Multimedia Systems (2007) 13:51–68
DOI 10.1007/s00530-007-0077-x

REGULAR PAPER

Metadata-driven optimal transcoding in a multimedia proxy

Laszlo Böszörmenyi · Hermann Hellwagner · Peter Schojer

Published online: 8 March 2007

Abstract An *adaptive* multimedia proxy is presented which provides (1) caching, (2) filtering, and (3) media gateway functionalities. The proxy can perform media adaptation on its own, either relying on layered coding or using transcoding mainly in the decompressed domain. A cost model is presented which incorporates user requirements, terminal capabilities, and video variations in one formula. Based on this model, the proxy acts as a general *broker* of different user requirements and of different video variations. This is a first step towards *What You Need is What You Get* (WYNIWYG) video services, which deliver videos to users in exactly the quality they need and are willing to pay for. The MPEG-7 and MPEG-21 standards enable this in an interoperable way. A detailed evaluation based on a series of simulation runs is provided.

Keywords Video proxy · Video caching · Media gateway · Media adaptation · Metadata · MPEG-7 · MPEG-21 · Cache replacement

1 Introduction

It is well-known that client-side proxies can give substantial support for video delivery over the Internet. Traditionally, such proxies provide two basic functionalities: they serve (1) as a *firewall* and (2) as a *cache*.

These basic functionalities can be considerably extended if we take into consideration that video delivery is getting ever more challenging, partly due to the heterogeneity in user requirements, partly also due to greatly diverse equipment, characterized by (1) different connectivity (ranging from high speed LANs over UMTS to slow connections over modem or GSM) and (2) different computational power (ranging from workstations over PDAs to cell phones). In such a heterogeneous environment, the proxy can take over a much more general role than usually, by serving different user and terminal types by different video quality classes. It can act as a kind of *media broker* that (1) understands the *preferences* and *capabilities* of the user and (2) can handle different *variants* of the same video. Based on these inputs, it can perform an optimal *match* between the needs of the user and the possibilities of the provider. It can further detect when a request cannot be fulfilled at all due to irreconcilable quality mismatch, and act in this case as a *request filter* that protects both server and client from attempting to serve "hopeless" requests (such as sending a video in HDTV quality to a PDA).

The *MPEG-7* standard provides tools to describe different variants of a video and the emerging *MPEG-21* standard provides tools to describe user preferences as well as terminal and network capabilities in an *interoperable* way. This enables us to build *What You Need is What You Get* (WYNIWYG) video services. The users do not get just the available quality nor the best possible quality, but exactly the quality they need and are ready to pay for.

This paper introduces a novel concept for such a proxy with broker functionality. The proxy may cache videos in different quality variants. If a video is getting popular both in high and in low quality, then it caches both variants. Thus, it acts not only as a cache but also as a *media gateway*.

L. Böszörmenyi (✉) · H. Hellwagner · P. Schojer
Department of Information Technology,
Klagenfurt University, Klagenfurt, Austria
e-mail: laszlo@itec.uni-klu.ac.at

Lower quality variants may be produced in a very efficient way due to layered coding. In the MPEG standardization group, great efforts are in progress to define an efficient layered video coding scheme. Currently, however, layered coding (as defined in MPEG-4) is not supported by virtually any codec. If the video is not available in layered coding, then the proxy can perform *transcoding* on its own. Transcoding may be a time consuming process, therefore the proxy has to consider its costs.

We introduce a *cost model* that controls the decisions of the proxy. To the best of our knowledge, this is the first work on a quality aware video cache that combines partial caching in the quality domain with a differentiated model of user preferences, of video variations, and of the caching costs. A nice feature of this approach is that we get the gateway and a basic filtering functionality dynamically as a "side effect" for free.

The presented concepts are implemented in QBIX-G (Quality Based Intelligent proXy Gateway), relying on previous work of the authors [12]. In this article, we present the architecture of the whole system, the underlying cost and quality model and a set of results based on a combination of the running implementation and simulation.

2 Basic notions

2.1 Client-side proxy cache

A proxy cache is a computer residing between client and server and caching data which the client is requesting from the server. A *client-side proxy* (in the following, simply proxy) has normally a better (faster) connection to its clients than to its servers, which are not aware of its existence. A client sends its request to the proxy, which tries to fulfill the request from its local cache, if possible. If not, the proxy forwards the request to the server. It stores a local copy of the data sent back by the server as a reply to the client. Ideally, a client-side proxy reduces load and network traffic on the server side and gives clients a reduced startup delay.

2.2 Media gateway

A media gateway is similar to a proxy in inspecting the data flows between server and client but it may also modify them according to some transcoding rules (see Fig. 1 for a typical scenario). The transcoding itself can be hinted by metadata, such as user preferences or terminal capabilities, or it can be hard coded. A media gateway node transcodes videos to some specific quality

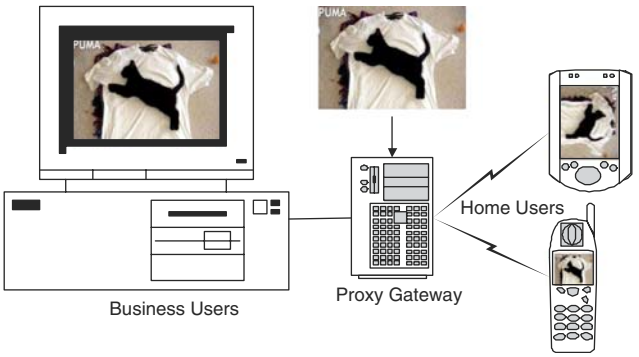


Fig. 1 Media gateway scenario

characteristics, such as a given dimension, bitrate, color etc. Note that transcoding converts a higher quality level to a lower one, but not the other way round.

2.3 Media adaptation

In the context of video transmission, media adaptation means the transformation of an already compressed video stream. Media adaptation can be classified into three major categories: (1) bitrate conversion or scaling (including frame dropping, i.e., temporal conversion), (2) resolution or spatial conversion, and (3) syntax conversion. Bitrate scaling can adapt to shortages in available bandwidth. Resolution conversion can adapt to bandwidth limitations, but it can also accommodate for known limitations in the user device in processing power, memory, or display size. Syntax conversion is used in hybrid networks to match server and client compression protocols. In the current work we assume that frame dropping can be done in the *compressed domain*, for other kinds of transcoding we need to decompress and re-encode a video, which is called adaptation in the *decompressed domain* — a slow and resource-intensive task. Efficient algorithms for general transcoding in the *compressed domain* do exist, but are out of the scope of this paper.

2.3.1 Adaptation and cache replacement

Cache replacement strategies in the area of video caching are divided into two categories: *full* and *partial caching*.

With *full caching*, videos are handled like normal Web objects, with the disadvantage that videos are huge and only a small number of videos can be cached at one node; thus, hit rate is low. With *partial caching*, only a selected part of a video is cached, e.g., only a prefix [13], or bursty parts of a video [16], hotspot segments [3], etc.

In this paper we concentrate on partial caching in the quality domain. Related work in this area mostly relies on layered coded videos, which reduces adaptation to the simple case of deleting the highest available enhancement layer. Examples are periodic caching of layered coded videos [4], combination of replacement strategies and layered coded videos [6], quality adjusted caching of GoPs (groups of pictures) [10], adaptive caching of layered coded videos in combination with congestion control [8] or simple replacement strategies (patterns) for videos consisting of different quality steps [7]. Most of these proposals rely on simulation to evaluate the performance of the caching techniques.

None of these proposals considers user preferences or reload behavior due to quality mismatches.

2.3.2 Adaptation and codecs

Most codecs do not support layered coding, although this is a requirement for fast and efficient adaptation. One of the first widely used standards with rudimentary adaptation support was MPEG-2, which allowed the definition of a single enhancement layer. This feature was pretty much ignored by content providers.

MPEG-4 is actually the first standard that offers extensive adaptation options, i.e., temporal, spatial and bitrate scalability through the means of layered coding. Most implementations of the standard in software and hardware do not (yet) support this feature, but restrict themselves to the *simple profile* part of MPEG-4, which does not even support B-frames.

2.3.3 Adaptation and media gateways

Using adaptation in media gateways on content that does not support layered coding creates several problems. The first problem is the high burden on the CPU created by resource intensive decoding and encoding operations. For example a 2.0 GHz Pentium IV processor is capable of performing bitrate transcoding on only six CIF (352×288) streams in parallel [11].

Another problem is the reduced hit rate in the cache. It is a common assumption that request patterns follow the Zipf distribution. The higher the Zipf α value, the higher is the hit rate in the proxy.[1] Without layered coding the proxy stores n variations of the same video in the cache. Due to differing user preferences, the requests now do not accumulate on a single object but are distributed over n variants. This "scattering" disturbs the original Zipf distribution and has the effect as if the α value was reduced. The number of one-timers (videos that are requested only once) increases and even a class of *zero-timers* is introduced. Zero-timers are videos that are used only as transcoding sources but are never explicitly requested by any client. As we have to lock these videos when using them as transcoding sources, they may remain in the cache for a fairly long time.

Moreover, the size of these n variants is in total greater than the size of the stream in layered coding format. Thus, a media gateway can store more video objects than a Web proxy but fewer *different* videos.

The advantage of the gateway functionality is that reloading of a video due to quality mismatches happens significantly less frequently and that customer satisfaction should be considerably higher. Costumer satisfaction is of course generally hard to measure — that is the reason why we take user preferences and costs into consideration.

2.4 User preferences

There are currently two major standards available for communicating user preferences to a server. The first one is CC/PP (Composite Capabilities/Preference Profiles) [9] which is a standardized framework developed by the W3C as an extension to the HTTP 1.1 standard. It is a collection of the capabilities and preferences associated with a user and the configuration of hardware, software, and applications used by the user to access the World Wide Web. The disadvantage of this protocol is that it fails to allow users to specify priorities for features, e.g., to prefer higher video frame rate over larger spatial resolution.

The other major standard is MPEG-21 [1], specifically the Digital Item Adaptation (DIA) part. The advantage of MPEG-21 DIA is that it was designed with content adaptation in mind. The goal of MPEG-21 is to define a normative open framework for multimedia delivery and consumption, spanning the whole delivery and consumption chain for a multitude of client devices differing in processing power, display, and bandwidth. The availability of this information helps a proxy/server to decide on optimal client-oriented adaptation. To define what a proxy/server/client is allowed to do with a media stream, e.g., playing, transcoding, etc., the Rights Expression Language (REL [2]) part was created. The schema for metadata describing adaptation and its usage is defined in the DIA [14] part, which is essential for a transcoding capable proxy gateway.

Digital Item Adaptation. In MPEG-21 a Digital Item (DI) is defined as a container for all types of different

[1] Higher α means more skewed popularity distribution.

media assets. This can be a Web page, a complete video or just one elementary stream from a video. To allow clients to transparently access DIs without having to worry about network and terminal installation or management, DIA was specified [14]. Using input provided by DIA tools, a Digital Item is transformed into an Adapted Digital Item.

The DIA tools consist of three major parts:

- Usage Environment Descriptions (UEDs)
  These include user characteristics, terminal capabilities, network characteristics, and natural environment characteristics. They are used to provide descriptive information about the usage environment of the users which in turn is used to adapt Digital Items to meet the requirements of the user for transmission, storage and consumption of these DIs.
- Digital Item Resource Adaptation (DIRA)
  A DI consists of resources, e.g. text, images, video streams. For each of these resources a *Bitstream Syntax Description* (BSD) exists. Using such a description, a Digital Item resource adaptation engine can transform the bitstream and the corresponding description using editing-style operations such as data truncation and simple modifications.
- Digital Item Declaration Adaptation (DIDA)
  The description of a DI is defined in an XML schema file, the DI Description Language (DIDL). Whereas BSD is used to adapt resources, DIDA defines tools to adapt DIs based on DIDL.

User Characteristics. User Characteristics describe user specific preferences like display preferences (e.g., preferred contrast, saturation, color profile) or audio preferences (e.g., audible frequency range). A user can also specify how important each of these parts is for him/her.
Terminal Capabilities. Terminal Capabilities are used to describe the hardware and software that exists on the client side. In terms of software, it is described which encoding/decoding capabilities a client has, e.g., whether it can decode MPEG-4 elementary streams containing B-frames.

3 QBIX-G

QBIX-G (Quality Based Intelligent proXy Gateway) realizes the combined media gateway/cache functionality. It supports standard-compliant RTSP communication, with extensions that allow clients to transmit their user preferences to the proxy, and allows real-time transcoding of AVI and MPEG-1/-2/-4 videos to the MPEG-4 format. The implementation of QBIX-G is based on the *ViTooKi* video toolkit, which was developed at our department. A detailed description [11] and the entire source code are freely available.[2] Here we present only the most essential parts.

3.1 Media caching and access scenarios

From the point of view of caching, the operation of QBIX-G can be described as follows. The client sends an RTSP DESCRIBE request containing the URL of the requested video and the user preferences of the client. The proxy checks in its cache whether it can find a version that matches the user preferences. Four different scenarios can now occur:

1. *Object miss with quality miss*: A full miss is given when the proxy either finds a version with a quality lower than acceptable (quality is acceptable if it is in the range specified by the client, see Sect. 4.2) or if it does not find any entry for the given URL. In this case, the proxy has to forward the request to the server. The server's reply needs to be adapted to the user preferences as specified by the client. (If the server does not support transcoding at all, the proxy has to remove the user preferences from the DESCRIBE.)
2. *Object miss with quality hit*: This situation is similar to the first one, except that the video coming from the server matches the user requirements, either because the server did the transcoding, or because the original version happens to have the required properties.
3. *Object hit with quality miss*: The proxy finds a cached version of the video but the quality of the object is higher than requested by, or acceptable for, the client and therefore transcoding is needed.
4. *Object hit with quality hit*: The proxy finds a cached version where quality is within the ranges specified in the user preferences.

The proxy forwards the created/found version of the video to the client when receiving the RTSP SETUP and PLAY requests and tries to cache the original/transcoded version, if possible and if not yet cached.

3.2 QBIX-G architecture

Figure 2 presents the basic modules of the ViTooKi library. There are some common classes that are used

[2] http://vitooki.sourceforge.net/

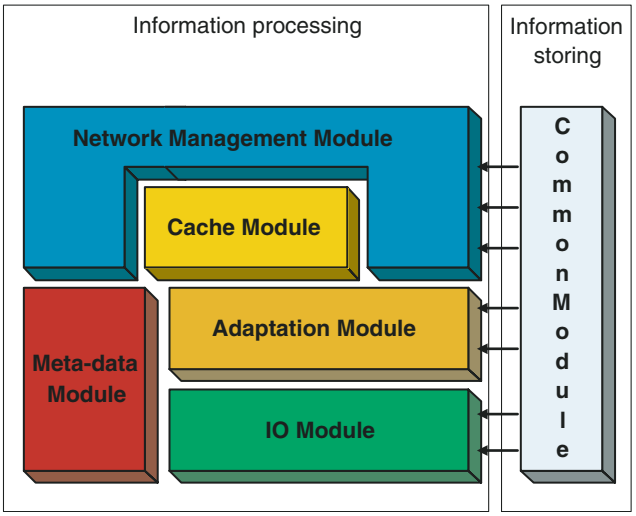


Fig. 2 QBIX-G architecture

throughout the whole library in all other modules. The *IO Module* is used for reading and writing data from/to files or the network. The *Adaptation Module* is used for data processing, i.e., media frame manipulation for adaptation purposes (see Sect. 3.3). ViTooKi's *Network Management Module* consists of classes used for RTSP message parsing and session management. The *Metadata Module* realizes support for MPEG-7 and MPEG-21 information. The *Cache Module* realizes several adaptive cache replacement strategies and supports admission control.

3.3 Frame manipulation

One of the strengths of ViTooKi is its ability to easily manipulate single frames via *Adaptor* classes. An adaptor expects as input a single frame and returns a list of adapted frames, which can contain zero or more frames. An adaptor is allowed to buffer frames, until it has enough data available to perform one adaptation step. Currently, only visual media adaptors are supported; system-level (e.g., scene) adaptation is not yet implemented, nor is audio processing.

Adaptors are used to decode, encode and manipulate frames. Gray-scaling, temporal frame rate reduction, or bit rate reduction are just a few examples for frame manipulation adaptors. In addition, adaptors are used for logging and gathering of statistic information.

The *AdaptorChain* class, which itself is a subclass of *Adaptor*, allows combining several adaptors into a list of adaptors, which are executed sequentially. Figure 3 shows an example. A compressed video frame is fed into the AdaptorChain. The AdaptorChain

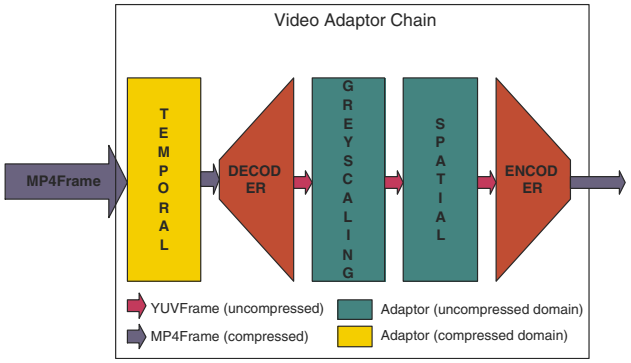


Fig. 3 Example of an *AdaptorChain*

forwards it to its first adaptor (in this case a *TemporalAdaptor*). The TemporalAdaptor acts as a B-frame filter which deletes B-frames and only returns I- and P-frames. The AdaptorChain forwards the returned frame to the next adaptor, which returns a decoded frame, which is then forwarded to the next adaptor, etc. A single invocation of the *Adapt* method of an AdaptorChain ends either when one of its adaptors returns no frame, or when the last adaptor of the AdaptorChain returns.

3.4 Metadata support

The metadata module of ViTooKi contains support for parsing and creating terminal capabilities according to MPEG-21 DIA and support for VariationSet descriptions as specified in MPEG-7. The player software can attach terminal capabilities to an RTSP request, thus indicating an adaptation request. If enough resources are available, the video stream will be transcoded in real-time according to this information.

3.4.1 RTSP

ViTooKi supports its own RTSP library. The main reason to write a new RTSP library from scratch was to allow non-standard conforming extensions that were needed for adaptation. Another reason was the lack of good RTSP libraries when the project started.

We decided to use MPEG-21 DIA instead of CC/PP because the scope of MPEG-21 DIA is much larger than the scope of CC/PP, e.g., allowing for session mobility to be integrated at a later point in time.

3.4.2 Extension to DESCRIBE

An extended DESCRIBE is used which includes an MPEG-21 DIA document containing the user's terminal

capabilities [15], as the following example shows:

```
DESCRIBE rtsp://192.168.0.2/coastguard.mp4 RTSP/1.0 CSeq: 1
Accept: application/sdp User-Agent: MPEG4ITEC Player
Content-Type: application/mpeg21_dia Content-Length: 458
<xml version="1.0" encoding="UTF-8"?>
<DIA xmlns="urn:mpeg:mpeg21:dia:schema:2003"
  xmlns:mpeg7="urn:mpeg:mpeg7:schema:2001"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:mpeg:mpeg21:dia:schema:2003">
  <Description xsi:type="UsageEnvironmentType">
    <Terminal>
      <InputOutput>
        <Display bitsPerPixel="8" colorCapable="false">
          <Resolution horizontal="240" vertical="320"/>
        </Display>
        <AudioOut numChannels="1"/>
      </InputOutput>
      <DeviceProperty>
        <Storage size="64.0" writable="false"/>
        <DeviceClass>PDA</DeviceClass>
      </DeviceProperty>
    </Terminal>
    <Network>
      <Capability maxCapacity="128000" minGuaranteed="32000"
        inSequenceDelivery="false" errorDelivery="true"/>
    </Network>
  </Description>
</DIA>
```

If no terminal capabilities are specified, adaptation is disabled for this client.

3.5 Stream processing

Elementary streams are processed via *DataChannels*. The main function of a DataChannel is to read, transform, and write data. Frames can be either read from the network or from a local file, are then passed to an *Adaptor* object for optional transformation, and the result is then written out to several *DataSinks*. A *DataSink* can further transform the frames via its local adaptor before writing it out to the network, a file, or displaying it on the screen. This approach allows one to do global adaptation in the *DataChannel* which affects all clients and to do a fine-tuned, client-specific adaptation in the *DataSink* class.

A sample DataChannel which consists of three clients is shown in Fig. 4. All clients require the video to be transcoded in the spatial dimension. *DataSink 1* is decoding the frame and using some *Display* class to show the video on the screen (a short-cut bypassing directly connected encoder-decoder steps can be applied as a local optimization). *DataSink 2* requires further adaptation and discards all B-frames of the video. The resulting stream is sent via RTP/UDP to the client. Client 3 does not need to change the adaptation result, the video stream is directly forwarded to it via RTP/UDP.

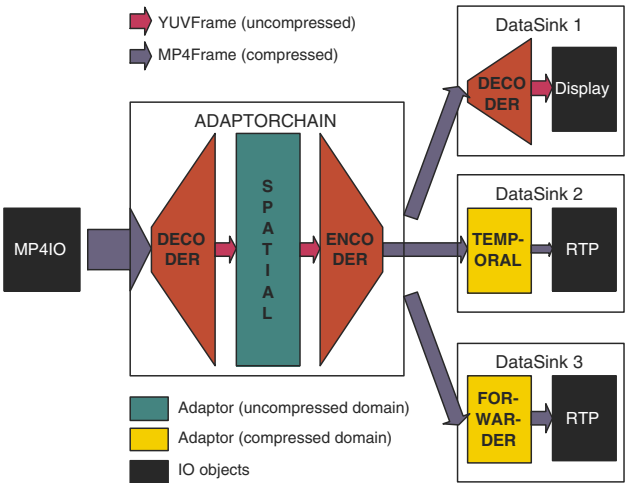


Fig. 4 Example of a *DataChannel*

3.6 Transcoding costs

Figure 5 shows CPU costs for all adaptors that can be used in transcoding a video. Decoder and encoder were tested on a Pentium IV running at 2 GHz with both the *XVID* and the *FFMPEG* codecs.[3] The test video used was the video "Alien", which is included in the "media" directory of ViTooKi. The operating system used was Linux (kernel 2.4) with gcc 3.2.

The percentage values presented in Fig. 5 are relative to the media stream duration (62 s) and specify how much time was spent in the specific adaptor. For example, a percentage value of 4% at the decoder means that $0.04 \times 62 = 2.48$ s of CPU time were spent for decoding the complete stream. Tests were repeated 10 times, average values are presented here, absolute time variation in the single test runs was small in the range of at most 70 ms. I/O usage was excluded from the benchmark.

One conclusion from Fig. 5 is that XVID is slower than FFMPEG, especially at encoding, and therefore only FFMPEG should be used for transcoding. In both codecs, B-frame support was switched off in the encoder (but was enabled in the decoder) and they were configured to use the fastest, but also lowest quality mode.

Another — obvious — conclusion is that temporal adaptation, which does not require decoding and encoding, is the cheapest solution in terms of CPU usage. The maximum number of temporal adaptors running in parallel is thus not limited by the CPU but by the speed of the network and hard disk. Unfortunately, the typical size of B-frames in a stream accumulates only to approximately 20% of the stream size, but discarding

[3] We compared XVID version 2002-09-28 with FFMPEG version 0.4.8. See also http://ffmpeg.sourceforge.net/.

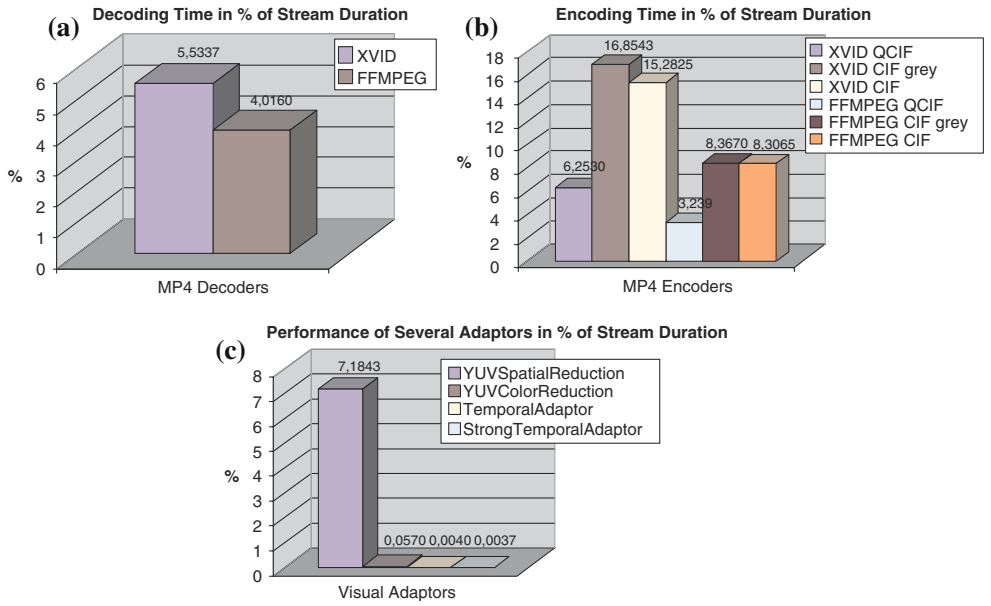


Fig. 5 Performance of selected adaptors

the B-frames at least halves the frame rate. Also the number of useful transcoding steps is limited to a single adaptation step.[4] Splitting this step up into several smaller ones does not make sense because then the relation between the reduction of stream size and the I/O overhead for the transcoding (bytes transferred from and to disk) is even worse.

Reducing the bit rate of a single CIF stream (352×288 pixel) involves costs of approximately 12.3% CPU time, allowing in theory up to eight transcoders to run in parallel in real-time. Practically, the number is lower (6), because I/O and an increasing number of threads generate additional overhead.

Gray-scaling a single stream consists of a decoder, a *YUVColorReduction* adaptor, and an encoder. Costs are nearly identical to the bit rate reduction case.

Resizing a stream is the most expensive operation. Changing the spatial dimension from CIF (352 × 288) to QCIF (176×144) requires 14.5% of CPU time to finish in real-time. As shown in Fig. 5, 7.1% were used by the *YUVSpatialReductionAdaptor* alone. Note that costs for a *YUVSpatialReductionAdaptor* increase linearly with the number of output pixels. In the worst case, when the output dimension is close to CIF, i.e., actually merely copying a stream (including a small size reduction), more than 28.2% CPU utilization were measured for the spatial reduction adaptor alone. Such a transcoding operation would require about $Cost(decoding)+Cost(spatial)+Cost(encoding) = 4.0+28.2+8.3 = 40.5\%$ CPU time.

Thus, depending on the requested transcoding operations, the number of parallel transcoding tasks varies between 2 and 6. Such a low number of maximum parallel transcoding steps is one of the reasons why a cost model must take these operations into consideration.

4 Calculating stream quality

When receiving a request, a proxy has to compare how "well" cached videos match a request. To be able to perform this calculation, we must first define the features characterizing elementary media streams.

4.1 Stream features

Let S be the set of elementary media streams requested by all clients. Each stream $s \in S$ is uniquely defined by a set of variable features F_s and a set of constant features C_s. F_s contains all features that a client is allowed to specify via a request. The set of constant features C_s contains features which are changeable neither by the proxy nor by the client. Depending on the stream type, different features are defined, as shown in Table 1.

For visual streams, *dimX* identifies the spatial resolution of a stream in the x dimension, *avgBitRate* specifies the average bit rate in bits per second, *color* can either

[4] We throw away all B-frames in one adaptation step. The next step, removing P-frames, results in an unacceptably low frame rate.

Table 1 Feature set definition

Stream type	F_s	C_s
Visual	{dimX, avgBitRate, color, frameRate}	{URL, aspectRatio, esID}
Audio	{samplingRate, numOfChannels, avgBitRate}	{URL, esID, language}
Other	{*avgBitRate*}	{*URL*, *esID*}

be *true* or *false*, and *frameRate* defines the rate in frames per second. The *URL* uniquely assigns an elementary stream to a video file, the *aspectRatio* is used to calculate *dimY*, the elementary stream id *esID* is used to uniquely identify a stream within a video file.

For audio streams, *numOfChannels* defines if a client wants no sound (=0), mono sound (=1), stereo (=2) or surround sound (>2). The feature *samplingRate* is expressed in Hertz with meaningful discrete values in the range [8000,48000]. *Language* is a string that defines the language of the audio stream; for music-only streams the language is set to *none*.

For all other streams, only *avgBitRate*, *URL* and *esID* are defined.

4.2 Request definition

A single request *r* to an elementary stream *s* consists of the URL, the elementary stream id and, for each single feature $f_i \in F_s$, an *acceptance range* $[min, best, max]$ and an associated *importance* value. The sum of importance values over all features f_i must sum up to 1: $\sum_{i=1}^{|F_s|} importance_{f_i} = 1$.

The client also may specify request specific parameters like the *maximum delay* (*maxDelay*) he/she is willing to wait for the service (in milliseconds). For commercial scenarios a client must also specify an upper limit of *money* he/she is willing to pay for the video. *maxDelay* and *money* define critical conditions, i.e., if they cannot be fulfilled the proxy has to return an error. We further define *user preferences* to include both the feature set F_s and the request specific parameters *maxDelay* and *money*. An example for a request looks like this:

- F_s ={dimX, avgBitRate, color, frameRate} = {[176,240,352]/**0.4**, [64000,128000,256000]/**0.5**, [0,1,1]/**0.0**, [15.0/15.0/25.0]/**0.1**}.
- C_s ={URL, aspectRatio, esID} = {rtsp://www.server.at/test.mp4, $\frac{4}{3}$, 2} and $maxDelay = 2000$ ms.

In this example, F_s's notation is extended by the importance values, expressing, e.g., that the client does not care about color (0.0), but that spatial resolution (0.4) and bit rate (0.5) are of great importance.

4.3 Quality formula

Matching a request to the feature set of the requested stream allows the proxy to calculate an abstract quality value that could be reached by forwarding a specific version of the requested stream to the client. This value tells, how "good" an answer to a given request is.

4.3.1 Feature quality function

Feature quality is calculated with the help of a utility function which takes as input a triple: the *acceptance range* and *importance* as specified by the client, and *val*, the actual feature value of the requested stream.

Figure 6 shows an actual implementation of a utility function used in QBIX-G. Intuitively, quality is highest when *val* is closest to the value considered optimal by the client (*best*). In this case the maximum quality value, which is given by *importance*, is returned. The quality value is zero if *val* is outside the $[min, max]$ acceptance range.

The *border* parameter, $0 < border < 1$, defines the percentage of the *importance* value that can be achieved at the edge points *min* and *max*. This parameter is used to distinguish between cases where the edge points of the range $[min, max]$ are inclusive or exclusive, respectively.

For simplicity, we assume that quality degrades linearly from *best* to *min* and *best* to *max*. The lowest value

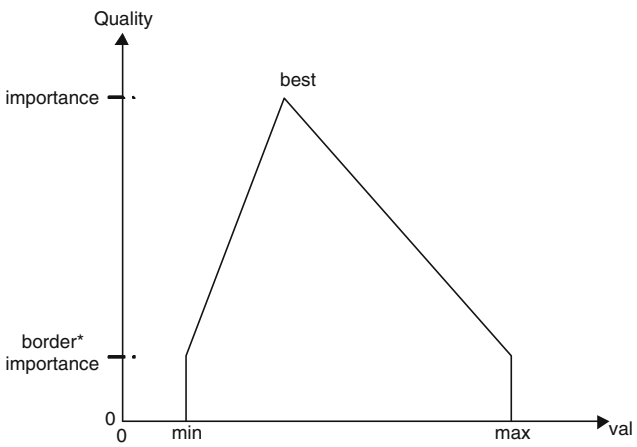


Fig. 6 Quality function for a single feature

is reached at the edge points *min* and *max* with a value of $border * importance$, the peak value is at the position *best* with the value *importance*.

4.3.2 Overall quality function

The overall quality of a stream s for a request r is defined as:

$$Quality(r) = \sum_{i=1}^{|F_s|} FeatureQuality(val_i, range_i, importance_i) \text{ or } 0 \text{ if } \exists j, 1 \leq j \leq |F_s| : FeatureQuality(val_j, range_j, importance_j) = 0$$

The maximum possible quality value is 1, which means that a perfect hit was found, a quality value of 0 is interpreted as "*Do not send! Client will reject the stream.*"

Note that the quality function does not state anything about visual quality. It is only a metric on how close the features of a specific stream are to the requested features. By specifying such ranges, the client explicitly states that he/she will not reject a video inside the range due to a quality mismatch.[5] If no feature ranges are specified, it is assumed that the client requests the video with the original quality.

5 Calculating stream costs

Being able to calculate the quality of an existing stream variation allows a proxy to act as a request filter, i.e., it can detect in advance when stream rejection is likely to occur. A typical use case is that of a mobile device requesting a visual stream which in the cached version exceeds its display size and its bandwidth. A non-adaptive, but metadata aware proxy cache will simply refuse to serve such a request to a client, yet an adaptive proxy cache will try to fit the stream to the request by means of adaptation. Due to the fact that adaptation can be rather expensive, the proxy must be able to calculate how much transcoding will cost.

For this purpose, a cost function was defined which calculates a cost value over all affected resources. (We do not consider explicitly the question of changing bandwidth during a streaming session. The suggested model does not exclude a renegotiation during a session, but this feature is not implemented currently.)

5.1 Resource definition and costs

QBIX-G regards network, hard disk, and CPU as relevant resources. Each resource is described by an upper limit (namely $limit_{net}$, $limit_{hd}$, $limit_{cpu}$), a current load ($load_{net}$, $load_{hd}$, $load_{cpu}$), and a price ($price_{net}$, $price_{hd}$, $price_{cpu}$). For the network the upper limit is expressed in Mbit/s, for the hard disk in MByte/s, and for the CPU the upper limit is defined as the number of video pixels the system can decode per second (as discussed in Sect. 5.1.1). The resource load is expressed as a value in the range [0,1], with 0 indicating that the resource is not used at all and 1 expressing 100% resource usage.

5.1.1 CPU costs

While the calculation of the load for the resources hard disk and network is rather simple, CPU load calculation is more complex. For this purpose, a multi-threaded benchmark tool is used. The benchmark simulates several sessions, each of them reading a video file from the disk and decoding it. The videos are in MPEG-4 Visual format and CIF resolution, the decoding software is based on the open source library *ffmpeg*. The overall execution time is measured and the CPU decoding capability is calculated. For a one-processor system (Pentium 4 @ 2 GHz), a $limit_{cpu}$ value of 33.6 Mpixels/s was measured. This interpolates to the system being able to decode $\frac{33.6 M pixel}{352 * 288} \approx 331$ CIF sized frames, or approximately 11 video streams having a frame rate of 30 fps each.

Transcoding costs. A transcoding operation A is defined as a function that maps the feature set F of a source stream *src* to an output stream *target*; the constant features C do not change:

$$A : F_{src} \rightarrow F_{target}; C_{target} = C_{src}. \quad (1)$$

We will define transcoding costs in relation to the costs of decoding. Hence, for each single transcoding algorithm, its costs relative to the decoding operation have to be determined.

Table 2 gives an overview of the transcoding operation costs which were derived from the benchmark results. The costs for decoding the *src* video depend directly on the spatial resolution and the frame rate, i.e. amount to $pixel_{src}/s = dimX_{src} * dimY_{src} * frameRate_{src}$.

Table 2 Approximated CPU costs for transcoding operations

Operation	Costs
Decoding	$pixel_{src}/s$
Encoding	$2 * pixel_{target}/s$
Spatial reduction	$3.72 * pixel_{target}/s$
Greyscaling	$0.008 * pixel_{src}/s$
B-frame dropping	0
B/P-frame dropping	0

[5] He/she might still reject the video for other reasons.

Costs for all other operations depend on the same features. Some operations depend on the features of the *src*, some on the features of the *target* video. For example, encoding is roughly twice as expensive as decoding, a spatial reduction nearly four times as expensive as decoding. The costs for temporal adaptation, i.e. B- or B/P-frame dropping, are negligible and set to zero.

Note that these values are only approximations of the real costs. For example, decoding costs also depend on the number of B-frames present in a video, on the amount of motion present in the scene, the codec used and its configuration. Also, operations can be concatenated. For example, in order to greyscale a stream in the decompressed domain, the stream must be first decoded, then greyscaled, and finally encoded again, as described in Sect. 3.6. The costs for the combined operation are $Decoding + Greyscaling + Encoding = pixel_{src}/s + 0.008 * pixel_{src}/s + 2 * pixel_{target}/s = 3.008 * pixel_{src}/s$ (due to $pixel_{src} = pixel_{target}$).

5.1.2 Example

Assume that the following system configuration is given. The upper network limit $limit_{net}$ is set to 5,000,000 bit/s, the upper CPU limit $limit_{cpu}$ to 30,000,000 pixel/s and the maximum hard disk speed $limit_{hd}$ is 10,000,000 byte/s (=80,000,000 bit/s). The features of the source stream are defined as follows:

F_{src}	C_{src}
dimX = 352 pixel	URL=rtsp://t.at/test.url
avgBitRate=200,000 bit/s	esID=2
color = 1	aspectRatio = 4/3
frameRate = 25 fps	

A client requests a version with the features $dimX_{target}$ = 176 pixels and $avgBitRate_{target}$ = 100,000 bit/s. This requires the proxy/server to decode the stream, re-size it and encode the result of the re-size operation with the new bit rate. We assume that the source is read from the disk and the transcoding result is written both to the network and to the disk cache.

Disk costs are calculated as reading 200,000 bit/s and writing back 100,000 bit/s. Thus, the increase of the disk load hd_r generated by the request r is

$$hd_r = \frac{avgBitRate_{src} + avgBitRate_{target}}{limit_{hd}} = 0.00375$$

Network costs are limited to sending the generated video to the requester. The increase of the load net_r is

$$net_r = \frac{avgBitRate_{target}}{limit_{net}} = 0.02$$

The CPU load consists of the sum of the load of all three operations involved in creating the target stream. The load increase cpu_r due to request r is

$$cpu_r = \frac{Decoding(f_{src}) + Spatial(f_{target}) + Encoding(f_{target})}{limit_{cpu}}$$
$$= \frac{(352 * 288 * 25.0) + (176 * 144 * 25.0) * 3.72 + (176 * 144 * 25.0) * 2.0}{30,000,000} = 0.205$$

5.2 Requirements on a cost function

The cost function should fulfill the following requirements: (1) The higher the actual load of a resource, the more expensive should the usage of that resource be. (2) It should act also as an admission control tool, i.e., requests which would exceed the limit of at least one of the resources should be rejected. (3) Cache hits should be preferred over cache misses. (4) It should be possible to assign weights to resources, according to their importance. (5) In a commercial scenario, requests from clients that require transcoding and do not pay enough, should be rejected. (6) It should try to maximize quality if enough resources are available.

5.3 Cost formula

The cost formula of QBIX-G distinguishes between several different types of costs. Resource costs occur at the participant that is servicing the request. In commercial scenarios, QBIX-G must consider content costs where content owners receive a fee for each request. In a commercial scenario, resource costs should also reflect the monetary costs that occur at the servicing side for a single request.

5.3.1 Resource costs

Let $load_{net}$ specify the actual load for the resource network, $load_{hd}$ for the hard disk, and $load_{cpu}$ for the processor. Each resource has assigned a price, namely $price_{net}$, $price_{cpu}$ and $price_{hd}$.

The first step in calculating costs is to calculate the additional resource load caused by a request r. As shown in Sect. 5.1.2, we calculate the additional load for the resources network (net_r), disk (hd_r), and CPU (cpu_r).

Resource costs depend directly on the current load of the resource and the assigned price. Thus, for a given request r the formula for overall resource costs is given as

$$\begin{aligned} ResourceCosts(r) = & \frac{net_r}{1 - futureLoad_{net}} * price_{net} \\ & + \frac{cpu_r}{1 - futureLoad_{cpu}} * price_{cpu} \\ & + \frac{hd_r}{1 - futureLoad_{hd}} * price_{hd} \\ & + \frac{startupDelay}{maxDelay}, \\ & \text{or } \infty \quad \text{if } futureLoad_{net} \geq 1.0 \\ & \qquad \text{or } futureLoad_{cpu} \geq 1.0 \\ & \qquad \text{or } futureLoad_{hd} \geq 1.0, \end{aligned}$$

where $futureLoad_{net} = load_{net} + net_r$, $futureLoad_{cpu} = load_{cpu} + cpu_r$ and $futureLoad_{hd} = load_{hd} + hd_r$. Note that, by including the actual load of the current resources, resource intensive operations are penalized if free resources are already sparse. By having $1 - futureLoad$ in the denominator, we ensure that the costs for using a resource will grow exponentially with increasing load. With a *futureLoad* of 1, costs will be infinite. Admission control functionality is achieved by the additional conditions which return infinite costs if for at least one resource $futureLoad \geq 1$. The inclusion of the startup delay increases the costs for a cache miss.

5.3.2 Billing costs

Billing costs are only calculated in a commercial scenario and consist of two parts. They include the resource costs, the content costs, and a certain profit for the service provider:

$$\begin{aligned} BillingCosts(r) = & ResourceCosts(r) \\ & + ContentCosts(r, quality) + profit_r. \end{aligned}$$

Note that in a commercial scenario, billing costs return the amount of money that a client is charged for consuming a video. Thus, the resource costs formula must return the monetary costs that occur at the servicing side for a single request. This can be achieved by setting $price_{net}$, $price_{hd}$, and $price_{cpu}$ accordingly.

If the billing costs exceed the amount of money a client is willing to pay, the request is rejected. Note that the content costs function has to be provided by the content owner. If no such function exists, the proxy must take full costs even for a lower quality version of the video. $Profit_r$ is the minimum profit the proxy generates by servicing a single request.

In a non-commercial scenario, content costs and profit are zero, thus billing costs are equal to resource costs.

5.4 Quality versus costs

The previous sections have shown how QBIX-G calculates costs and quality for a single request/stream-version pair. Still, a mechanism is needed which combines the quality measure with the cost value and returns weighted final costs. *FinalCosts* are calculated as

$$\begin{aligned} FinalCosts(r) = & (1 - Quality(r)) * ResourceCosts(r) \\ & \text{or } \infty \text{ if Quality(r)} = 0 \\ & \qquad \text{or ResourceCosts(r)} = \infty \end{aligned}$$

QBIX-G chooses the stream version with the lowest final costs and streams it to the client.

The final costs formula meets the requirement that it spends more resources if the outcome of a transcoding operation has a high quality, trying to maximize quality.

Another possibility to calculate *FinalCosts* is simply: $FinalCosts(r) = \frac{ResourceCosts(r)}{Quality(r)}$. This formula favors quality for individual streams as well and has the additional advantage of returning infinite costs if $Quality(r) = 0$. The disadvantage is that quality is overemphasized. Some few, expensive (high quality) operations could monopolize the system resulting in an overall higher rejection rate and thus impaired service for the clients. Due to the limited CPU power available in current systems, we choose to use the first formula which allows to service more concurrent requests.

A further requirement is that the cost formula should only have a minimal impact on the object hit rate of the proxy. This requirement is fulfilled by obeying the following *stream creation rules*.

5.4.1 Stream creation rules

Stream creation rules are used to reduce the number of possible variations generated in the proxy and to increase the chance for a future hit. For features, this means that transcoding is limited to specific discrete points.

For visual streams we define the following rules:

- Feature *dimX* must be a multiple of 44 or equal to the dimension of the source stream. *dimY* is calculated according to the aspect ratio of the original video. This rule generates the most commonly used resolutions like QCIF (176 × 144), CIF (352 × 288)

and 4CIF (704 × 576), thus increasing the chance for a quality hit.
- Average bit rate is defined as $dimX * dimY * b$ bit/s, $b \in \{1,2,4,8\}$. The value b stands for the number of bits used to code a pixel in a compressed stream. This is a rough (but practical) approximation of the compression effect, regardless of the frame rate.
- Frame rate should be a multiple of 5 or equal to the original frame rate, except if the video contains B-frames, because we never cache a version which was created by dropping B-frames.[6]
- Color can be only true or false.

For audio streams the following rules are used:

- The discrete points for the feature *samplingRate* are $\{11025 * i\} \bigcup \{8000 * j\}, i \in \{1,2,3,4\}, j \in \{1,...,6\}$. Higher sampling rates than 48kHz are deliberately not supported.
- The feature *numOfChannels* is limited to mono or stereo sound only. Creating surround sound in real-time is considered to be too expensive: $numOfChannels \in \{1,2\}$.
- The feature *avgBitRate* is defined as: $samplingRate * numOfChannels * r$ bit/s, $r \in \{1,2\}$.

5.4.2 Version selection algorithm

Let S be the set of elementary streams cached at a proxy, let V be the set of streams requested by all clients, $S \subseteq V$. Also assume that for each stream $s \in V$ the content provider has specified a set of eligible variations V'_s and a set of available adaptation steps A_s that can create these variations.

For a request r, the proxy searches all streams in S that have the same URL. For each stream version found, the proxy calculates its quality value. If a version is found that returns a quality value greater than zero, we have an object hit with quality hit (no transcoding is necessary). For each quality hit, costs are calculated and both costs and quality are combined to a final costs value. The version with the lowest final costs is considered best and streamed to the client.

In the other case, when no quality hit was found and transcoding is necessary, one has to generate a set of possible transcoding sources. For a request r this set is defined as $S_r = \{s \in S | s.url = r.url \wedge s.cached = true\}$. If $S_r = \{\}$, an object miss was encountered.

For each stream $s \in S_r$ a feature set F_s is defined. Transcoding is the process of mapping a feature set F to a feature set F'. An adaptor $a \in A$, with A being the set of adaptation steps the proxy supports, performs such a mapping $F \rightarrow F'$. The definition for a transcoding operation t thus contains an adaptor and the features of the source and target streams:

$$t : F \xrightarrow{a} F'$$

For each feature $f \in F_s$ the client specified a feature range with a minimum, a best, and a maximum value. For each feature, stream creation rules are known, which are used to transform a feature range to a set of discrete target transcoding points P. For example, for the feature $dimX$, only multiples of 44 are allowed, or the original resolution. If a client specified $dimX$ as $[150, 200, 300]$ and the spatial dimension of the original stream is outside the specified range, P_{dimX} will result in the set $\{176, 220, 264\}$.

In the visual case, we get sets for dimX (P_{dx}), bit rate (P_{br}), frame rate (P_{fps}), and color (P_{col}). Thus, the set of all transcoding possibilities T_s is defined for a single visual stream $s \in S_r$ with the properties $F_s = (dx_s, br_s, fps_s, col_s)$ as

$$T_s = \{(dx_s, br_s, fps_s, col_s) \xrightarrow{a} (dx, br, fps, c)\},$$
$$\forall\, dx \in P_{dx}, \forall\, br \in P_{br}, \forall\, fps \in P_{fps}, \forall\, c \in P_{col},$$

where $a \in A'_s$ and $A'_s = A \bigcap A_s$, i.e., a is an adaptor that is both supported by the proxy (A) and allowed by the content provider (A_s).[7] If for a mapping from a source F_s to a target F'_s ($F_s \rightarrow F'_s$) no suitable adaptor a is found (or allowed), the mapping is not included in T_s.

This step is repeated for all source files $s \in S_r$ which gives for a single request r the set of all eligible transcoding steps:

$$T_r = \bigcup T_s, \quad \forall s \in S_r.$$

$\forall t \in T_r$ the final costs are calculated. The transcoding steps yielding infinite final costs are removed from T_r.

Assuming $|T_r| > 0, \forall\, t \in T_r$, the $F_s \rightarrow F'_s$ pair with the minimum final costs is chosen and the video is adapted with the adaptor a associated with the mapping and streamed to the client in real-time.

If $T_r = \{\}$ after this step, either because of object miss or quality miss, the original video has to be fetched from the server. The proxy then repeats its calculation with the original video as the source. If T_r is still empty after this step, the request is rejected.

[6] It would be counterproductive to cache a version which can be generated on the fly so easily.

[7] An adaptor may be defined as a short, predefined sequence of elementary adaptation steps. Dynamic construction of adaptation chains is out of scope of this paper.

Table 3 User preference classes

DimX	Bit rate	Color	Frame rate
(144,172,200)/0.25	(48000,172999,297999)/0.25	false/0.25	(15.00,15.00,30.00)/0.25
(201,229,257)/0.25	(298000,422999,547999)/0.25	true/0.25	(15.00,20.00,30.00)/0.25
(258,286,314)/0.25	(548000,672999,797999)/0.25	true/0.25	(15.00,25.00,30.00)/0.25
(315,344,374)/0.25	(798000,923000,1048000)/0.25	false/0.25	(15.00,30.00,30.00)/0.25

In a commercial scenario, there is yet another step prior to adaptation. $\forall t \in T_r$ the $BillingCosts_t$ are calculated and only adaptation steps where $BillingCosts_t < money_r$ holds, remain in T_r, with $money_r$ being the maximum amount of money the client is willing to pay.

Layered coded streams. The algorithm allows to handle layered coded streams by treating them as a set of sources. A stream for which x layers out of n are cached is treated as if x different versions were cached. CPU costs are calculated as in the non-layered case. They are zero if a version matches the request; if not, an adaptor must be found that decodes, adapts, and encodes the stream.

6 Evaluation

Before integrating the cost function into the operational implementation of our proxy [12], experiments were performed in order to test the idea. We assumed $price_{net} = 10$, $price_{cpu} = 10$, and $price_{hd} = 10$ as "abstract" price values. For the capacities of the disk and the network, we assumed a disk bandwidth of 10 Mbyte/s and a network bandwidth of 10 Mbit/s. For CPU speed, we assumed a two-processor system, which is capable of decoding 75 million pixel/s.

We used WebTraff [5] to generate a list of (nearly) 10,000 requests. For the request pattern we assumed Zipf distribution with $\alpha = 1.0$, 0.75, and 0.3. [8] We simulated 1,000 visual streams each with a dimension of 352×288, a frame rate of 30 fps and a constant bit rate of 912,384 bit/s. All files used the same codec, i.e., syntax conversion did not occur. The total size (thus also the duration) of the streams followed a Pareto distribution with the tail index set to 1.2. Streams had a duration between one and 3,600 seconds; on average, stream duration was 84 seconds. Request interarrival time was set to 20 s.

The frame pattern was set to *IPBPB*... which allowed a 30 fps stream to be temporally adapted down to 15 fps. B-frames contributed 20% of the total bit rate. The total size of all streams was approximately 9 GB, cache sizes were set in the range from 1% up to 10% of this total size, the number of one-timers in the request sequence was set to 30 and 70%. For cache replacement we used standard LRU.

We ran several benchmarks with the number of users requiring transcoding (e.g. for mobile devices) varying between 0 and 100%, each benchmark was repeated 10 times with different request patterns. Differences in user requirements were simplified to four different devices, with the corresponding user preferences shown in Table 3. Requests with user preferences were pseudo-randomly distributed over the whole request sequence with every class being equally important.

We assume that the server does not perform transcoding, all transcoding work is done by the proxy. None of the videos is present in a layered coding format, thus, transcoding — usually an expensive operation — is never used just for cache replacement, as suggested in the literature [7].

6.1 Transcoding rules

In case a quality miss is encountered, the proxy tries to match the stored media stream versions to the request. In case of a *bit rate miss*, it drops B-frames until the specified bit rate range is reached or no more B-frames are left. If the resulting bit rate is still too high, transcoding in the decompressed domain is performed. Results generated by temporal adaptation in the compressed domain are never cached, the hit is assigned to the original video. A *spatial miss* always requires transcoding. The generated video is moved to the beginning of the LRU list, the position of the original video is not changed. A *frame rate miss* is dealt with by B-frame dropping until the specified frame rate range is reached or no more B-frames are left. If the resulting fps value is still too high, transcoding in the decompressed domain is used. Only transcoded versions are inserted into the cache.

If the proxy cannot create a version that matches the request (because of an invalid request or due to its admission control), it rejects the request. Adaptation costs are calculated as shown in Sect. 5. The proxy first determines which adaptors are needed and then calculates the costs that each adaptor causes.

[8] Due to space constraints we only present $\alpha = 1.0$ figures. For more details see [11].

6.2 Measured parameters

The following parameters are measured during benchmark execution:

- Rejected requests: How many requests were rejected due to unsatisfiable requests, either because the admission control rejected the request or because the requested transcoding step is not supported in the proxy? A rejected request also counts as an object and byte miss.
- Quality hits: How many requests could be fulfilled directly from the cache without the need for further transcoding?
- Object hits: How many requests could be fulfilled from the cache (including hits that needed adaptation)?
- Byte miss rate: How many of the requested bytes had to be fetched from the media server when an object miss was encountered or a request was rejected? We decided to count a reject as a byte miss — although no bytes are transferred — to distinguish between our proxy which detects rejection caused by quality mismatch in advance and a traditional proxy which would try to service even absurd requests such as streaming an HDTV video to a cell phone.
- Not cached due to locking: How many streams could not be inserted into the proxy cache because it could not free up enough space due to file locking?
- Not cached due to size: How many streams were not inserted due to the stream object exceeding a size threshold value?

6.3 Experimental setup

We tested with three different proxy configurations:

- Traditional Web proxy: A metadata *unaware* Web proxy employing LRU cache replacement. Metadata appended to a request is ignored.
- Traditional intelligent proxy: A metadata *aware* traditional Web proxy which is not capable of adaptation but the proxy is at least "smart" enough to detect and parse the metadata and to reject requests which would require transcoding.
- QBIX-G media gateway: Our adaptive media gateway which uses our cost formula and can perform adaptation in the decompressed domain.

These configurations were — where meaningful — benchmarked with different numbers of adaptation request:

- No adaptation scenario: This is the reference scenario where no client requires adaptation.
- 25% adaptation scenario: Every fourth request enforces transcoding in the decompressed domain.
- 100% adaptation scenario: All requests enforce transcoding in the decompressed domain.

Note that in the *No Adaptation Scenario* all configurations yield identical results because in this case our QBIX-G media gateway works simply as a traditional proxy. Thus we only present the results of the *Traditional Web Proxy* configuration, in the figures denoted by *Traditional Proxy, No Adaptation Requests*.

The *Traditional Intelligent Proxy* configuration was measured additionally in the *25% Adaptation Scenario* (see the *Traditional Intelligent Proxy, 25% Adaptation Requests* graph). The *100% Adaptation Scenario* was not meaningful with this configuration, since it would have resulted in 100% rejection rate.

The *QBIX-G Media Gateway* was benchmarked with 0, 25 and 100% adaptation requests, the no-adaptation scenario giving the same results as the traditional proxy. The 25% scenario is named *Media Gateway, 25% Adaptation Scenario*, the 100% scenario is called *Media Gateway, 100% Adaptation Scenario*.

Furthermore, we repeated some benchmarks with an *object size limit* for individual video objects which was set to 25% of the proxy disk cache size.

6.4 Results

The following results were found during the evaluation: (1) Adaptation is always better than rejection. (2) If the number of client requests demanding adaptation increases, the general cache characteristics (such as object hit rate, quality hit rate, and byte miss rate) are getting worse. (3) In the case of request rejections, an adaptive proxy avoids sending unusable data over the network. (4) Locking is a major problem for a media gateway, often leading to a situation where streams cannot be cached because not enough disk space could be freed (see Sect. 6.4.3 for a short discussion). (5) Introducing the object size limit is advantageous for both the media gateway and the traditional proxy. It improves quality and object hit rate and degrades byte miss rate. (6) Current hardware is fast enough to perform real-time adaptation in the decompressed domain for small proxy systems.

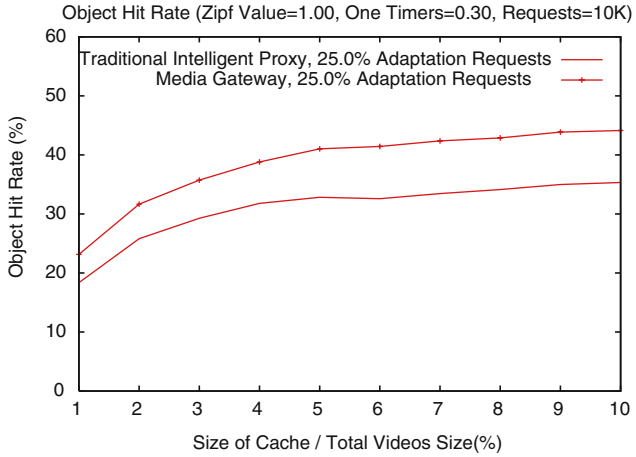
The following sections will provide more details on each of the findings.

6.4.1 Adaptation versus Rejection

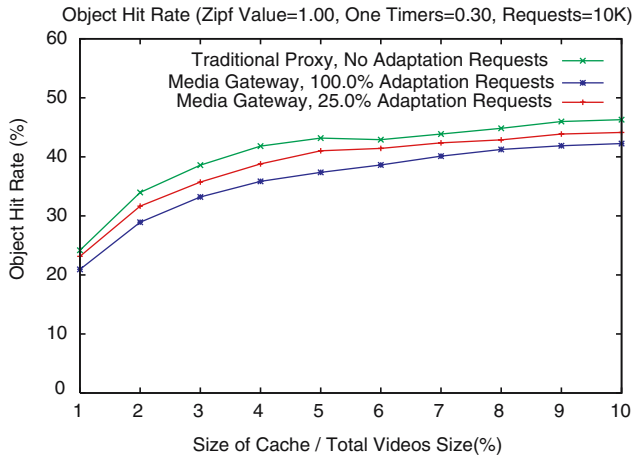
It is an obvious conclusion that adaptation is always better than rejection, yet it is a conclusion worth quantifying. For example, Fig. 7a shows the gain in object hit rate we can achieve by supporting media gateway functionality. The cases where requests for adaptation are simply ignored lead to an object hit rate as shown in the traditional intelligent proxy graph in Fig. 7a. In general, the higher the Zipf α value (meaning more skewed popularity), the more substantial is the gain in hit rate; with $\alpha = 1.0$ the difference is up to 10%.

6.4.2 Impact of number of adaptation requests

Figure 7b shows how the object hit rate evolves with an increasing number of adaptation requests. Generally, the hit rate decreases with an increasing number of adaptation requests. Analogous observations can be made for byte miss rate (Fig. 8a) and quality hit rate (Fig. 8b).

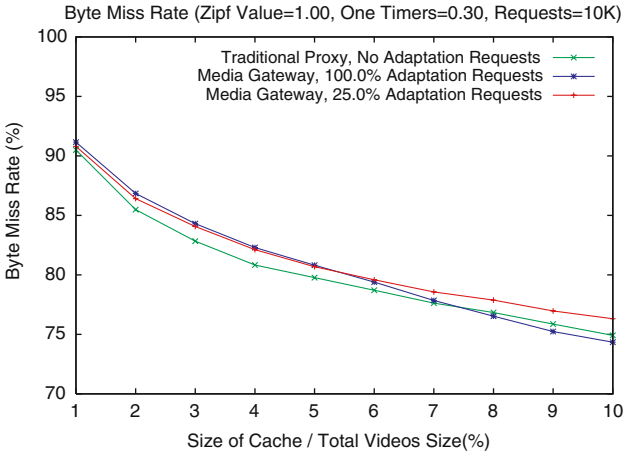


(a) Adaptation vs. Rejection Zipf $\alpha = 1.0$

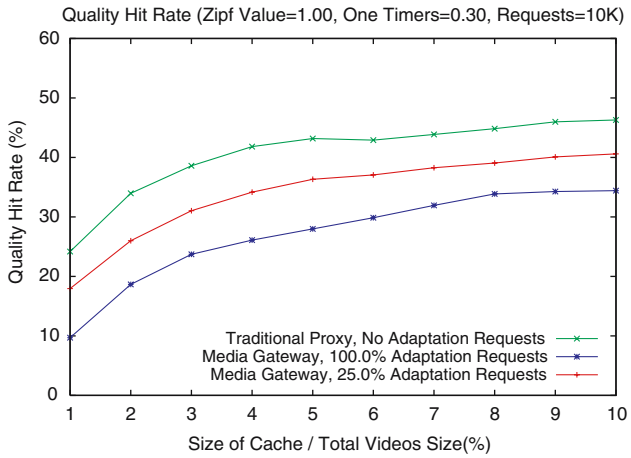


(b) Effect of Adaptation Zipf $\alpha = 1.0$

Fig. 7 Object hit rate



(a) Byte Miss Rate Zipf $\alpha = 1.0$



(b) Quality Hit Rate Zipf $\alpha = 1.0$

Fig. 8 Effect of adaptation (1)

The reason for this behavior lies in our assumption that no stream is available in a layered coding format, thus the sum of the sizes of all transcoded versions of a stream is larger than the size of a single layered coded version of the same stream. As soon as the cache size is less restricted, the media gateway can close the gap to the traditional proxy. The byte miss rate in Fig. 8a with a large cache size of 10% of the total size of all videos is a good example for that.

Another reason is that if there is enough disk space in the proxy cache, we also store the original versions of the stream. By making two insertions into the proxy cache for an object miss, we evict more streams from the cache for a single request than a traditional proxy (which only stores the original version). For a sequence of 10,000 requests, where all requests require transcoding, the number of insertions is as high as 16,200 for the

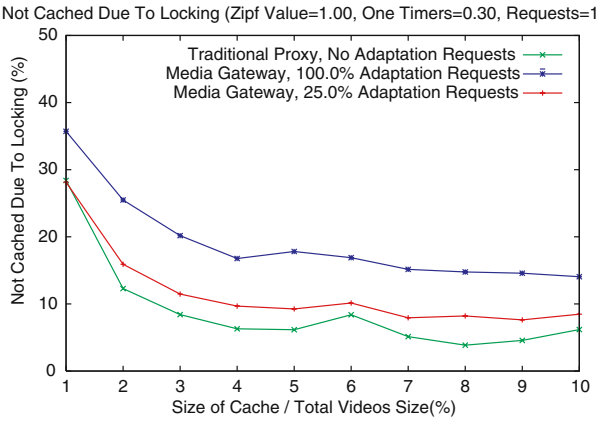
Zipf parameter $\alpha = 0.3$ case (cache size 10%). With $\alpha = 1.0$ this value is reduced to less than 10,800 due to a higher object and quality hit rate.

Quality hit rate is defined as how many requests can be reused directly from the cache without requiring transcoding. Thus, the higher the number of adaptation requests, the lower is the quality hit rate. The parameter tells us how much transcoding work was saved by caching transcoded versions. Quality hit rate is very sensitive to the Zipf α value. The higher this value, the better is the achieved hit rate.

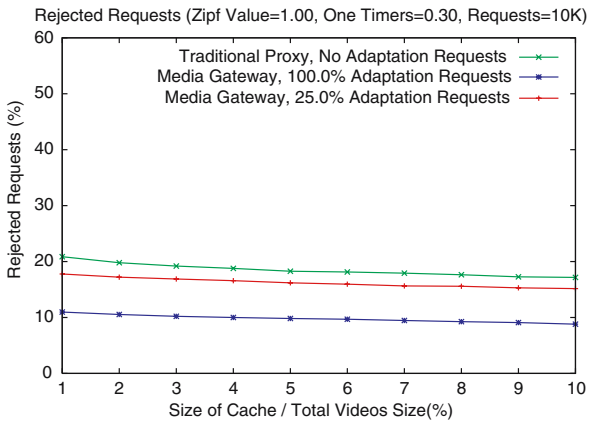
6.4.3 The problem of locking

Generally, locking in a video cache is worse than in a standard Web proxy cache. As in every cache, objects must be locked on the file level and excluded from cache replacement as long as they are in use by a client. While this amount of time is very short for small Web documents, it depends in the case of videos directly on the duration of the video streams. For example, consider the case where a client requests a two hour video. The video may be locked even for the whole play-out time, i.e., two hours. (The lock time can be less, however, if partial caching in the time domain is used.) Thus, large files can remain in the cache for a very long time, even if they are only one-timers. The situation gets worse with adaptive video proxies that support transcoding in the decompressed domain. Now, not only the original source video is locked in the cache but also the generated transcoded version. In the worst case, this means that for a single request, a zero-timer (the original video) and a one-timer (the transcoded version) block the cache for other (popular) objects.

As shown in Fig. 9a, 16–37% of all video insertions fail due to locking when all clients request adaptation, whereas in the non-transcoding scenarios this value is clearly lower. Interestingly, locking seems to be worse with highly skewed request patterns. The reason is that some large source streams remain in the cache for a very long time, e.g., consider a popular 100 s video being used as transcoding source. With $\alpha = 1.0$ it is very likely that during the time the original video is locked, another request will need the same video for a different transcoding step and extend the lock time for the original video. Thus, a constant (large) amount of the proxy cache is always locked, with the other less popular requests competing for the reduced space. Also, a growing cache size allows the proxy to cache larger videos that remain locked longer than short videos, leading to the "bumps" in the graphs in Fig. 9a.



(a) Not Cached Due To Locking Zipf $\alpha = 1.0$



(b) Request Rejection Rate Zipf $\alpha = 1.0$

Fig. 9 Effect of adaptation (2)

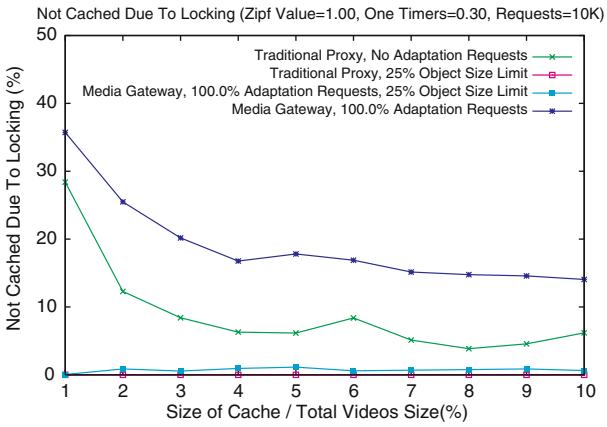
6.4.4 Improved request rejection rate

Figure 9b shows the percentage of requests being rejected by the admission control function. The traditional proxy rejects around 20% of all requests due to insufficient network bandwidth.

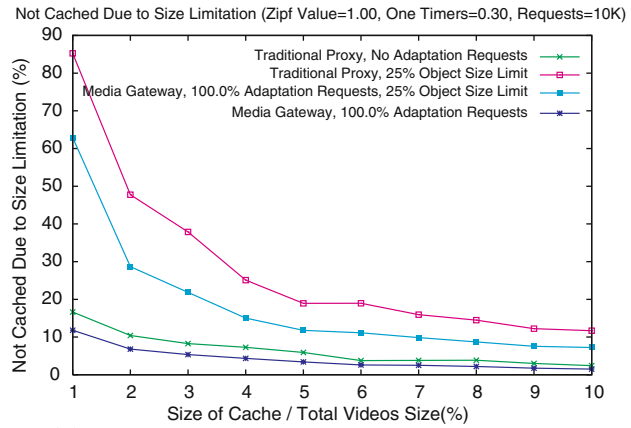
With increasing number of transcoding requests, request rejection decreases to approximately 10%. This is due to two reasons. First, the CPU of the benchmarked system is fast enough to cope with all transcoding requests, so if a request is rejected it is always due to insufficient network bandwidth. Second, 75% of low-end (mobile) clients require a bit rate lower than the original bit rate, which reduces the load on the resource network.

6.4.5 Effect of object size limit

For most measured parameters, the object size limit shows the same effect on both adaptation and non-adaptation scenarios, e.g. quality and object hit rates improve, byte miss rate decreases in both scenarios.



(a) Not Cached Due To Locking Zipf $\alpha = 1.0$



(b) Not Cached Due To Size Zipf $\alpha = 1.0$

Fig. 10 Effect of object size limit

Introducing an object size limit of 25% of the overall proxy cache size, significantly reduces the locking problem in both scenarios (Fig. 10a). In the traditional proxy the locking problem no longer exists, in the media gateway it reaches 1% at most. Due to the size limit and the higher bit rate of original video versions, these are more likely to exceed the size limit and are not cached. Thus, the size limit favors smaller objects, i.e., transcoded versions, with a bit rate lower than the original one. This allows the media gateway to store more objects, which improves quality and object hit rates. Figure 10b shows this behavior. The traditional proxy always has a higher request rejection rate than the media gateway. When only original video objects are requested, the traditional proxy rejects, for a medium cache size (5%), approx. 20% of all insertions because of their size, while the media gateway only rejects about 13%. Compared to the scenarios where no object size limit is set, these values are still high, though, which explains why byte miss rate is negatively affected. Generally speaking, introducing the object size limit improves object and quality hit rates at the price of worsening byte miss rate.

7 Conclusion and future work

We have presented a multimedia proxy gateway that makes a first step towards offering *What You Need is What You Get (WYNIWYG)* services. By combining user preferences, resource usage, and quality into one cost formula, we are able to determine which media stream version will give the client good enough quality with acceptable costs at the proxy. We have shown the effects of transcoding in the decompressed domain on the byte, object, and quality hit rates, and that locking is a major problem if the number of adaptation requests is high and one is forced to rely solely on transcoding. As long as the devices requiring adaptation remain a minority, transcoding is a feasible processing step in a media gateway proxy and a useful complement to layered coding support. The minor loss of object hit rate is compensated by the functionality gained and will be further reduced when efficient layered coding is available.

Further work is to be done, in order to extend the simulations to include layered coding and to implement the support for cost function and user preferences into our open-source QBIX-ViTooKi implementation. A combination of different partial caching schemes and domains, such as quality and time, promises further improvements and should be investigated.

Acknowledgements This project was funded in part by FWF (Fonds zur Förderung der wissenschaftlichen Forschung) P14788 and by KWF (Kärntner Wirtschaftsförderungsfonds).

References

1. Bormans, J., Hill, K.: N5231 — MPEG-21 Overview v.5. http://www.chiariglione.org/mpeg/standards/mpeg-21/mpeg-21.htm (2002)
2. DeMartini, T., Wang, X., Wragg, B.: N5599 — Study of Text of ISO/IEC FCD 21000-5 Rights Expression Language. http://xml.coverpages.org/MPEG21-W5599-StudyOfREL-FCD-200303.pdf (2003)
3. Fahmi, H., Latif, M., Sedigh-Ali, S., Ghafoor, A., Liu, P., Hsu, L.H: Proxy servers for scalable interactive video support. IEEE Comput. **43**(9), 54–60 (2001)
4. Kangasharju, J., Hartanto, F., Reisslein, M., Ross, K.W.: Distributing layered encoded video through caches. In: Proceedings of IEEE INFOCOM, pp. 622–636 (2001)
5. Markatchev, N., Williamson, C.: WebTraff: A GUI for web proxy cache workload modeling and analysis. In: IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, vol. 10, pp. 356–363 (2002)
6. Paknikar, S., Kankanhalli, M., Ramakrishnan, K.R., Srinivasan, S.H., Ngoh, L.H.: A caching and streaming framework for multimedia. In: Proceedings of ACM Multimedia, pp. 13–20 (2000)

7. Podlipnig, S., Böszörmenyi, L.: Replacement strategies for quality based video caching. In: IEEE International Conference on Multimedia and Expo (ICME), vol. 2, pp. 49–52 (2002)
8. Rejaie, R., Kangasharju, J.: Mocha: A quality adaptive multimedia proxy cache for internet streaming. In: 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video, pp. 3–10 (2001)
9. Reynolds, F., Hjelm, J., Dawkins, S., Singhal, S.: Composite capabilities/preference profiles (CC/PP): a user side framework for content negotiation. W3C Note 27 July 1999, http://www.w3.org/TR/NOTE-CCPP [2004-06-04] (1999)
10. Sasabe, M., Wakamiya, N., Murata, M., Miyahara, H.: Proxy caching mechanisms with video quality adjustment. In: Proceedings of the SPIE Conference on Internet Multimedia Management Systems, pp. 276–284 (2001)
11. Schojer, P.: QBIX-G: A quality based intelligent proXy gateway. PhD Thesis, Klagenfurt University (2005)
12. Schojer, P., Böszörmenyi, L., Hellwagner, H., Penz, B., Podlipnig, S.: Architecture of a quality based intelligent proxy (QBIX) for MPEG-4 Videos. In: ACM World Wide Web Conference, pp. 394–402 (2003)
13. Sen, S., Rexford, J., Towsley, D.: Proxy Prefix Caching for Multimedia Streams. In: Proceedings of IEEE INFOCOM'99, pp. 1310–1319 (1999)
14. Vetro, A., Timmerer, C.: N5845 - Text of ISO/IEC 21000-7 FCD - Part 7: Digital Item Adaptation. http://www.chiariglione.org/mpeg/working_documents/mpeg-21/dia/dia_fcd.zip (2003)
15. Wang, C.N., et al: M8887 — FGS-based video streaming test bed for MPEG-21 universal multimedia access with digital item adaptation. ISO/IEC Input Document (2002)
16. Zhang, Z.L., Wang, Y., Du, D.H.C., Shu, D: Video staging: a proxy-server-based approach to end-to-end video delivery over wide-area networks. IEEE/ACM Trans. Netw. **8**(4), 429–442 (2000)