argument must be a bitwise or of the `XDV_NOTIFY_*` values below.

- `XDV_NOTIFY_BUFFER_THRESHOLD`: send a notification when the buffer fill (measured in bytes of data) passes a defined threshold value (see `XdvSelectThresholdNotify`)
- `XDV_NOTIFY_PICTURE_THRESHOLD`: send a notification when the buffer fill (measured in number of pictures) passes a defined threshold value (see `XdvSelectThresholdNotify`)
- `XDV_NOTIFY_BUFFER_OVERRUN`: send a notification when the buffer fill (measured in bytes of data) is about to exceed the defined maximum value
- `XDV_NOTIFY_PICTURE_OVERRUN`: send a notification when the buffer fill (measured in number of pictures) is about to exceed the defined maximum value
- `XDV_NOTIFY_PICTURE_ALLOCATE`: send a notification when a new picture is allocated
- `XDV_NOTIFY_PICTURE_MISSING`: send a notification when a picture was requested, but could not be decoded (data unavailable)

Diagnostics.

The function may raise a `XdvBadStream` error if `streamid` does not refer to a valid video stream resource.

```
#define XdvThresholdNotify 0
typedef struct XdvThresholdEvent {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    XdvStreamID stream;
    Time time;
    int threshold;
    int direction;
    int reason;
} XdvThresholdEvent;
```

Description.

This type of event is sent to interested clients whenever a given threshold value is passed or the buffer is about to overrun. The `reason` member variable is equal to either `XDV_NOTIFY_BUFFER_THRESHOLD`, `XDV_NOTIFY_PICTURE_THRESHOLD`, `XDV_NOTIFY_BUFFER_OVERRUN` or `XDV_NOTIFY_PICTURE_OVERRUN` and indicate the exact reason this notification was sent. In the first two cases the `threshold` and `direction` members indicate the threshold value and in which direction it was crossed (`direction` is either +1 or -1).

No notification is sent unless the corresponding event is activated by the client through `XdvSetEventMask`.

```
#define XdvPictureNotify 1
typedef struct XdvPictureEvent {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    XdvStreamID stream;
    Time time;
    int frameid;
    int reason;
} XdvPictureEvent;
```

Description.

This type of event is sent to deliver notifications relating to a single picture; **reason** may be either `XDV_NOTIFY_PICTURE_ALLOCATE` or `XDV_NOTIFY_PICTURE_MISSING`, while **frameid** identifies the pictures allocated or missing.

`XDV_NOTIFY_PICTURE_ALLOCATE` may be sent in response to `XdvCreateFrame` as described below; its main purpose is to inform the client about available pictures for display when the controlling client is not identical to the data source.

`XDV_NOTIFY_PICTURE_MISSING` is generated whenever the client requests a picture out of the stream, but it can not be provided; this may be because no data has been supplied for this picture at all, or because decoding depends on other pictures which are unavailable themselves.

No notification is sent unless the corresponding event is activated by the client through `XdvSetEventMask`.

```
int XdvSelectThresholdNotify(  
    Display *dpy,  
    XdvStreamID stream,  
    int threshold_type,  
    int threshold);
```

```
int XdvCancelThresholdNotify(  
    Display *dpy,  
    XdvStreamID stream,  
    int threshold_type,  
    int threshold);
```

Description.

These two functions allow setting and removing threshold values for notifications; `XdvSelectThresholdNotify` adds the value indicated by `threshold` to the list of thresholds, while `XdvCancelThresholdNotify` removes it again.

`threshold_type` must be either `XDV_NOTIFY_BUFFER_THRESHOLD` or `XDV_NOTIFY_PICTURE_THRESHOLD`, indicating what quantity the threshold applies to.

No notification is sent unless the corresponding event is activated by the client through `XdvSetEventMask`.

Diagnostics.

The function may raise a `XdvBadStream` error if `streamid` does not refer to a valid video stream resource. `XdvSelectThresholdNotify` may generate a `BadAlloc` error, indicating that the server was unable to allocate the required memory. `XdvCancelThresholdNotify` may generate a `XdvBadNotifier`, indicating that the given threshold was not previously selected with `XdvSelectThresholdNotify` by this client.

Extracting pictures from the stream

```
int XdvSetOutputDrawableRGB(  
    Display *dpy, XdvStreamID streamid, Drawable d, VisualID visual)
```

Description.

Sets up the video stream resource referred to by `streamid` to generate output to an X Drawable on the same screen as the drawable `d`, using `visual` as pixel format.

Diagnostics.

```
int XdvCopyPictureToDrawable(  
    Display *dpy, XdvStreamID streamid, Drawable d, GC gc, int pictureid,  
    short srcx, short srcy, short srcw, short srch,  
    short dstx, short dsty, short dstw, short dsth)
```

Description.

Copies the subrectangle denoted by `srcx`, `srcy`, `srcw`, `srch` of the picture denoted by `pictureid` of the stream `streamid` into the destination rectangle specified by `dstx`, `dsty`, `dstw`, `dsth` of the drawable `d`. The graphics context `gc` is used in the same way as in the function `XCopyArea`. The image is scaled accordingly.

The request may result in a `XdvPictureMissing` event sent to interested clients if the picture denoted by `pictureid` could not be decoded. It may also generate `XdvBufferNotify` and `XdvPictureNotify` events as the data associated with this picture will be freed after the copy operation and causes the buffer fill to change.

Diagnostics.

The function may raise a `XdvBadStream` error if `streamid` does not refer to a valid video stream resource. It may also raise `BadDrawable`, `BadGC`, `BadMatch` and `BadValue` errors, which have a meaning analog to the `XCopyArea` function.

Embedding frame data in the X protocol stream

```
int XdvCreateFrame(  
    Display *dpy, XdvStreamID streamid, int frameid,  
    int refbitmap, int numrefframes, int refframes[])
```

Description.

Allocates resources for a new frame for the indicated stream; the parameter `frameid` will later be used to identify this frame, and should be derived from decoding or presentation sequence numbers or timestamps, as the server assumes that pictures are accessed by monotonically increasing `frameid` (modulo wrap-around). The last three parameters are used to indicate referenced frames, with `numrefframes` indicating the size of the `refframes` array, and `refbitmap` containing a set bit if the corresponding entry of the `refframes` array is valid. The contents of the `refframes` array must be `frameids` of frames allocated earlier (though no data needs to have been supplied for these frames yet, see below).

Diagnostics.

The function may raise a `XdvBadStream` error if `streamid` does not refer to a valid video stream resource.

```
int XdvFrameData(  
    Display *dpy,  
    XdvStreamID streamid,  
    int frameid,  
    int nbytes,  
    void *data)
```

Description.

Supplies data to the previously allocated frame indicated by `frameid`. The data is treated as opaque and not interpreted in any way; it is later passed through to the decoder without modification.

Diagnostics.

The function may raise a `XdvBadStream` error if `streamid` does not refer to a valid video stream resource.

B Xtime client API

Extension initialization

```
int XtimeQueryExtension(  
    Display *dpy, unsigned int *version, unsigned int *revision,  
    unsigned int *req_base, unsigned int *ev_base, unsigned int *err_base);
```

Description.

This function checks for availability of the `Xtime` extension in the X server designated by the `dpy` parameter; it returns the `version` and `revision` number of the extension, as well as the base request opcode, base event number and base error number assigned to this extension into the locations indicated by the parameters.

Diagnostics.

The function returns 0 on success (that is, the extension has been found), or `XtimeBadExtension` if the extension is unavailable.

Creating and destroying schedulers

```
extern XtimeSchedulerID XtimeCreateScheduler(  
    Display *dpy);
```

Description.

This function creates a new scheduler for deferred X request execution.

Diagnostics.

The function returns the XID of the scheduler. It may raise a `BadAlloc` error if the resource could not be created.

```
extern int XtimeDestroyScheduler(  
    Display *dpy,  
    XtimeSchedulerID id);
```

Description.

Destroys a previously created scheduler.

Diagnostics.

The function may raise a `BadScheduler` error if the passed XID does not refer to a valid scheduler resource.

Scheduling requests for deferred execution

```
#define XTIME_NOTIFY_SUCCESS 1
#define XTIME_NOTIFY_FAILED 2
#define XTIME_NOTIFY_EXPIRED 4
#define XTIME_DEPENDS 8

extern XtimeGroup XtimeBegin(
    Display *dpy,
    XtimeSchedulerID id,
    int groupid, int flags, int when, int expires, int depends,
    int reserve_bytes);

extern int XtimeEnd(
    Display *dpy,
    XtimeGroup group);
```

Description.

These two functions must be braced around a group of X requests that should not be executed immediately, but be scheduled for later execution. The first function, `XtimeBegin` sets up a suitable context for deferred execution; `when` and `expires` define the validity interval during which the requests are to be executed. `groupid` allows to attach an identifier to this group of requests; this identifier is used when events about this group are communicated to the client, or to express dependencies. `flags` is a bitmask describing what kind of notifications about this request group the client is interested in and should be a logical or of the `XTIME_` values defined above; the meaning of the individual flag bits is as follows:

- `XTIME_NOTIFY_SUCCESS`: Notify if the group of requests has been executed successfully
- `XTIME_NOTIFY_FAILED`: Notify if the group of requests was executed, but during execution one of the dispatcher functions returned with an error
- `XTIME_NOTIFY_EXPIRED`: Notify if the group of requests was not executed because its expiry time was reached
- `XTIME_DEPENDS`: This group of requests must only be executed if the group of requests referenced by the `depends` argument has previously been executed successfully.

If the `XTIME_DEPENDS` flag is set, then `depends` must refer to the `groupid` of a previously submitted request group; it indicates that the following requests must only be executed if the referenced group was executed as well. `XtimeEnd` closes the group of requests, and must be called with the request group handler, as returned from the `XtimeBegin` function.

Due to the nature of `Xlib` it is necessary to pre-allocate the buffer space required to store the scheduled requests; for this, the caller must supply a `reserve_bytes` value that is larger than the combined size of all X requests generated between the calls to `XtimeBegin` and `XtimeEnd`. If the buffer space is insufficient, the behaviour is undefined. No `Xlib` functions that may implicitly flush the request queue may be used between `XtimeBegin` and `XtimeEnd`; see section 5.3.4 for a description of which functions are affected by this restriction.

Diagnostics.

The functions may raise a `BadScheduler` error if `id` does not refer to the `XID` of a valid scheduler resource.

Choosing a clock source

```
#define XTIME_DEFAULT_CLOCK 0

extern int XtimeSetClockSource(
    Display *dpy, XtimeSchedulerID id,
    int clocksource,
    int offset, int scale);
```

Description.

This function associates the scheduler identified by `id` with the clock source identified by `clocksource`. A clock source is required to provide time information for a scheduler. Currently only one clock source is available, derived from the system timer, so the `clocksource` parameter must always equal `XTIME_DEFAULT_CLOCK`.

`scale` defines the time scale of the scheduler, indicated in ticks per second. It is to be interpreted as the divisor for the `when` and `expires` arguments to the `XtimeBegin` function discussed above. Generally, the zero value of the scheduler time scale is associated with the current time of the clock source; setting `offset` to some other value than zero allows to change this association.

Associating a scheduler with a clock source will generally implicitly start the scheduler, so some actions should be scheduled before this association is made.

Diagnostics.

The function may raise a `BadScheduler` error if `id` does not refer to the XID of a valid scheduler resource.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Freiberg, den 26.11.2002