

Architecture and pragmatics of server-directed transcoding

Björn Knutsson*

University of Pennsylvania
(bjornk@dsl.cis.upenn.edu)

Honghui Lu

University of Pennsylvania
(hhl@cis.upenn.edu)

Jeffrey Mogul

HP Western Research Lab
(JeffMogul@ACM.org)

Abstract

Proxy-based transcoding adapts Web content to be a better match for client capabilities (such as screen size and color depth) and last-hop bandwidths. Traditional transcoding breaks the end-to-end model of the Web, because the proxy does not know the semantics of the content. *Server-directed transcoding* preserves end-to-end semantics while supporting aggressive content transformations.

We show how server-directed transcoding can be integrated into the HTTP protocol and into the implementation of a proxy. We discuss several useful transformations for image content, and present measurements of the performance impacts. Our results demonstrate that server-directed transcoding is a natural extension to HTTP, can be implemented without great complexity, and can provide good performance when carefully implemented.

1 Introduction

Many web site designers face a dilemma: they must balance the richness of the user experience against the limited bandwidth and user interfaces of many Web clients. Media content, e.g. images create particular problems: large images increase download times, and may be hard to display on small screens. The Internet is seeing growth not just in users with wideband access, but also in people using portable narrowband clients with small displays.

To avoid discouraging the latter users, site design-

ers often employ fewer, smaller, or lower-quality images than they would otherwise prefer. Even so, many users encounter sites with excessive image complexity, either because a thoughtful site designer was unwilling to unduly compromise the experience of well-connected users, or because a thoughtless site designer failed to consider the impact of image size.

One can cope with this mismatch between content and capabilities by *transcoding* content to a more appropriate representation. For example, images can be reduced in size, cropped to eliminate details, or converted to monochrome. Transcoding can provide the user with the essential information of the original content, without straining the client or network capabilities.

Transcoding is lossy: while it preserves essential information, it removes (“distills”) inessential or unrenderable information, in order to meet goals such as bandwidth reduction. A transcoding system must make a tradeoff between loss of detail and loss of effectiveness at meeting its goals. Too little distillation, and the bandwidth costs (for example) will still be prohibitive; too much distillation, and the underlying message is lost.

Transcoding is often done at a proxy. Proxy-based transcoding, as we discuss in Section 1.1, can improve performance and scaling. Traditional transcoding proxies operate autonomously; they make transcoding decisions based on heuristics and implicit information, such as the HTTP “Content-type” header. Implicit information can be ambiguous, so autonomous transcoders can make bad tradeoffs.

*Supported by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795.

This problem inspired one of us to propose *server-directed transcoding*, or SDT, in a previous paper [12]. In this approach, the origin server provides explicit guidance to the transcoding system about whether and how to convert between representations. While autonomous transcoding breaks the end-to-end model of the Web, because the proxy does not know the semantics of the content, SDT preserves end-to-end semantics. SDT also supports more aggressive content transformation than can autonomous transcoding, because there is no risk of accidentally eliminating important content.

In this paper, we show how SDT can be integrated into the HTTP protocol, with a compatible and simple extension. We present the architecture and implementation of a proxy system that supports SDT, using a combination of mobile applets and native code. We discuss several specific, useful transformations for image content. Finally, we report on some simple experiments with our system, including performance measurements.

1.1 Benefits of proxy-based transcoding

One argument against proxy-based transcoding is that the origin server could itself produce any necessary transcoded results, avoiding the complexity and risks of transcoding. This argument asserts that server resources are plentiful, and that server-based transcoding provides the same reduction in last-hop bandwidth requirements.

We see several benefits of proxy-based transcoding, deriving mostly from the increased opportunities for proxy caching. Because a proxy can cache both the original and transcoded instances, generating new transcodings from the same cache entry, it can provide responses to a wide variety of clients with only one access to the origin server (see Figure 1). Proxy-based transcoding enables the proxy cache to do a better job.

Proxy-based transcoding thus improves the long-term scaling of the Web, by shifting load from origin servers and the core of the Internet to easily-replicable proxies. Proxy-based transcoding

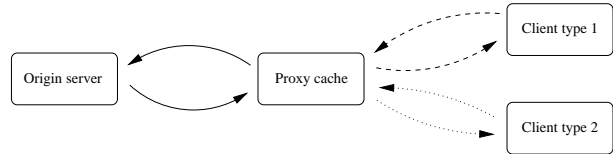


Figure 1: Caching, transcoding proxy serving multiple client types.

also improves client-perceived latency, by avoiding round-trips to the origin server when transcodable content can be served from the cache.

Proxy-based transcoding, even without proxy caching, would still shift computation load away from origin servers. However, we believe that it is best when combined with caching.

The argument has also been made that transcoding, at any location, is unnecessary because bandwidths and client capabilities are increasing.

We reject this premise. The skew between “high-bandwidth” and “low-bandwidth” connections is increasing, not decreasing, as is the variety of client devices and capabilities. So we cannot just split the world into high- or low-bandwidth. In fact, diversity in other client capabilities, such as screen size, are just as important as bandwidth, and the variety in the device space increases rapidly. Transcoding allows a site to serve a diverse population without having to generate distinct sub-sites for each category of client.

2 Concepts and context

In this section, we summarize the concepts behind server-directed transcoding, and show how SDT fits into the context of the Web.

2.1 Problems with autonomous transcoding

In traditional transcoding, a proxy infers the nature of response, and tries to transform the response without losing the essential information [5]. We refer to this as *autonomous transcoding*. For example, if the response carries `Content-type:`

image/gif then the proxy might convert it to a JPEG image. However, the proxy has no specific information about what level of lossy compression can be applied to the image without losing its meaning, and so the proxy must employ heuristics.

Several researchers have recognized the problem of choosing the right transformation, and have investigated various heuristics [3, 6]. However, no heuristic can avoid risking either the loss of important semantics, or the loss of opportunities for aggressive transcoding.

Another problem inherent in autonomous approaches is that of maintenance. The code that implements transcoding in the proxy will need to be continuously updated to cope with new formats and client requirements. Unless this code is run in a secure execution environment, each such update also risks creating new bugs, exposing the proxy to attacks.

2.2 General design of server-directed transcoding

Server-directed transcoding solves the inference problem by eliminating it. Instead, while transcoding occurs at the proxy, all decisions are made at the direction of the origin server. The transcoding process (conceptually) runs as part of the origin server application, and executes remotely from the origin server solely as a performance optimization, not for any functional reason. The proxy provides the execution environment and certain deterministic subroutines, but makes no autonomous decisions about transcoding. (SDT can take place at an end-client, as well as at a proxy, but this paper focuses on proxy-based SDT.)

To provide flexibility, our approach to SDT uses mobile code (such as Java applets) to extend server functionality into the proxy. Each SDT-capable HTTP response carries a reference to a transcoding applet, chosen by the site designer. To avoid the costs of loading a different applet for each response, we expect that each of a small set of applets can transcode many responses from a wide variety

of servers. Proxies would cache such applets. To support response-specific control, an SDT-capable HTTP response can also carry parameters for the applet.

For example, a widely-useful applet might crop an image based on coordinates given in parameters, as in this HTTP response message:

```
HTTP/1.1 200 OK
Date: Tue, 04 May 1999 22:51:34 GMT
Content-type: image/gif
SDT-applet: java=http://example.com/crop.jar
SDT-parms: crop-origin=140;300,
           crop-size=100;100
```

We present more details of the protocol extension in Section 4.

The transcoding choice depends both on the semantics of the content, and on client-specific attributes. Transcoding depends on a means for clients to express these attributes. In some prior designs, this expression is implicit: for example, the proxy looks at the User-agent header in a request to infer client capabilities. We believe these attributes should be explicit, and propose that clients use “feature sets” [10] to communicate this information to a proxy, as we discuss in Section 4.1.

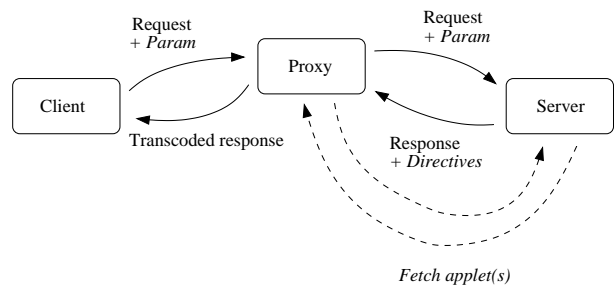


Figure 2: SDT communication paths. *Italics* indicate new HTTP headers; dashed arrows indicate additional messages.

Figure 2 illustrates the communication paths in SDT. The only new messages are for applet retrieval, which can usually be avoided via caching. (The figure shows the proxy fetching the applet from the origin server, but it could come from another server.)

2.2.1 Terminology: what does transcoding operate on?

We have described transcoding as operating on HTTP “responses.” This is imprecise, because HTTP allows partial responses (e.g., the results of byte-range requests), and because some HTTP responses carry no transcodable content. The term *instance*, meaning “the full response to a GET request the requested resource at the current time,” precisely describes both the input and the output of transcoding, and we use that term for the rest of this paper. (For more discussion of this issue, see [13].)

2.3 Who is in control?

Because there are three parties to a transcoding operation (the origin server, the proxy, and the client), the question arises as to “who is in control?” – that is, which party decides what transcoding to apply, if any? The question appears to require resolution of competing and irreconcilable interests. For example, the client only wants images smaller than 50x50 pixels, but the server cannot provide useful semantics for an image smaller than 100x100.

We argue that “who is in control?” is the wrong question; one should instead ask “who is in control of what?”. We answer:

- The origin server should control the *semantics* of transcoded content (through its agent, the transcoding applet).
- The client should control the *presentation* of transcoded content.
- The proxy should control the *resources* used for transcoding, caching, and network traffic.

What if two of these parties disagree? We considered proposing rules for setting priorities, but this seems arbitrary and complex. Instead, we use the simple rule that “any party has a veto.” Either the server (via the applet) or the proxy can decide not to transcode an instance, to preserve semantics or protect resources. If an instance is not transcoded, the client can then decide to reject it (e.g., to omit an image from a page). The proxy can implement the

client’s decision, to avoid wasting bandwidth sending an image that the client won’t display.

These rules leave the transcoding applet free to arbitrarily compromise semantics in order to meet client preferences, *if* the server is willing.

3 End-to-end issues in transcoding

HTTP expects proxies to preserve various “end-to-end” properties that applications rely on. The use of proxies (caching or not) should affect performance, but not the meaning of the communication.

Transcoding appears to break these end-to-end properties: By definition, transcoding changes the body of a response message. And because transcoding breaks the relationship between an entity tag and the associated instance, a proxy that receives an ETag header cannot transcode without violating the specification. The specification resolves this by excusing “non-transparent” proxies from the rules about headers, but then provides no guarantees at all.

These paradoxes arise from the autonomous operation of proxy-based transcoders. In SDT, however, the transcoding process is conceptually part of the origin server, not of a proxy. The proxy provides an execution environment and resources, but it does not make transcoding decisions. The transcoder runs at the proxy (rather than the server) solely as a performance optimization, not for functional reasons.

Therefore, SDT preserves HTTP’s end-to-end model. The transcoder *is* the origin server: it can generate any necessary headers, assign entity tag values, etc.

Space does not permit detailed discussion of several end-to-end issues. These include the assignment of entity tags to transcoded results (see [12, section 5.7] for more about this), the possibility of re-transcoding an SDT result at a subsequent proxy, and the controlled combination of SDT and autonomous transcoding.

4 Protocol extensions

Server-directed transcoding requires a minor extension to HTTP, to allow servers to communicate their directives. Our (limited) experience has led us to a specific design, although surely this requires refinement before standardization.

In our applet-based approach to SDT, the server must communicate two kinds of information: A reference to the transcoding applet(s), and optional response-specific parameters for the applet(s). Using the Augmented BNF notation used by HTTP [4], we define two new response headers:

```
SDT-applet = "SDT-applet" ":"  
            1#( exec-env "=" absoluteURI )  
exec-env = "java" | "perl" | token  
SDT-params = "SDT-params" ":" field-value
```

The SDT-applet header specifies one or more applets; each applet is associated with an *execution environment*, such as Java or Perl. The provision for multiple applets allows the proxy to choose among several applets, perhaps based on which execution environments it supports (or prefers to use). Each execution environment imposes its own requirements on applets.

We place no prior restrictions on the syntax of the SDT-params header (beyond the basic HTTP rules). This header is passed, verbatim, from the origin server to the applet; its syntax is of no interest to any other parties.

4.1 Communicating client capabilities and preferences

We do transcoding to meet the specific needs of a client, so clients must have a means to express their capabilities and preferences. The IETF has already started standardizing a syntax for representing *Media Feature Sets* [10], which appears adequate for SDT. For example, this feature set:

```
(& (pix-x=150) (pix-y=100) (color=15) )
```

says that the client can display a 150x100 image using 15 colors. The syntax also allows a client to quantify its relative preferences. (We believe the

quantification syntax supports the “veto” mechanism of Section 2.3).

No standard yet exists for an HTTP header to carry feature sets. Holtman and Mutz proposed “Accept-Features” but suggested that “Feature-Set-Info” is more accurate. We propose using the abbreviation “FSI” as the name of a header carrying “feature set information” about a response, and “FSP” to carry a “feature set predicate” in a request. (A feature set predicate describes what the client can accept [10].)

The currently-registered *Media Feature Tags* [9] cover client capabilities, but not the relevant network properties. Because transcoding is often done to save bandwidth, we would like a means for clients to tell proxies how they are connected. (The client-to-proxy path can be complex, but the last hop is quite often the bottleneck.) We propose using Feature Tags to represent a client’s last-hop bandwidth and preferred maximum image size (in bytes), although this does stretch their intended purpose.

If a transcoder cannot produce a result that both preserves semantics and meets client preferences, normally it should not send any result (or it could waste bandwidth). However, we propose defining a Feature Tag to allow a client to request that it always receive some result that meets its preferences, even if the semantics are lost. (The applet is free to ignore this request.)

4.2 A protocol example

We present a simple example to show how the protocol extensions would be used. (This repeats the partial example from Section 2.)

Suppose that desktop client *A* requests `http://www.example.com/logo.gif` via a proxy:

```
GET http://www.example.com/logo.gif HTTP/1.1  
Host: www.example.com  
FSP: (& (pix-x<=1280) (pix-y<=1024)  
      (color<=65536))
```

Assuming a cache miss, the proxy would forward that request to `www.example.com`, which might re-

ply with a 400x500 pixel color image:

```
HTTP/1.1 200 OK
Date: Tue, 04 May 1999 22:51:34 GMT
Content-type: image/gif
SDT-applet: java=http://example.com/crop.jar
SDT-parms: crop-origin=140;300,
           crop-size=100;100
FSI: (& (pix-x=400) (pix-y=500) (color=156))
Etag: "123efg"
```

The proxy simply forwards this response to *A*, because the features of the response match the feature-set-predicate of the request [10].

Now imagine that client *B* is a handheld with a 160x160 pixel, 4-bit grayscale screen. *B* requests the same image via that proxy:

```
GET http://www.example.com/logo.gif HTTP/1.1
Host: www.example.com
FSP: (& (pix-x<=160) (pix-y<=160)
      (color=0) (grey<=16))
```

The proxy recognizes this as a cache miss, but detects¹ that the cached response is incompatible with the client's feature-set-predicate. Therefore, the proxy obtains the applet `http://example.com/crop.jar` (either from its cache or from the server), and then invokes the applet with the cached response and client *B*'s request as input. The applet, which (as its name implies) crops images, might generate this response:

```
HTTP/1.1 200 OK
Date: Tue, 04 May 1999 22:51:34 GMT
Content-type: image/gif
FSI: (& (pix-x=100) (pix-y=100) (grey=16))
Cache-control: no-transform
Etag: "123efg.100x100g16"
```

Note that the applet has cropped the original 400x500 image to 100x100, because this fits within *B*'s 160x160 screen capability. The applet has also converted the color image to a grayscale image, using the full pixel depth available at the client. The SDT-related headers have been removed because the result cannot undergo further transcoding, as indicated by the Cache-control directive. The entity tag has been transformed by the applet (see [12,

¹Either by explicitly trying to match the client's feature set predicate with the server's feature set information, or by invoking a method of the applet to make this decision.

section 5.7]) in such a way that the applet can validate the client's cache entry. (No other agent besides this applet and the origin server can interpret this entity-tag.)

5 Proxy environment: architecture and extensions

A proxy that supports SDT must provide an environment for executing transcoding applets. This includes:

- **A runtime environment for applet execution:** For Java applets, a Java Virtual Machine provides a secure environment and isolation between applets. For Perl applets, the Perl interpreter provides similar features.
- **An applet loader and cache:** the proxy retrieves transcoding applets via URLs specified in the SDT-applet headers, and can cache them. Traditional HTTP retrieval and caching mechanisms should suffice.
- **Input and output instances:** the proxy provides the input instance, and allocates storage for the output instance.
- **A means to supply client request headers to applets:** this gives applets access to client capabilities and preferences.
- **Proxy-specified parameters:** for example, the proxy may set limits on resource use by the applet, or provide access to cache entries holding previous transcodings of the input instance.
- **A standardized set of APIs for use by applets:** space does not permit discussion of these APIs, and we are still developing the details.

Figure 3 shows the interaction between an applet and its environment.

The model shown in Figure 3 is sufficient to support SDT. However, this places the entire transcoding

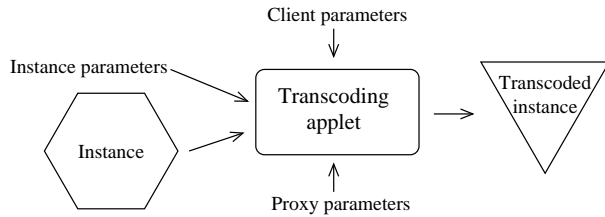


Figure 3: Proxy environment for transcoding applets.

functionality into one applet. We can instead factor the functionality into three pieces: a *control applet*, one or more *function applets*, and one or more *native functions* implemented by the proxy itself. The latter might be invoked via applet wrappers. Only the control applet is mandatory; the other pieces are optional optimizations.

The motivation for function applets is that while resources may differ enough to require separate control applets, these control applets may want to share underlying functions. For example, two control applets that use color-map reduction may have quite different algorithms for choosing colors, but can share a function applet that manipulates GIF color maps.

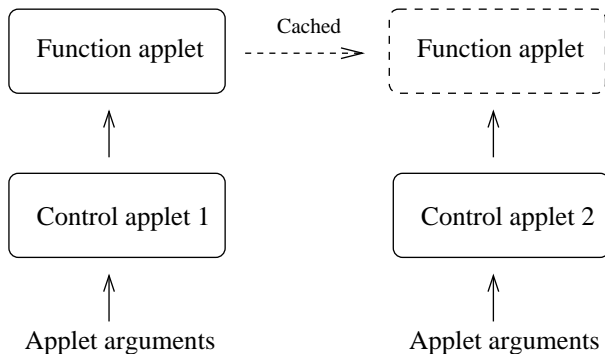


Figure 4: One *function applet* can be cached and reused by different *control applets*. Only the control applet has a thread of control.

This separation of control from function, as depicted in Figure 4, can reduce the size of control applets, and increases the cache hit rates for the implementations of common functions.

Another problem with placing the entire transcoding functionality into an applet is that e.g. Java may not provide the most efficient way to implement the

CPU-intensive functions used in image manipulations. Our architecture therefore allows the proxy to use *native functions* to implement function applets. This allows proxy vendors to differentiate, by offering high-efficiency execution of SDT without changing the semantics.

5.1 Practical issues for execution environments

Each execution environment imposes its own requirements on transcoding applet. For example, in the Java environment, control applets must export certain methods such as one to transcode a response, and another to estimate the cost of transcoding a response). The proxy must export a standardized mechanism for control applets to load and call functional applets.

The environment must also ensure security. In general, this means that the execution of one transcoding operation cannot affect any other proxy operations, and cannot overutilize proxy resources. Java ostensibly provides adequate security; the “taint mode” of Perl might also suffice. Because we do not allow transcoders to modify any persistent or external state, except for the transcoded values that they generate, opportunities for information leakage (e.g., privacy violations) are limited. However, we do not expect to fully control information leakage; this is best done through end-to-end security, such as encryption, for privacy-critical data. For responses that need not be secure, we *want* to allow “leakage,” in the form of cache hits.

5.2 Caching of transcoded results

Section 1.1 argued that proxy-based transcoding is best when combined with caching. We want to cache transcoded results, and use them for future client references, avoiding a transcoding operation if a suitable cache entry exists. How does the cache match transcoded results against the preferences in requests, given that two clients may have slightly different preferences that both match a single result?

One possible approach is for transcoders to tag re-

sults using Media Features, then have the cache use standard feature-set matching rules [10] to search for a matching entry.

The alternative is to let the applet decide: for a given URL, the proxy invokes a cache-match method of the associated transcoder, providing the client preferences and references to a set of cache entries for that URL. The applet can apply its response-specific knowledge to choose the best cache entry, or to decide to generate a new transcoding.

We suggest a hybrid approach: if the applet labels its output with feature tags, this implies that the cache should use standard matching rules. Otherwise, the cache should invoke the applet's decision method.

Note that these approaches imply a cache that stores multiple fresh entries per URL. This can affect the cache replacement policy. Previous studies have considered this problem [1, 14]; we view replacement policies as future work, and assume infinite caches in our tests.

6 Implementation status

Our current implementation is incomplete, but functions sufficiently to demonstrate the performance of SDT using actual HTTP transactions.

6.1 Proxy and applets

Our proxy prototype is implemented in Perl, and provides all the basic functions needed to write applets and dynamically transcode contents. It supports control applets written in Perl. Native functions (see Section 5) are external programs from the ImageMagick[8] package (e.g. `convert`).²

The current implementation caches neither instances nor applets, and operates sequentially with no pipelining or other optimizations.

²ImageMagick includes a library of image manipulation functions that could serve as native functions in a Java implementation.

We have implemented a few simple Perl applets. Section 7.1 describes our most interesting applet in some detail.

6.2 Clients and servers

Any HTTP client (with proxy support) can use our proxy. Support for sending feature sets (see Section 4.1) is desirable; alternatively, users could register their preferences via a Web interface to the proxy (as in [5]).

An HTTP server implements SDT simply by attaching response headers, as in Section 4. Apache supports this via the `MOD_HEADER` extension, or (as in our experiments) using `send-as-is`, with response headers embedded in the disk file containing the resource.

7 Useful transformations

Although almost any content could be transcoded, our work has focused on static images. We have identified several useful basic image transformations, including: cropping, scaling (in rendered dimensions), resolution reduction, lossy compression (“quality reduction”), and color remapping (see [12] for an example). All of these can reduce response size; all except compression can also adapt images to limited client capabilities.

An applet can use combinations of basic transformations either to satisfy multiple goals (e.g. small screen *and* slow network), or to better achieve a single goal without compromising semantics.

Each basic transformation takes its own parameters. For example, cropping can be described using a *crop box*: two corners in image coordinate space. Scaling can be described using *X*- and *Y*-axis factors.

While applet reuse helps the proxy avoid the cost of loading applets, we do not expect one applet to suit all purposes. Instead, we expect the use of a modest set of “category-specific” applets that would be associated with particular kinds of images. These

might include applets for “image with main subject,” “landscape panorama,” “banner ad,” etc. This categorization should greatly simplify both the control applets and their parameters.

7.1 An example applet

The applet we have chosen for our timing experiments (Section 8) transforms images with one central human subject. The goal of such an image is (usually) to convey the identity or emotional state of the subject, so when fitting the image to a small display, we prefer to crop out the surroundings (or even the person’s body), rather than shrinking the entire image to the point of unrecognizability.

This applet, called “crop-interp,” uses a list of crop-box parameters to reduce the image to the client’s desired size, by interpolating between the crop boxes. If the target size is smaller than the smallest crop box, the applet first crops to that box, then rescales the image as necessary. Otherwise, the applet interpolates a new crop box between the two that bound the target size. This preserves the semantics of “photo of human” as much as possible, while meeting the client’s desired size. If the client *can* display the surroundings that give flavor to the photo, these are preserved, too.

The crop-interp applet can also be used to meet a byte-size target, since cropping also reduces byte size.

Figure 5 shows an example image (one of the authors), with the crop boxes superimposed.³ Figure 6 compares the results of a traditional transcoding (quality reduction only, 5830 bytes) and of crop-interp (5831 bytes). While the transfer sizes are nearly identical, the image utilities differ greatly.



Figure 5: Example image, with crop boxes superimposed

7.2 Examples of other category-specific applets

Applets for other image categories might follow similar but distinct approaches. An applet for panoramic images could start by reducing quality or color depth, rather than cropping, and would only crop once further quality reductions would hurt semantics. A banner-ad applet could simplify the color map to eliminate graphically complex parts of the image. An applet for animated GIFs could cut to the main message, skipping a preliminary animation.

8 Experiments

We report two experiments. The first measures transcoding costs, while the second measures the whole system.

8.1 Transcoding costs

Transcoding costs for a response depends on the applet, its parameters, and on the response itself. Ta-

³The Postscript versions of the images in this paper have been transcoded to improve Postscript rendering speeds. The original JPEG images are available from <http://www.cis.upenn.edu/~7ebjornk/wcwimages/>



600x800 pixels, quality=3, 5830 bytes



200x267 pixels, quality=50, 5831 bytes

Figure 6: Transcoded images: quality-reduced (left) and crop-interp (right).

Table 1 shows elapsed times and size reductions for crop-interp, when asked to shrink and rescale the image in Figure 5. Measurements are means over 10000 runs on an IBM ThinkPad T23 (1.2GHz Pentium III, 384MB RAM).

A crop-interp operation consists of image unpacking, zero or more image manipulation operations, and image packing (including a possible format conversion and/or JPEG Quality Factor (QF) reduction). Cropping is cheap, so most of the cost of crop-interp is unpacking and packing the image. These costs are a function of the image size, and of the QF requested.

8.1.1 Crop-interp compared to simple compression

The left image in Figure 6, uncropped but with a $QF = 3$, took 0.170 seconds to transcode (comparable to the uncropped result in Table 1). The cropped image on the right (still quite recognizable) took just 0.086 seconds, so SDT seems to offer faster results than autonomous transcoding. One could

argue that had an autonomous proxy known that a 200x276 image was acceptable, it could have generated Figure 7, by rescaling and changing the QF to 34. However, this transcoding took 0.276 seconds.

Also, while autonomous transcoders need not load and process server directives, they should probably spend time (not measured here) on heuristics [3].

8.1.2 Evaluating image quality

While transcoding costs and image sizes are objective metrics, image quality is subjective. The scaled image in Figure 7 seems clearly superior to the quality-reduced image in Figure 6, but is the cropped image in Figure 6 even better?

Assuming that crop-interp does capture the important essence of the original image, we note that the scaled image in Figure 7 reduces this “important” 200x267 section of the original to 66x88, only 11% of the original pixels.

We conclude that SDT can yield better results

Resolution	Pixel reduction	Byte size	Size reduction	Time
600x800	Original	65982	Original	---
600x800	0%	48617	26%	0.178
485x650	34%	27428	58%	0.152
300x400	75%	10433	84%	0.102
200x267	89%	5831	91%	0.086

Table 1: Size savings from crop-interp, combining cropping with QF reduction from 75 to 50.



200x267 pixels, QF=34, 5815 bytes

Figure 7: Autonomous transcoding, matching size of crop-interp result.

than autonomous transcoding, with similar or better costs.

8.2 Whole system tests

We measured whole-system performance with a client (1.5 GHz Pentium 4, 512MB) making requests to our department’s Web server via our proxy (same hardware as in Section 8.1). Applets were downloaded from the server.

We simulated various proxy-to-client bandwidths

using the Linux traffic shaper on the proxy.⁴

We ran the same transcodings as in Table 1. For calibration, we measured non-transcoding transfers, with and without the proxy, of the original file and of an empty file. Table 2 gives the results.

The proxy uses “store and forward”, so it adds some overhead, even when not transcoding, which increases with response size. For any transcoding trial, our non-caching proxy downloads the applet; we measured this cost as 0.030 seconds for crop-interp. Starting crop-interp takes 0.008 seconds, and actual transcoding costs are shown in Table 1.

Transcoding markedly improves latency for slow networks (up to 144Kbps, for our prototype). For faster connections, the main reason to transcode is for content adaptation. A faster proxy implementation might shift the tradeoff point.

We conclude that, even for our unoptimized prototype, SDT adds minimal overhead compared to the potential benefits. Once we add caching to our prototype, the benefits should be even greater.

9 Future work

Much work remains to prove the value of SDT. We continue to work on implementation, including caching and performance optimization, and further measurements. We expect to refine the protocol extension and API designs, as we gain experience with new applets. We need to devise resource controls to protect proxies from buggy applets and

⁴This only affects outgoing packets. Packets to the proxy (HTTP request and TCP ACKs) were not shaped.

Resolution	Byte size	56Kbps (e.g., modem)	144Kbps (e.g., 3G wireless)	768Kbps (e.g., DSL)	10Mbps Ethernet	No proxy (Ethernet)
Empty	0	0.080	0.036	0.032	0.033	0.021
600x800	65982	9.680	3.720	0.510	0.110	0.080
600x800	48617	7.440	3.030	0.624	0.327	—
485x650	27428	4.340	1.800	0.430	0.272	—
300x400	10433	1.810	0.850	0.268	0.208	—
200x267	5831	1.120	0.560	0.211	0.185	—

Table 2: Whole-system results (means of 30 trials). First two entries are not transcoded.

denial-of-service attacks. We believe that the sand-boxing inherent in SDT can resist other security attacks, but this requires more analysis.

We think SDT could be extended to support CDNs and streaming media, although the benefits of these extensions may be more difficult to prove.

10 Additional related work

Maheshwari *et al.* [11] discuss the problem of matching client requests to cached transcoded results. Instead of tagging cache entries with explicit media features, they subdivide entries based on a small set of client categories.

Our decomposition of transcoding into control and functional applets resembles the approach of Ihde *et al.* [7], although they use pattern-matching instead of server direction to discover the composition.

Active Cache [2] and Active Names [15] also introduce code into the proxy. In Active Cache [2], server-supplied applets implement a cache-consistency mechanism; they can also collect client access logs and rotate advertising banners. In Active Names, mobile programs can customize how the resource is located and how the response is transported to the client. In contrast with these approaches, SDT operates only locally on each response to be transcoded, and does not interact with other systems or objects. Transcoding is a well-defined problem that can be solved by running applets in a restricted execution environment, thus avoiding difficult security and resource-

management issues.

Older related work is discussed in detail in [12, section 2].

11 Summary and conclusions

We have specified an HTTP protocol extension for Server Directed Transcoding, and an architecture for SDT applets and SDT proxies. Our architecture decomposes transcoding into category-specific control applets, generic functions, and response-specific parameters. We also decompose generic functions into loadable function applets and fast, native functions.

Experiments on an unoptimized prototype demonstrate the potential of this architecture, with performance sufficient to improve end-to-end latency even on 768Kbps ADSL connections.

We showed that SDT allows the use of significantly cheaper and better transformations than possible with autonomous transcoding, by exploiting response-specific information.

Acknowledgments

We would like to thank Benjamin Pierce and Bryan Hopkins for their helpful comments on drafts of this paper. We also thank the anonymous reviewers for their comments and suggestions.

References

- [1] S. Acharya, H. F. Korth, and V. Poosala. Systematic multiresolution and its application to the World Wide Web. In *Proc. 15th International Conference on Data Engineering*, pages 40–49, Sydney, Australia, March 1999. IEEE Computer Society.
- [2] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching dynamic contents on the Web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, Sept. 1998.
- [3] S. Chandra and C. S. Ellis. JPEG compression metric as a quality aware image transcoding. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems*, pages 81–92, Boulder, CO, October 1999.
- [4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol—HTTP/1.1. RFC 2616, IETF, June 1999.
- [5] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 160–170, Cambridge, MA, October 1996.
- [6] R. Han, P. Bhagwat, R. LaMaire, T. Mumert, V. Perret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile WWW browsing. *IEEE Personal Communication*, 5(6), Dec 1998.
- [7] S. C. Ihde, P. P. Maglio, J. Meyer, and R. Barrett. Intermediary-based transcoding framework. In *Poster Proc. of the 9th Intl. World Wide Web Conf.*, Amsterdam, Netherlands, May 2000. <http://www9.org/final-posters/63/poster63.html>.
- [8] ImageMagick Studio LLC. Imagemagick. <http://www.imagemagick.org/>.
- [9] Internet Assigned Numbers Authority. Media feature tags registry. <http://www.iana.org/assignments/media-feature-tags>.
- [10] G. Klyne. A syntax for describing media feature sets. RFC 2533, IETF, March 1999.
- [11] A. Maheshwari, A. Sharma, K. Ramaritham, and P. Shenoy. TranSquid: Transcoding and caching proxy for heterogenous e-commerce environments. In *Proc. 12th IEEE Workshop on Research Issues in Data Engineering (RIDE '02)*, San Jose, CA, Feb 2002. <http://lass.cs.umass.edu/papers/ps/RIDE02-trans.ps>.
- [12] J. C. Mogul. Server-directed transcoding. In *Proc. 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.
- [13] J. C. Mogul. Clarifying the fundamentals of HTTP. In *Proc. 11th Int'l World Wide Web Conf.*, pages 444–457, Honolulu, May 2002.
- [14] A. Ortega, F. Carignano, S. Ayer, and M. Vetterli. Soft caching: Web cache. In *Proc. 1997 Workshop on Multimedia Signal Processing*, Princeton, NJ, June 1997. IEEE Signal Processing Society.
- [15] A. Vahdat, M. Dahlin, T. E. Anderson, and A. Aggarwal. Active Names: Flexible location and transport of wide-area resources. In *USENIX Symposium on Internet Technologies and Systems*, 1999.