Internet Engineering Task Force
INTERNET-DRAFT
draft-singer-rtp-qtfile-01.txt

Audio-Video Transport WG
D. Singer
Apple Computer, Inc.

October 22 1999
Expires : April 22 1999

Support for RTP in a stored QuickTime Movie File

Status of This Memo

Abstract

This document documents structures within a QuickTime movie file which permit easy transmission of the media content over RTP. This specification is intended to assist those who wish to stream stored movies over RTP, those wishing to prepare movies for streaming, and for those who might wish to record into QuickTime while preserving RTP information. The bit-stream(s) of RTP packets are normally compliant with the RTP payload definitions for their content, and full inter-operability can be achieved. Each QuickTime media track within a movie is sent over a separate RTP session and synchronized using standard RTP techniques. This specification builds on the

published QuickTime file format specification, and matches the hint track format used by the Darwin open-source streaming server.

1 Introduction

This document outlines how a set of sessions using the Realtime Transport Protocol (RTP) [1] may be transmitted by a server program by reading a QuickTime movie. RTP is a generic protocol designed to carry realtime media data along with synchronization information over a datagram protocol (mostly UDP over IP).

QuickTime files form the storage basis of the QuickTime media architecture; however, it is not necessary to use the QuickTime software to read, construct, or stream RTP from the files. The file format, without support for streaming or RTP, is fully described in the published specification [2].

The file format is capable of referring to media data in other files; this enables re-use of content. These other files need not be structured as QuickTime movies, and a number of 'foreign' formats can thus be streamed over RTP under this specification, provided that they can also be described by the QuickTime movie (i.e. described by the movie meta-data), and that the streaming server is willing and able to follow the links to these other files.

2 QuickTime File Format Overview

This section gives a brief overview of the file format. Readers wanting a detailed description are encouraged to refer to the published specification [2].

A fundamental underlying concept in the QuickTime file format is that the physical structure of the media data (the mapping of the media onto physical storage records) is independent of the logical structure of the media file. A QuickTime media composition is described by a set of "movie" meta-data; this meta-data provides declarative, structural/compositional, and temporal information about the actual media data.

The media data may be in the same file as the descriptive logical data (i.e., with the "movie" meta-data) or in separate files. A movie structured into one file is commonly called "flat" or "self-contained". Movies which are not self-contained may reference some or all of their media data in other files.

This separation between logical organization and physical organization makes the QuickTime file format ideally suited to optimization in different ways for different scenarios. When editing

and compositing, this means that media data need not be copied or re-coded as edits are applied and media is re-ordered; the meta-data file may be extended and temporal mapping information adjusted. When editing is completed, the relevant media data and meta-data may be rewritten into a single, interleaved, optimized file for efficient local or network access. However, both the structured and the optimized files are valid QuickTime files, and both may be inspected, played, streamed, and reworked.

The use of movies which are not self-contained enables the same basic media data to be used and re-used in any number of presentations. This same advantage applies when serving, as will be seen below.

In both editing and streaming, this also permits any number of other files to be treated as part of a presentation without copying the media data which they contain. Editing can change and re-write just the meta-data in the movie file, which is much quicker than reading and re-writing all the media data..

The QuickTime file is divided into a set of objects, called atoms. Each object starts with an atom header, which declares its size and type:

```
class Atom {
        int(32) size;
        char            type[4];
        int(8)  contents[];
}
```

The size is a 32-bit integer, in bytes, including the size and type header fields. There is also provision for 64-bit size fields. The type field is four characters (usually printable), to permit easy documentation and identification. The data in an object after the type field may be fields, a sequence of contained objects, or both. All field data are stored in big-endian format.

A QuickTime file consists of a sequence of objects. The two highest-level objects are the media-data (mdat) and the meta-data (moov) atoms.

The media-data object(s) contain the actual media (for example, sequences of sound samples or video frames). Their format is not constrained by the file format; they are not usually objects. Their format is described in the meta-data, not by any declarations physically contiguous with them. So, for example, in a movie consisting solely of motion-JPEG, JPEG frames are stored contiguously

in the media data with no required intervening extra headers. The media data within the media data objects is logically divided into chunks; however, there are no explicit chunk markers.

When the QuickTime file references media data in other files, it is not required that these 'secondary' files be formatted to this specification, since these media data files are formatted as if they were the contents of a media object. Since the format here does not require any headers or other information physically contiguous with the media data, it is possible for the media data to be files which contain 'foreign' headers (e.g. UNIX ".au" files, or AVI files) and for the QuickTime meta-data to contain the appropriate declarative information and reference the media data in the 'foreign' file. In this way the file format can be used to update, without copying, existing bodies of material in disparate formats. Thus editing and serving may be done directly from these files, greatly extending their utility. The QuickTime file format is a true unifying concept; it is both an established format and is able to work with, include, and thereby bring forward, other established formats. (The full range of supported file types is large; consult the QuickTime web site <http://www.apple.com/quicktime> for more information.).

Free space (e.g. deleted by an editing operation) can also be described by an object at this level. Any software reading the file should ignore free space objects, and objects at any level which it does not understand; this permits extension of the file at any level by introducing new objects. The primary meta-data is the movie object. A QuickTime file normally has exactly one movie object; it is typically at the beginning or end of the file, to permit its easy location (although this is not required).

The movie header provides basic information about the overall presentation (its creation date, overall timescale, and so on). In the sequence of contained objects there would normally be at least one track, which describes temporally presented data. A track is a media stream.

The track header provides basic information about the track (its ID, timescale, and so on). Information at the track level is independent of the media type contained in the track. Objects contained in the track might be references to other tracks (e.g. for complex compositing), or edit lists. In this sequence of contained objects there would normally be a media object, which describes the media which is presented when the track is played.

The media object contains declarations of the exact presentation required by the track (e.g. that it is sampled audio, or MIDI, or orientation information for a 3D Scene). The type of track is

declared by its handler.

Within the media information there is likewise a handler declaration for the data handler (which fetches media data), and a data information declaration. This defines which files contain the media data for this track; it is by using this declaration that movies may be built which span several files. At the lowest level, a sample table is used which relates the temporal aspect of the track to the data stored in the file:

```
class sampletable {
        int(32) size;
        char            type[4] = 'stbl';
        sampledescription       sd;
        timetosample            tts;
        syncsampletable         syncs;
        sampletochunk           stoc;
        samplesize                      ssize;
        chunkoffset             coffset;
}
```

The sample description contains information about the media (e.g. the compression formats used in video). The time-to-sample table relates time in the track, to the sample (by index) which should be displayed at that time. The sync sample table declares which of these are sync (key) samples, not dependent on other samples.

The sample-to-chunk object declares how to find the media data for a given sample, and its description given its index.

The sample size table gives the size of each sample; and the chunk offset table gives the offset into the containing file of the start of each chunk. The chunk offset table can contain 32-bit or 64-bit file offsets for chunks, permitting the use of very large files.

Walking this structure to find the appropriate data to display for a given time is straightforward, mostly involving indexing and adding. Using the sync table it is also possible then to back-up to the preceding sync sample, and roll forward 'silently' accumulating deltas to the desired starting point. Note that these tables which give sample timing, size, and position information, are constructed in such a way that they are naturally compact.

3 Support for streaming protocols

The QuickTime file format supports streaming of media data over a network as well as local playback. The process of sending protocol data units is time-based, just like the display of time-based data, and is therefore suitably described by a time-based format. A QuickTime file or 'movie' which supports streaming includes information about the data units to stream. This information is included in additional tracks of the movie called "hint" tracks.

Hint tracks contain instructions for a streaming server which assist in the formation of packets. These instructions may contain immediate data for the server to send (e.g. header information) or reference segments of the media data. These instructions are encoded in the QuickTime file in the same way that editing or presentation information is encoded in a QuickTime file for local playback. Instead of editing or presentation information, information is provided which allows a server to packetize the media data in a manner suitable for streaming using a specific network transport.

The same media data is used in a QuickTime file which contains hints, whether it is for local playback, or streaming over a number of different transport types. Separate 'hint' tracks for different transport types may be included within the same file and the media will play over all such transport types without making any additional copies of the media itself. In addition, existing media can be easily made streamable by the addition of appropriate hint tracks for specific transports. The media data itself need not be recast or reformatted in any way.

This approach to streaming is more space efficient than an approach that requires that the media information be partitioned into the actual data units which will be transmitted for a given transport and media format. Under such an approach, local playback requires either re-assembling the media from the packets, or having two copies of the media-one for local playback and one for streaming. Similarly, streaming such media over multiple transports using this approach requires multiple copies of the media data for each transport. This is much less space efficient than hint tracks, unless the media data must be heavily transformed to be streamed (e.g., by the application of error-correcting coding techniques, or by encryption).

Support for streaming in the QuickTime file format is based upon the following three design parameters:

(1) The media data is represented as a set of network-independent standard QuickTime tracks, which may be played, edited, and so on, as normal;

(2) There is a common declaration and base structure for server hint

tracks; this common format is protocol independent, but contains the declarations of which protocol(s) are described in the server track(s);

(3) There is a specific design of the server hint tracks for each protocol which may be transmitted; all these designs use the same basic structure. For example, there may be designs for RTP (for the Internet) and MPEG-2 transport (for broadcast), or for new standard or vendor-specific protocols.

The resulting streams, sent by the servers under the direction of the hint tracks, need contain no trace of QuickTime information. This design does not require that QuickTime, or its structures or declaration style, be used either in the data on the wire or in the decoding station. For example, a QuickTime file using H.261 video and DVI audio, streamed under RTP, results in a packet stream which is fully compliant with the IETF specifications for packing those codings into RTP.

The hint tracks are built and flagged so that when the presentation is viewed directly (not streamed), they are ignored.

3.1 RTP Hint Tracks

The RTP specification recommends sending each media stream as a separate RTP stream; multiplexing is achieved by using IP's port-level multiplexing, not by interleaving the data from multiple streams into a single RTP session. However, MPEG specifications do define methods to multiplex several media tracks into one RTP track, and this may be necessary in some applications. Each hint track is therefore tied, not to one, but a set of media tracks by track references. The set of references form a table, which is indexed by the samples (see below) when selecting data from the media tracks. This makes either multiplexing scheme possible.

This design decides the packet size at the time the hint track is created; therefore, in the sample description for the hint track (a data structure which can contain fields specific to the 'coding' - which in this case is a protocol), we indicate the chosen packet size. Note that it is valid for there to be several RTP hint tracks for each media track, with different packet size choices. Other protocols can be parameterized in a similar way. Similarly the time-scale for the RTP clock is provided in the sample description.

3.1.1 Sample Description Format

In the file format, each track has a description of its contents; for hint tracks, this description defines and parameterizes the protocol.

RTP hint tracks are hint tracks (media handler 'hint'), with an entry-format in the sample description of 'rtp '

```
aligned(8) class RtpSampleEntry extends SampleEntry('rtp ') {
     unsigned int(32) timescale;
     unsigned int(16) rtphinttrackversion = 1;
     unsigned int(16) rtplastcompatibleversion = 1;
     unsigned int(32) maxpacketsize;
     rtptags[] rtpdata;
}

aligned(8) class rtptag(tagtype) {
     unsigned int(32) size;
     unsigned int(32) type = tagtype;
}

aligned8) class timescaletag extends rtptag('tims') {
     unsigned int(32)    timescale;
}

aligned8) class timestampoffsettag extends rtptag('tsro') {
     unsigned int(32)    timeoffset;
}

aligned8) class sequenceoffsettag extends rtptag('snro') {
     unsigned int(32)    sequenceoffset;
}
```

The semantics of these fields are as follows: rtphinttrackversion is the version of this hint track; this document is version 1 rtplastcompatibleversion is the version of the oldest compatible reader that should be able to read this hint track maxpacketsize is the size, in bytes, of the largest packet this track will form rtpdata is a series of rtptags, to fill the rest of the atom, selected from the subclasses of rtptag timescale is an obligatory tag; it is the rtptimescale that was used to form this hint track timeoffset and sequenceoffset are optional; they indicate that the server should use these fixed offsets for these fields in the RTP packets, instead of truly random numbers

3.1.2 Declarative and Session Description data

To aid servers which use the SDP format, the hint tracks contain base data which can be used in assembling a complete SDP description. This data is stored in hint-information ('hnti') atoms within user-data ('udta') atoms in the movie atom, or in each track. In the movie, the hnti atom has a sub-atom of type 'rtp ' and starts with

'sdp ' (note the spaces). Within RTP hint tracks, the sub-atom has the type 'sdp ' (again, note the space). The contents in either case is ASCII text, suitable for forming into complete SDP descriptions. The server will need to generate a number of the lines of the SDP; the data supplied here is only partial, limited to that known at hinting time. There is also an optional user-data atom giving overall information about the hint track.

```
aligned(8) class hintinformation extends Atom('hinf') {
     infotags[] infodata;
}

aligned(8) class infotag(tagtype) {
     unsigned int(32) size;
     unsigned int(32) type = tagtype;
}
```

The following information tags and values are defined. They are all optional, and unrecognized tags should be ignored.

tag	value field type	value
trpy	unsigned int(64)	total bytes that will be sent, including RTP headers, but not other headers outside that (e.g UDP, IP or link layer headers)
nump	unsigned int(64)	total number of packets sent
tpyl	unsigned int(64)	total bytes that will be sent, not including RTP headers
maxr	unsigned int(32)[2]	maximum data rate. two values, granularity (in milliseconds), and m, the maximum data transmitted in any interval of that duration. There may be multiple maxr tags.
dmed	unsigned int(64)	total bytes copied by reference from media tracks
dimm	unsigned int(64)	total bytes sent as immediate data from the hint track
drep	unsigned int(64)	total bytes of repeated data that will be sent
tmin	unsigned int(32)	smallest relative transmission time, in milliseconds
tmax	unsigned int(32)	largest relative transmission time, in milliseconds
pmax	unsigned int(32)	largest packet sent, including RTP header
dmax	unsigned int(32)	largest packet duration, in milliseconds

payt	unsigned int(32), string	the payload type, followed by a counted string of the rtpmap information

3.1.3 RTP Sample Format

Each sample in the RTP hint track contains the instructions to send out a set of packets which must be transmitted at a given time. The time in the hint track is transmission time, not necessarily the media time of the associated media.

Notice that we now describe the internal structure of samples, which are media data, not meta data, in the terminology of this proposal. These need not be structured as objects.

Each sample contains two areas: the instructions to compose the packets, and any extra data needed when sending those packets (e.g. an encrypted version of the media data).

```
aligned(8) class RTPsample {
        unsigned int(16)        packetcount;
        unsigned int(16)        reserved;
        RTPpacket       packets[packetcount];
        byte            extradata[];
}
```

Each RTP packet contains the information to send a single packet. In order to separate media time from transmission time, an RTP time stamp is specifically included, along with data needed to form the RTP header. Other header information is supplied; the algorithms for forming the RTP header given the information here are simple. Then there is a table of construction entries:

```
aligned(8) class RTPpacket {
        signed int(32)  relative-time;
        // the next fields form initialization for the RTP
        // header (16 bits), and the bit positions correspond
        bit(2)  reserved;
        bit(1)  P-bit;
        bit(1)  X-bit;
        bit(4)  reserved;
        bit(1)  M-bit;
        bit(7)  payload-type;

        unsigned int(16)        RTPsequenceseed;
        unsigned int(13)        flags;
        unsigned int(1) x-flag;
        unsigned int(1) b-flag;
        unsigned int(1) r-flag;
        unsigned int(16)        entrycount;
        dataentry       constructors[entrycount];
        if (x-flag) {
                unsigned int(32)        extra-information-size;
                TLV     tlventries[];
        }
}

aligned(32) class TLV {
        unsigned int(32) tlvsize;
        unsigned int(32) tlvtype;
        unsigned int(8) tlvdata;
}
```

The relative-time field is a signed value in the hint track's timescale, adjusting the transmission time of the packet away from the RTP sample time. This allows the hinter to smooth the data rate of the transmitted packets.

The x-flag indicates that there is extra information after the constructors, in the form of TLVentries. Only one such entry is currently defined; tlvtype = 'rtpo' gives a 32-bit signed integer offset to the actual RTP time-stamp to place in the packet. This enables packets to be placed in the hint track in decoding order, but have their presentation time-stamp in the transmitted packet be in a different order. Note that all TLVentries are defined to be 32-bit aligned, and therefore their length should be padded to a 4-byte boundary; the only existing entry has a length of 4 bytes, so this is not currently an issue.

The b-flag indicates a disposable 'b-frame'. The r-flag indicates a 'repeat packet', one that is sent as a duplicate of a previous packet. Servers may wish to optimize handling of these packets.

There are various forms of the constructor. Each constructor is 16 bytes, to make iteration easier. The first byte is a union discriminator:

```
        aligned(8) class RTPconstructor(type) {
                unsigned int(8) constructor-type = type;

        }

        aligned(8) class RTPnoopconstructor
                extends RTPconstructor(0)
        {
                unsigned int(8) pad[15];                // 15 bytes
ignored
        }

        aligned(8) class RTPimmediateconstructor
                extends RTPconstructor(1)
        {
                unsigned int(8) count;
                unsigned int(8) data[count];
                unsigned int(8) pad[14-count];
        }

        aligned(8) class RTPsampleconstructor
                extends RTPconstructor(2)
        {
                unsigned int(8) trackrefindex;
                unsigned int(16)        length;
                unsigned int(32)        samplenumber;
                unsigned int(32)        sampleoffset;
                unsigned int(16)        bytesperblock = 1;
                unsigned int(16)        samplesperblock = 1;
        }

        aligned(8) class RTPsampledescriptionconstructor
                extends RTPconstructor(3)
        {
                unsigned int(8) trackrefindex;
                unsigned int(16)        length;
                unsigned int(32)        sampledescriptionindex;
                unsigned int(32)        descriptionoffset;
        }
```

The immediate mode permits the insertion of payload-specific headers (e.g. the RTP H.261 header). For hint tracks where the media is sent unchanged, the sample entry then specifies the bytes to copy from the media track, by giving the sample number, data offset, and length to copy. For complex cases (e.g. encryption or forward error correction), the transformed data would be placed into the hint samples, and then hintsample mode would be used. Note that this would be from the extradata field in the RTPsample itself.

The bytesperblock and samplesperblock concern compressed audio. This allows translation of the samplenumber into an actual byte offset in the audio track. The sampledescription mode allows sending of (portions of) sample descriptions as part of an RTP packet.

Note that these structures should be flexible enough to cover not only the standard RTP payloads (H.261, MPEG, etc.) but also private packings such as the QuickTime-in-RTP [3], or generic packing as is now being proposed [4].

Notice that there is no requirement that successive packets transmit successive bytes from the media stream. For example, to conform with RTP-standard packing of H.261, it is sometimes required that a byte be sent at the end of one packet and also at the beginning of the next (when a macroblock boundary falls within a byte). Conversely, payload packings that interleave the data to achieve error resilience will skip some bytes, to send them in another packet.

Note that it is possible, and legal, to copy all data into the hint track, and use sample constructors with a trackrefindex of -1 uniformly. These will be simpler to interpret for the server, but the file will be larger.

Acknowledgments

The author would like to thank a number of people, particularly Peter Hoddie (Apple Computer), William Belknap (IBM Corporation), Christopher Walton (Netscape), Dave Pawson (Oracle), Ronald Jacoby (Silicon Graphics, Inc.), and Gerard Fernando and Michael Speer (Sun Microsystems).

References

[1] H. Schulzrinne, et. al., "RTP : A Transport Protocol for Real-Time Applications", IETF RFC 1889, January 1996.

[2] Apple Computer, Inc., "QuickTime File Format Specification", May 1996.
<ftp://ftp.apple.com/Quicktime/devworld/QuickTime/mac/QuickTime.pdf>.

Expires : April 22 1999

Author's Contact Information
David Singer
Email: singer@apple.com
Tel: (408) 974 3162

Apple Computer, Inc.
One Infinite Loop, MS:302-3MT
Cupertino CA 95014
USA