Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation

Armando Fox and Eric A. Brewer
University of California, Berkeley

Low Bandwidth Surfing in a High Bandwidth World

Today's WWW is beginning to burst at the seams due to lack of bandwidth from servers to clients. Most WWW pages are designed with high-bandwidth users in mind (i.e. 10 Mbit Ethernet) , yet a large percentage of web clients are run over low-speed 28.8 or 14.4 modems, and users are increasingly considering wireless services such as cellular modems running at 4800-9600 baud. A recent study by a popular server of shareware, Jumbo, revealed that about 1 in 5 users were connecting with graphics turned off, to eliminate the annoying latency of loading web pages. This is of particular concern to corporate advertisers, who are paying for the visibility of their corporate logo.

Since there is no way to optimize a single page for delivery to both high-bandwidth and low-bandwidth clients, some sites, such as [Xinside] , offer multiple versions of pages (no graphics, minimal graphics, full graphics). Most sites, however, do not have the human resources or disk space to do this.

Clearly an automated mechanism for closing the bandwidth gap is needed. Several such mechanisms are currently deployed, but all are inadequate:

- Progressive and interlaced GIF and JPEG display a blurry image initially and refine it as more image data arrives. However, for large images, the initial latency is still high, and once the client makes the initial requests, the capability for "choreographing" the download of multiple large images is limited because the data is being pushed by the server, not pulled by the client. The same is true of the LOWSRC image tag extension proposed by Netscape.
- The ALT tag provided in the HTML IMG environment allows a text string to be displayed in place of an inline graphic, but the ALT text usually cannot carry the semantic load of the image it replaces, and most advertisers would readily concede that text is no substitute for corporate logo visibility.
- The Bandwidth Conservation Society [BCS] has published a number of suggestions for content providers to minimize the size of their images without sacrificing image quality, but their techniques typically provide at most a factor of 1.5-2 in compression. This is not sufficient to bridge the order-of-magnitude gap in bandwidth between, e.g., Ethernet and consumer modems.
- A mechanism is proposed in [NPS95] for clients to negotiate for one of several document representations stored at a server. Under the proposed scheme, the server creates a fixed number of representations in advance, possibly with human guidance, and advertises to clients which representations are available. Even if this mechanism were widely deployed (which would require changes to servers), it would not satisfy clients whose connectivity would best be exploited with an intermediate-quality representation not present at the server.
- Caching [Gla93],[MLB95],[ASA+95] and prefetching reduce initial server-client latency and server-cache bandwidth requirements, but do not reduce cache-client bandwidth. We believe that distributed intelligent caching will ultimately be necessary, but not for reducing latency and bandwidth to the client.

The methods described above are ineffective at closing the bandwidth gap because they either require changes at the server (content or control), force additional interaction with the user (e.g. to explicitly select one version of a page), do not allow the user to explicitly manage the available client bandwidth, or require the user to sacrifice graphics altogether. We describe a mechanism that addresses all of these problems: real-time adaptive distillation and refinement.

How Distillation and Refinement Can Help

The Concept of Datatype-Specific Distillation

Distillation is highly lossy, real-time, datatype-specific compression that preserves most of the semantic content of a document. The concept is best illustrated by example. A large color graphic can be scaled down to one-half or one-quarter length along each dimension, reducing the total area and thereby reducing the size of the representation. Further compression is possible by reducing the color depth or colormap size. The resulting representation, though poorer in color and resolution than the original, is nonetheless still recognizable and therefore useful to the user. Of course there are limits to how severe a degradation of quality is possible before the image becomes unrecognizable, but as we discuss below, we have found that order-of-magnitude size reductions are possible without significantly compromising the usefulness of an image.

Our definition of distillation as *lossy* compression is independent of the specific encoding of the image. For example, GIF is a lossless image encoding format, but the distillation process throws away information in shrinking the image and quantizing its colormap.

As another example, a PostScript text document can be distilled by extracting the text corresponding to document content and analyzing the text corresponding to formatting information in order to glean clues about the document structure. These clues can be used to compose a "plaintext-plus" version of the document, in which, for example, chapter headings are in all caps and centered, subsection headings are set off by blank lines, etc. The distilled representation is impoverished with respect to the original document, but contains enough semantic information to be useful to the user. Adobe Systems' *Distiller Pro* package (not to be confused with our use of the term "distillation") performs a similar function, constructing a portable document format (PDF) file from a PostScript file.

Clearly, distillation techniques must be datatype-specific, because the specific properties of a document that can be exploited for semantic-preserving compression vary widely among data types. We say "type" as opposed to "subtype" (in the MIME sense), since, for example, the techniques used for image distillation apply equally well regardless of whether the source image is in GIF or JPEG format.

Refinement

Although a distilled image can be a useful representation of the original, in some cases the user may want to see the full content of the original. More commonly, the user may want to see the full content of *some part* of the original; for instance, zooming in on a section of a graphic, or rendering a particular page containing PostScript text and figures without having to render the preceding pages.

We use the term *refinement* to refer to the process of fetching some part (possibly all) of a source document at increased quality. We can define a *refinement space* for a given datatype, whose axes correspond to the properties of the datatype exploited by the corresponding distillation technique. For example, some obvious axes for still graphics are scale (as a fraction of the original) and color depth. The source image corresponds to the tuple <1,1,...,1> in refinement space. Distillation and refinement can then be thought of as parameterized mappings between points in refinement space. The example interface by which a user specifies a desired refinement is application-specific; for example, using a mouse to select a subregion of an image for zooming.

Refinement space for a given datatype may be discrete or continuous. For example, the pixel-dimension refinement axis is (nearly) continuous, but for distilling rich text, we may be able to identify only a relatively small fixed number of intermediate quality representations. For PostScript, these would likely consist of "plain text" (ASCII only with minimal formatting clues), structured rich text (such as PDF or HTML), and original PostScript.

Trading Cycles for Bandwidth

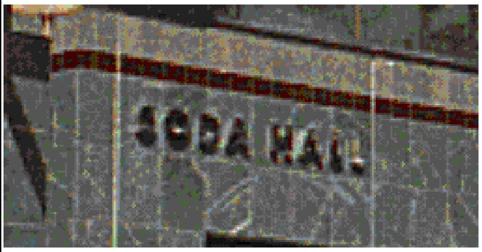
Because distillation can be performed in real time, it eliminates the need for servers to maintain multiple intermediate-quality representations of a document: Any desired intermediate representation can be created on demand using an appropriate distiller. The computing resources necessary for real-time distillation are becoming cheaper and more plentiful, and we have found that at least certain kinds of distillation can be done almost trivially in real time using modest desktop-PC hardware. Distillation and refinement allow us to trade cycles, which are cheap and plentiful, for bandwidth, which is expensive and scarce.

Using Refinement for Bandwidth Management

As an example of refinement in action, consider a user who has downloaded the image of Soda Hall in the figure below to her laptop computer using a 28.8 modem. The 300x200 image, which occupies 17K bytes and contains 16 colors, was obtained by distilling the original which has pixel dimensions 880x610, contains 249 colors, and occupies 503K bytes. Although the distilled image is clearly recognizable as the building, due to the degradation of quality the writing on the building is unreadable. The user can specify a refinement of the subregion containing the writing, which can then be viewed at full resolution and color depth, as shown below. The refinement requires 15K bytes to represent.

Distilled to 320x200 (17KB)

Click here to see original image (503K GIF)



Refinement of writing on building (15KB)

Notice how distillation and refinement have been used to explicitly manage the limited bandwidth available to the user. The distilled image and refinement together occupy only a fraction of the size of the original. The total bandwidth required to transmit them is a fraction of what would have been required to transmit the original, which might have been too large to view on the user’s screen anyway. The process of distilling the original to produce the smaller representation took about 6 seconds of wall clock time on a lightly loaded SPARC-20 workstation; the process of extracting the subregion from the original for refinement took less than 1 second.

Optimizing for a Target Display

Some notebook computers and PDA’s have smaller screens and can display fewer colors or grays than their desktop counterparts, in addition to suffering from limited bandwidth. For such devices, we would like to use the scarce bandwidth for transmitting a distilled representation of higher resolution, rather than using it for transmission of color information in excess of the client’s display capability. Intelligent distillation will scale the source image down to reasonable dimensions for the client display, and preserve only the color information that the client can display. Distillation thus allows bandwidth to be managed in a way that exploits the client’s strengths and limitations. The following table gives a sampling of computing devices with typical display and bandwidth characteristics, with the minimum latencies in minutes and seconds to transfer 5K, 50K and 200K bytes. These numbers serve as zeroth-order approximations for a small inline image, a large inline image, and the total amount of inline image data on a page, respectively.

Device	CPU/MHz	Typ. bandwidth (bits/s)	Display size	Minimum xmit latency, 5K/50K/200K bytes
Apple Newton	ARM 610/20	2400	320x240, 1-bit	0:17/2:50/11:20
Sony MagicLink	Motorola 68340/20	14.4K	480x320, 2-bit gray	0:03/0:30/1:20
Typical notebook PC	Intel or PPC/60-100	28.8K wireline, 9600 cellular	640x480 to 800x600, 8-bit color	0:02/0:15/0:60 wireline, 0:04/0:42/2:48 cellular
Typical desktop PC	Intel or PPC/60-120	56K ISDN, 10M Ethernet	640x480 to 1024x768, 16-bit color	0:01/0:07/0:29

Optimizing for Rendering on Impoverished Devices

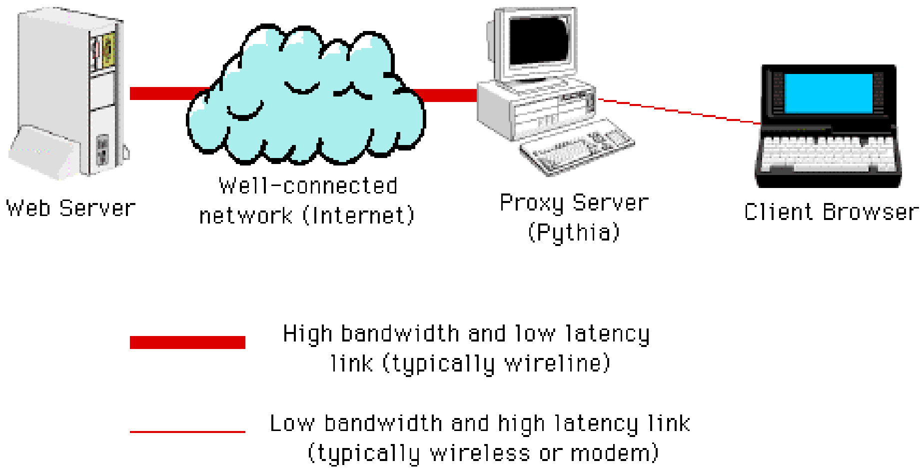
Some devices, particularly PDA's, have limited onboard computing power and understand only a small number of image formats. It would be painful and slow, for example, for an Apple Newton to receive a GIF image and transcode it to PICT, its native graphics format, for display on the screen. Instead, this transcoding can be done on a more capable desktop workstation as part of the distillation process,*before* the image is sent to the client. The idea of using transcoding to address client limitations has been explored in the Wireless World Wide Web experiment performed at DEC WRL [Bar95].

An Implemented HTTP Proxy Based on Real-Time Distillation

We have shown that distillation and refinement provide the user with a powerful mechanism for management of limited bandwidth, without completely sacrificing bandwidth-intensive nontextual (or richtext) content. Since such a mechanism is sorely needed on the WWW, we have implemented an HTTP proxy [LA94] based on real-time distillation and refinement. We have observed that using our proxy makes Web surfing with a modem much more bearable, and makes Web surfing over metropolitan-area wireless feasible. (Our work on distillation and refinement was originally done in the context of wireless mobile computing.)

Mosaic, Netscape, and other popular WWW browsers allow the user to designate a particular host as a *proxy* for HTTP requests. Rather than fetching a URL directly from the appropriate server, the fetch request is passed on to the proxy. The proxy obtains the document from the server on the client's behalf, and forwards it to the client. The proxy mechanism was originally included to allow users behind a corporate firewall to access the WWW via a proxy that had "one foot on either side" of the firewall. Our proxy, Pythia* , is intended to run near the logical boundary between well-connectedness and poorly-connectedness.

As a first-order approximation, if we take the majority of the wired Internet to be well-connected and consider a client using PPP or SLIP to be poorly-connected, Pythia can run anywhere inside the wired Internet. The architecture of a "proxied" WWW service is shown schematically below.



The idea of placing a proxy at this boundary has also been explored in the LBX (Low Bandwidth X) project, on which the Berkeley InfoPad's [BBB+94] "split X" server is based. The idea of using a proxy to transcode data on the fly was discussed in [BMM95].

Statistical Models for Real-Time Distillation

Pythia works by modeling the running time and achieved compression of distillation algorithms for various data formats. For example, given an input GIF or JPEG and a color quantization factor and scale, the model is used to predict how long the distillation will take and approximately how much compression will be achieved. Our current models provide a ballpark first cut for estimating compression and latency, though significant deviations from the model prediction are observed in a substantial fraction of cases. We expect the refinement of this model to be a focus of continuing research.

Pythia uses the model to meet user-specified bounds on inline image size (and therefore latency) while surfing the WWW. For example, suppose the user is using a 28.8 modem and has specified a maximum latency of 5 seconds per inline image, and Pythia encounters an inline image that is 40 Kbytes in size. The maximum traffic that can be carried in 5 seconds at 28.8Kbits/sec is about 5.6 Kbytes, so Pythia calculates the distillation parameters necessary to produce a representation of the image that is about 5.6 Kbytes in size. In practice, the bound on the image size will be tighter, since Pythia must account for the additional latency introduced by the distillation process itself.

The first graph below shows a breakdown of server fetch, distillation, and transmission times for a small sample of images found on Berkeley WWW servers, as transmitted to a client on a conventional 14.4 modem using PPP.

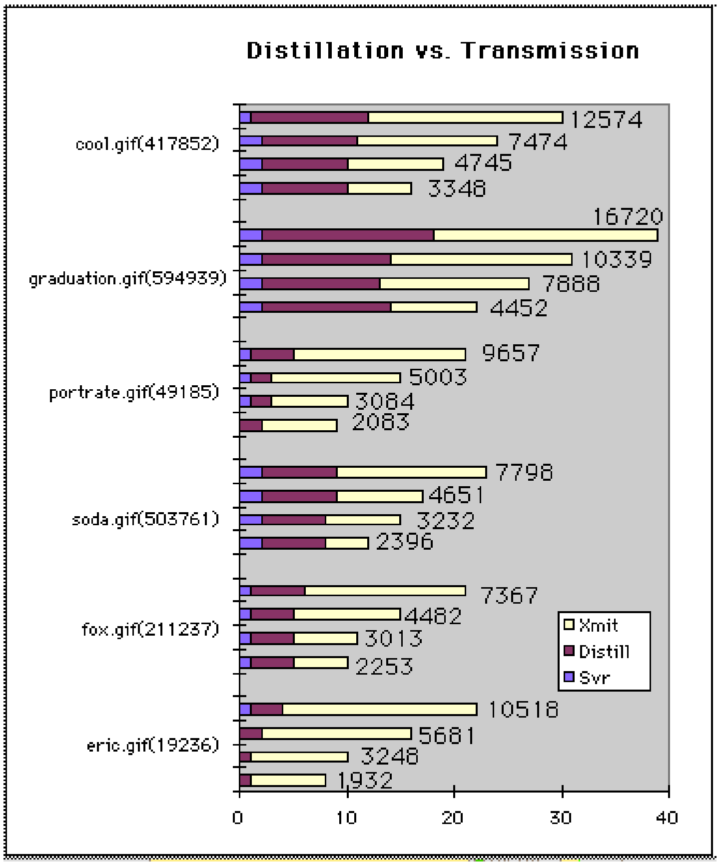
- The number in parentheses following the name of each image is the size of the source image, in bytes, as stored on the server.

- Each bar shows the breakdown of total client latency to receive the image: time for Pythia to retrieve the image from server (*svr*), time to distill the image (*distill*), and time to transmit the distilled image to the client (*xmit*). TCP roundtrip latencies between the client and proxy are absorbed into this last component.

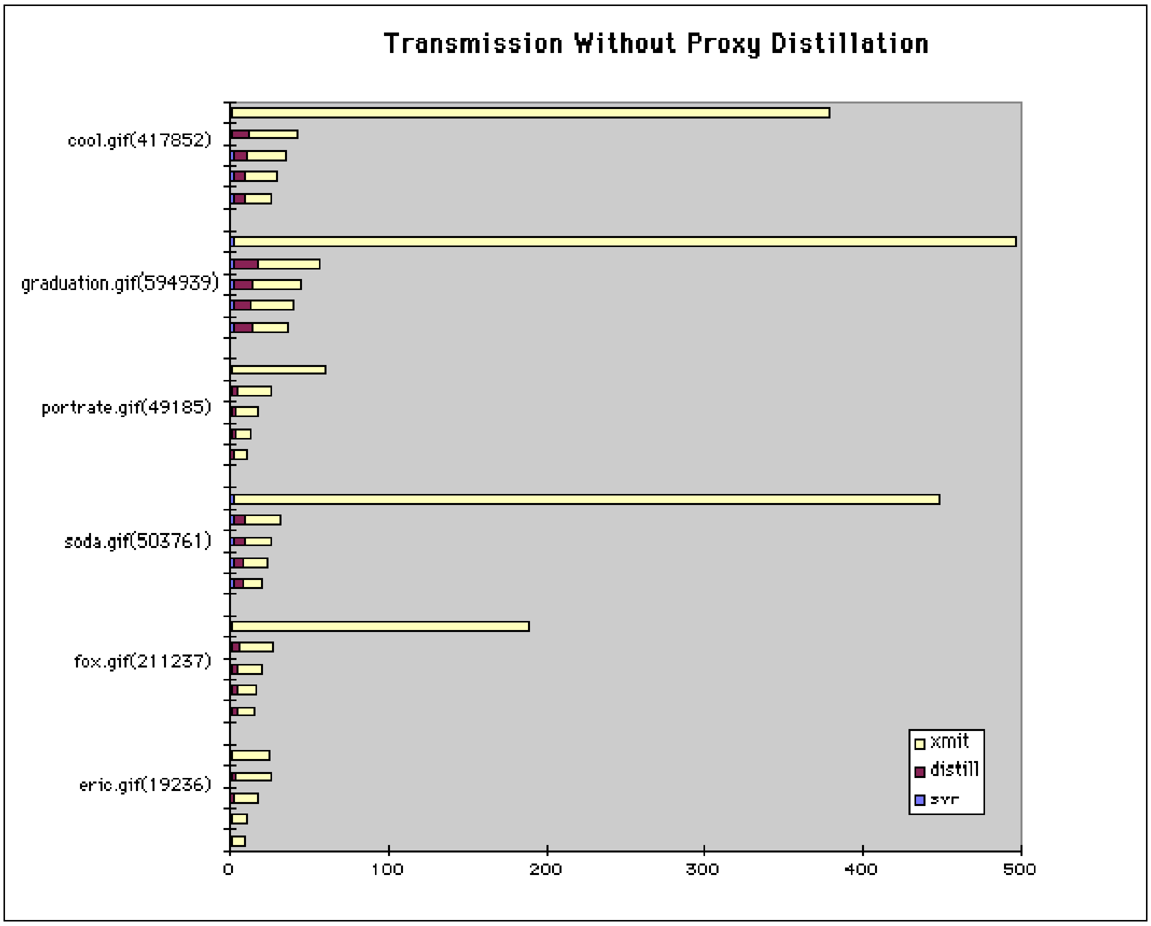
- The four different bars for each image represent four different Pythia user profiles, varying in the aggressiveness of distillation. In each case, the final size of the distilled representation is shown as a number to the right of each bar.

For example, the image *cool.gif* , whose undistilled size is 417,852 bytes, was delivered to the client in a distilled form of 12574 bytes. The delivery latency included about 2 seconds for Pythia to get the image from the local server, about 8 seconds to distill it, and about 32 seconds to send the distilled version to the client. The other three bars for *cool.gif* show similar latency measures for three different distilled representations, of 7474, 4545, and 3348 bytes. As the graph shows, distillation ranges from less than 1 second up to a couple of tens of seconds, on a lightly-loaded SPARCstation-20.

The unusually high transmit latencies for the small (3K) images reflect a highly loaded PPP gateway that typically adds up to .5 seconds each way per roundtrip; unfortunately, such performance is not unusual when using PPP-based ISP's.



The second graph below shows the raw transmit latencies, including TCP and PPP-gateway overhead, for transmitting the *undistilled* originals of the above images to the client using the same modem connection. For reference, the bars in the above graph are also reproduced below. As the graph shows, the total perceived latency at the client is reduced by approximately an order of magnitude when Pythia is used, even though the distillation process takes measurable time.



Pythia's User Interface

Pythia maintains a user profile associated with the IP address of each HTTP client that contacts it, and provides a mechanism for users to "register" if their IP address changes (as is the case, e.g., with ISP's that assign IP addresses dynamically when users dial up). The profile, which is user-settable via an HTML form, encodes the user's connection speed, some characteristics of the user's display device, and various other options. The display information is useful because exploiting the display's constraints may allow Pythia to produce a better representation of some graphic within the same latency bound (e.g. it will permit color information to be traded for resolution).

To use Pythia, a user specifies Pythia's host and port as the HTTP Proxy in the Preferences dialog of most browsers, and fills out the profile form. Pages delivered by Pythia look like their "unproxied" counterparts, except that some of the inline images have been distilled. Bounding boxes of the original images are preserved, to accommodate pages where the layout has been fine-tuned for viewing on a particular browser.

The user can request a refinement of a distilled image by following an HTTP link next to each distilled image. Depending on the user's profile, the original image will be fetched and displayed on a page by itself, or it will be refined "in place" and the current page re-rendered around it. Pythia adds these "fetch refinement" links to the HTML text on the fly, as described in the next section.

For example, here is a portion of a web page before refinement, and the same page after the user has refined the inline image.

If image dimension hints are supplied in the source page's IMG tag, Pythia passes them on to the client; however, Pythia cannot add dimension hints itself, since at the time it sees the referencing HTML tag, it cannot know what the actual image dimensions will be without prefetching part of the image, which might add unacceptable latency. We are experimenting with this tradeoff to determine which method will provide a higher perceived quality of service to the user.

Pythia also translates PostScript to HTML, using software developed in part by DEC SRC [McJ]. This distillation typically results in a reduction of 5-7x for PostScript text, and has the additional advantage that the text can be rendered on clients for which PostScript previewing is awkward, such as PC's running Windows. This is an example of distillation that provides both of the orthogonal benefits mentioned previously: content size reduction and optimization for rendering on the target display.

Implementation and Performance

URL Munging and HTML Modification

When Pythia returns HTML text to the client, the text is scanned for IMG tags. For each such tag found, Pythia does two things:

- Modify the URL of the image source, so that when the image is requested, Pythia will recognize the tag as belonging to an inline image. This could be omitted if the HTTP "Referer:" field was filled in consistently by all browsers.

- Insert a hyperlink immediately following the image tag. This new link will contain a URL, based on that of the original image, that will cue Pythia to deliver an undistilled representation of the image. If the user has elected to have Pythia re-render the entire page with the image refined in place, the URL is instead based on the name of the referring page concatenated with a bit vector in which each bit position indicates whether the corresponding inline image should be distilled or not.

A more detailed explanation of the munging mechanism, for those who are interested, can be found on Pythia's home page.

Exploiting URL-Level Parallelism

Pythia's internal architecture is modular: "distillation servers" for new datatypes can be easily added. The distillation server need only provide a statistical performance model and the functionality to read the source document and write out a distilled representation given some parameters. Although a distiller can be launched as a standard shell pipeline, we also provide a standard makefile and front-end for building somewhat more efficient distillation servers based on Berkeley sockets.

Distillers can run on the same physical machine as Pythia or on different machines. Pythia keeps track of which distillers are running on which machines, and attempts to do simple load balancing across them. The Berkeley Network of Workstations (NOW) project [ACP+95] has provided a job-queue interface for harvesting idle cycles on machines in the NOW; we are retrofitting Pythia to use this mechanism to spawn and destroy distillers dynamically as NOW resource levels fluctuate.

Refinement Cache

After distilling and forwarding an image to the client, Pythia caches a copy of the image locally to minimize latency in case the client requests refinement. The cache is a very simple fully-associative size-limited LRU whose keys are URL's and whose data fields are the original image data.

Implementation Status, Limitations, and Future Work

(You can click here to try Pythia live.)

Pythia's "front end" currently runs on a lightly-loaded SPARCstation-20 and distributes image distillers to 2-4 other workstations on the same subnet. Because it is a prototype and is not consistently running, its user community is limited to only about a dozen users, and it is rare to see more than two users at one time. Under these light conditions, the workstation console does not suffer noticeable performance degradation due to Pythia usage.

Since Pythia can farm out distillation work to other workstations, the cycles required to perform distillation for clients do not constitute a performance bottleneck. Instead, like HTTP servers, the limiting factor is the single pipe in and out of the "front end" that receives HTTP requests (i.e. the process listening on the TCP port designated as the HTTP Proxy). The current implementation of Pythia is in unoptimized Perl; translation to C should increase the number of requests that can be handled by the front end per unit time. Current usage patterns indicate that this metric will not be a bottleneck when only a few users are served by a single front end. We are planning joint work with the Berkeley Office of Telecommunications Services, which provides dial-up PPP and SLIP services to about 6,000 subscribers on the Berkeley campus, to allow them to provide web proxy service as part of their subscription package. This experiment will stress Pythia and allow us to explore various strategies for scaling the front-end using a "magic router" based on fast IP packet interposition [And95] .

Pythia currently performs a Unix *fork()* to handle each new HTTP request. It is well known that the latency of this operation is substantial [Ous90] . Future versions of Pythia will be multithreaded rather than relying on process-level parallelism, and idle worker threads rather than forked processes will handle multiple incoming requests.

The bandwidth of the client connection currently must be filled in on the User Preferences HTML form. Pythia takes the user's word for this quantity, rather than attempting to measure the quality of the connection directly (e.g., by estimating the latency between the transmission of HTML text to the client and reception of an HTTP request for an embedded image). The short lifetimes of HTTP TCP connections and the overhead of TCP slow start make it difficult to measure end-to-end bandwidth accurately.

Pythia cannot hide server-to-proxy latency, though it can mitigate it by distillation and caching. Pythia's distillation estimates are based solely on the proxy-to-client bandwidth stated by each user.

Pythia is fault-tolerant with respect to distillers: it will reap distillers that are killed due to NOW load balancing and will continue to function with degraded performance. If Pythia's front-end crashes, however, the user will see an error that the proxy has stopped accepting connections. We currently do not have a fault-tolerance strategy for the front-end.

Because Pythia works by munging URL's, it may cause cache inconsistency at the client. For example, after a user stops using Pythia, that user's cache will contain some entries whose keys (URL's) are the Pythia-modified URL's rather than the original source URL's. Flushing the client cache fixes this problem at significant inconvenience to the user. URL munging is necessary because HTTP provides no way for Pythia to maintain "session state" describing which inlines on a given page have been distilled and which have not. To circumvent this limitation, Pythia encodes this information into the URL's passed back and forth between client and proxy. HTTP-NG will include some notion of session control, which should allow us to maintain the appropriate state without resorting to URL-munging.

As part of our wireless and mobile computing effort, we are developing a variant of Pythia with a richer client API for building network-adaptive applications. This API will allow negotiation of a wider variety of datatypes, an environment in which agents can run, and distillation services for continuous-media streams such as MPEG (an implemented example is [AMZ95]).

Conclusions

Pythia provides three important orthogonal benefits to WWW clients:

- Real-time distillation and refinement, guided by statistical models, allow the user to bound latency and exercise explicit control over bandwidth that may be scarce and expensive (e.g. metered cellular phone service).
- Transcoding to a representation understood directly by the client may improve rendering on the client or result in a representation that can be transmitted more efficiently.
- Knowledge of client display constraints allows content to be optimized for rendering on the client.

Users have commented that even the prototype version of Pythia provides a qualitative increase of about 5x when surfing the WWW over PPP with a 14.4 modem. These are the same users that previously turned image loading off completely in order to make surfing bearable. With the continued growth of the WWW, the benefits afforded by proxied services like Pythia will represent increasingly significant added value to end users and content providers alike. Pythia is the first fruit of a comprehensive research agenda aimed at implementing and deploying such services.

References

[ACP+95] Thomas E. Anderson, David E. Culler, David A. Patterson, et al. *The Case for a Network of Workstations*. IEE Micro (to appear).

[AMZ95] Elan Amir, Steve McCanne, Hui Zhang. *An Application-Level Video Gateway* . Proc. ACM Multimedia 95, San Francisco, Nov. 1995.

[And95] Eric Anderson. *An Application of Fast Packet Interposing: The Magic Router.* http://www.cs.berkeley.edu/~eanders/262/

[ASA+95] Marc Abrams et al. *Caching Proxies: Limitations and Potentials*. Fourth International WWW Conference, Boston, MA, Dec. 1995.

[Bar95] Joel F. Bartlett. *Experience with a Wireless World Wide Web Client.* IEEE COMPCON 95, San Francisco, March 1995.

[BBB+94] B. Barringer, T. Burd, F. Burghardt, et al. *InfoPad: System Design for Portable Multimedia Access.* Proc. Calgary Wireless 94 Conference, July 1994.

[BCS] Bandwidth Conservation Society home page. http://www.infohiway.com/faster/

[BMM95] Charles Brooks, Murray S. Mazer, Scott Meeks, Jim Miller. *Application-Specific Proxy Servers as HTTP Stream Transducers*. Fourth International World Wide Web Conference, Boston, MA, Dec. 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/56/.

[Gla93] Steven Glassman. *A Caching Relay for the World Wide Web.* Computer Networks and ISDN Systems 27(2), Nov. 1994. Also appeared in Proc. First Int'l World Wide Web Conference.

[LA94] A. Luotonen and K. Altis. *World-Wide Web Proxies.* http://www.w3.org/hypertext/WWW/Proxies/.

[McJ] Paul McJones, personal communication.

[MLB95] Radhika Malpani, Jacob Lorch, David Berger. *Making World Wide Web Caching Servers Cooperate.* Fourth International World Wide Web Conference, Boston, MA, Dec. 1995.

[NPS95] Brian D. Noble, Morgan Price, and M. Satyanarayanan. *A Programming Interface for Application-Aware Adaptation in Mobile Computing.* 1995 Mobile and Location-Independent Computing Symposium.

[Ous90] John K. Ousterhout. *Why Aren't Operating Systems Getting Faster As Fast As Hardware?* USENIX Summer Conference Proceedings, June 1990.

[PG93] Venkata Padmanabhan and Jeffrey C. Mogul. *Improving HTTP Latency*. Computer Networks and ISDN Systems 28(1), Dec. 1995. http://www.cs.berkeley.edu/~padmanab/papers/www_fall94.ps

[Xinside] X Inside Inc. home page. http://www.xinside.com

---

* In Greek mythology, Pythia was the intermediary who carried a pilgrim's request to the Oracle at Delphi and conveyed the reply back to the pilgrim.