# Key Differences between HTTP/1.0 and HTTP/1.1


Balachander Krishnamurthy
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932
bala@research.att.com

Jeffrey C. Mogul
Western Research Lab
Compaq Computer Corp.
250 University Avenue
Palo Alto, CA 94301
mogul@pa.dec.com

David M. Kristol
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974 USA
dmk@bell-labs.com



**Abstract**

The HTTP/1.1 protocol is the result of four years of discussion and debate among a broad group of Web researchers and developers. It improves upon its phenomenally successful predecessor, HTTP/1.0, in numerous ways. We discuss the differences between HTTP/1.0 and HTTP/1.1, as well as some of the rationale behind these changes.


## 1 Introduction

By any reasonable standard, the HTTP/1.0 protocol has been stunningly successful. As a measure of its popularity, HTTP accounted for about 75% of Internet backbone traffic in a recent study[TMW97]. In spite of its success, however, HTTP/1.0 is widely understood to have numerous flaws.

HTTP/1.0 evolved from the original "0.9" version of HTTP (which is still in rare use). The process leading to HTTP/1.0 involved significant debate and experimentation, but never produced a formal specification. The HTTP Working Group (HTTP-WG) of the Internet Engineering Task Force (IETF) produced a document (RFC1945) [BLFF96] that described the "common usage" of HTTP/1.0, but did not attempt to create a formal standard out of the many variant implementations. Instead, over a period of roughly four years, the HTTP-WG developed an improved protocol, known as HTTP/1.1. The HTTP/1.1 specification [FGM+98] is soon to become an IETF Draft Standard. Recent versions of some popular agents (MSIE, Apache) claim HTTP/1.1 compliance in their requests or responses, and many implementations have been tested for interoperable compliance with the specification [Mas98, NG98].

The HTTP/1.1 specification states the various requirements for clients, proxies, and servers. However, additional context and rationales for the changed or new features can help developers understand the motivation behind the changes, and provide them with a richer understanding of the protocol. Additionally, these rationales can give implementors a broader feel for the pros and cons of individual features.

In this paper we describe the major changes between the HTTP/1.0 and HTTP/1.1 protocols. The HTTP/1.1 specification is almost three times as long as RFC1945, reflecting an increase in complexity, clarity, and specificity. Even so, numerous rules are implied by the HTTP/1.1 specification, rather than being explicitly stated. While some attempts have been made to document the differences between HTTP/1.0 and HTTP/1.1 ([Mar97, Yap97], Section 19.6.1 of [FGM+98]), we know of no published analysis that covers major differences and the rationale behind them, and that reflects the most recent (and probably near-final) revision of the HTTP/1.1 specification. Because the HTTP-WG, a large and international group of researchers and developers, conducted most of its discussions via its mailing list, the archive of that list [CLF98] documents the history of the HTTP/1.1 effort. But that archive contains over 8500 messages, rendering it opaque to all but the most determined protocol historian.

We structure our discussion by (somewhat arbitrarily) dividing the protocol changes into nine major areas:

1. Extensibility
2. Caching
3. Bandwidth optimization
4. Network connection management
5. Message transmission
6. Internet address conservation
7. Error notification
8. Security, integrity, and authentication
9. Content negotiation

We devote a section to each area, including the motivation for changes and a description of the corresponding new features.

## 2 Extensibility

The HTTP/1.1 effort assumed, from the outset, that compatibility with the installed base of HTTP implementations (including many that did not conform with [BLFF96]) was mandatory. It seemed unlikely that most software vendors or Web site operators would deploy systems that failed to interoperate with the millions of existing clients, servers, and proxies.

Because the HTTP/1.1 effort took over four years, and generated numerous interim draft documents, many implementors deployed systems using the "HTTP/1.1" protocol version before the final version of the specification was finished. This created another compatibility problem: the final version had to be substantially compatible with these pseudo-HTTP/1.1 versions, even if the interim drafts turned out to have errors in them.

These absolute requirements for compatibility with poorly specified prior versions led to a number of idiosyncrasies and non-uniformities in the final design. It is not possible to understand the rationale for all of the HTTP/1.1 features without recognizing this point.

The compatibility issue also underlined the need to include, in HTTP/1.1, as much support as possible for future extensibility. That is, if a future version of HTTP were to be designed, it should not be hamstrung by any additional compatibility problems.

Note that HTTP has always specified that if an implementation receives a header that it does not understand, it must ignore the header. This rule allows a multitude of extensions without any change to the protocol version, although it does not by itself support all possible extensions.

### 2.1 Version numbers

In spite of the confusion over the meaning of the "HTTP/1.1" protocol version token (does it imply compatibility with one of the interim drafts, or with the final standard?), in many cases the version number in an HTTP message can be used to deduce the capabilities of the sender. A companion document to the HTTP specification [MFGN97] clearly specified the ground rules for the use and interpretation of HTTP version numbers.

The version number in an HTTP message refers to the hop-by-hop sender of the message, not the end-to-end sender. Thus the message's version number is directly useful in determining hop-by-hop message-level capabilities, but not very useful in determining end-to-end capabilities. For example, if an HTTP/1.1 origin server receives a message forwarded by an HTTP/1.1 proxy, it cannot tell from that message whether the ultimate client uses HTTP/1.0 or HTTP/1.1.

For this reason, as well as to support debugging, HTTP/1.1 defines a `Via` header that describes the path followed by a forwarded message. The path information includes the HTTP version numbers of all senders along the path and is recorded by each successive recipient. (Only the last of multiple consecutive HTTP/1.0 senders will be listed, because HTTP/1.0 proxies will not add information to the `Via` header.)

### 2.2 The OPTIONS method

HTTP/1.1 introduces the `OPTIONS` method, a way for a client to learn about the capabilities of a server without actually requesting a resource. For example, a proxy can verify that the server complies with a specific version of the protocol. Unfortunately, the precise semantics of the `OPTIONS` method were the subject of an intense and unresolved debate, and we believe that the mechanism is not yet fully specified.

### 2.3 Upgrading to other protocols

In order to ease the deployment of incompatible future protocols, HTTP/1.1 includes the new `Upgrade` request-header. By sending the `Upgrade` header, a client can inform a server of the set of protocols it supports as an alternate means of communication. The server may choose to switch protocols, but this is not mandatory.

## 3 Caching

Web developers recognized early on that the caching of responses was both possible and highly desirable. Caching is effective because a few resources are requested often by many users, or repeatedly by a given user. Caches are employed in most Web browsers and in many proxy servers; occasionally they are also employed in conjunction with certain origin servers. Web caching products, such as Cisco's cache engine [Cis] and Inktomi's Traffic Server [Ink] (to name two), are now a major business.

Many researchers have studied the effectiveness of HTTP caching [KLM97, DFKM97, ASA$^+$95, IC97]. Caching improves user-perceived latency by eliminating the network communication with the origin server. Caching also reduces bandwidth consumption, by avoiding the transmission of unnecessary network packets. Reduced bandwidth consumption also indirectly reduces latency for uncached interactions, by reducing network congestion. Finally, caching can reduce the load on origin servers (and on intermediate proxies), further improving latency for uncached interactions.

One risk with caching is that the caching mechanism might not be "semantically transparent": that is, it might return a response different from what would be returned by direct communication with the origin server. While some applications can tolerate non-transparent responses, many Web applications (electronic commerce, for example) cannot.

### 3.1 Caching in HTTP/1.0

HTTP/1.0 provided a simple caching mechanism. An origin server may mark a response, using the `Expires` header, with a time until which a cache could return the response without violating semantic transparency. Further, a cache may check the current validity of a response using what is known as a *conditional request*: it may include an `If-Modified-Since` header in a request for the resource, specifying the value given in the cached response's `Last-Modified` header. The server may then either respond with a 304 (Not Modified) status code, implying that the cache entry is valid, or it may send a normal 200 (OK) response to replace the cache entry.

HTTP/1.0 also included a mechanism, the `Pragma: no-cache` header, for the client to indicate that a request should not be satisfied from a cache.

The HTTP/1.0 caching mechanism worked moderately well, but it had many conceptual shortcomings. It did not allow either origin servers or clients to give full and explicit instructions to caches; therefore, it depended on a body of heuristics that were not well-specified. This led to two problems: incorrect caching of some responses that should not have been cached, and failure to cache some responses that could have been cached. The former causes semantic problems; the latter causes performance problems.

### 3.2 Caching in HTTP/1.1

HTTP/1.1 attempts to clarify the concepts behind caching, and to provide explicit and extensible protocol mechanisms for caching. While it retains the basic HTTP/1.0 design, it augments that design both with new features, and with more careful specifications of the existing features.

In HTTP/1.1 terminology, a cache entry is *fresh* until it reaches its expiration time, at which point it becomes *stale*. A cache need not discard a stale entry, but it normally must revalidate it with the origin server before returning it in response to a subsequent request. However, the protocol allows both origin servers and end-user clients to override this basic rule.

In HTTP/1.0, a cache revalidated an entry using the `If-Modified-Since` header. This header uses absolute timestamps with one-second resolution, which could lead to caching errors either because of clock synchronization errors, or because of lack of resolution. Therefore, HTTP/1.1 introduces the more general concept of an opaque cache validator string, known as an *entity tag*. If two responses for the same resource have the same entity tag, then they must (by specification) be identical. Because an entity tag is opaque, the origin server may use any information it deems necessary to construct it (such as a fine-grained timestamp or an internal database pointer), as long as it meets the uniqueness requirement. Clients may compare entity tags for equality, but cannot otherwise manipulate them. HTTP/1.1 servers attach entity tags to responses using the `ETag` header.

HTTP/1.1 includes a number of new conditional request-headers, in addition to `If-Modified-Since`. The most basic is `If-None-Match`, which allows a client to present one or more entity tags from its cache entries for a resource. If none of these matches the resource's current entity tag value, the server returns a normal response; otherwise, it may return a 304 (Not Modified) response with an `ETag` header that indicates which cache entry is currently valid. Note that this mechanism allows the server to cycle through a set of possible responses, while the `If-Modified-Since` mechanism only generates a cache hit if the most recent response is valid.

HTTP/1.1 also adds new conditional headers called `If-Unmodified-Since` and `If-Match`, creating other forms of preconditions on requests. These preconditions are useful in more complex situations; in particular, see the discussion in Section 4.1 of Range requests.

### 3.3 The Cache-Control header

In order to make caching requirements more explicit, HTTP/1.1 adds the new `Cache-Control` header, allowing an extensible set of cache-control *directives* to be transmitted in both requests and responses. The set defined by HTTP/1.1 is quite large, so we concentrate on several notable members.

Because the absolute timestamps in the HTTP/1.0 `Expires` header can lead to failures in the presence of clock skew (and observations suggest that serious clock skew is common), HTTP/1.1 can use relative expiration times, via the `max-age` directive. (It also introduces an `Age` header, so that caches can indicate how long a response has been sitting in caches along the way.)

Because some users have privacy requirements that limit caching beyond the need for semantic transparency, the `private` and `no-store` directives allow servers and clients to prevent the storage of some or all of a response. However, this does not guarantee privacy; only cryptographic mechanisms can provide true privacy.

Some proxies transform responses (for example, to reduce image complexity before transmission over a slow link [FGBA96]), but because some responses cannot be blindly transformed without losing information, the `no-transform` directive may be used to prevent transformations.

### 3.4 The Vary header

A cache finds a cache entry by using a key value in a lookup algorithm. The simplistic caching model in HTTP/1.0 uses just the requested URL as the cache key. However, the content negotiation mechanism (described in Section 10) breaks this model, because the response may vary not only based on the URL, but also based on one or more request-headers (such as `Accept-Language` and `Accept-Charset`).

To support caching of negotiated responses, and for future extensibility, HTTP/1.1 includes the `Vary` response-header. This header field carries a list of the relevant *selecting* request-header fields that participated in the selection of the response variant. In order to use the particular variant of the cached response in replying to a subsequent request, the selecting request-headers of the new request must exactly match those of the original request.

This simple and elegant extension mechanism works for many cases of negotiation, but it does not allow for much intelligence at the cache. For example, a smart cache could, in principle, realize that one request header value is compatible with another, without being equal. The HTTP/1.1 development effort included an attempt to provide so-called "transparent content negotiation" that would allow caches some active participation, but ultimately no consensus developed, and this attempt [HM98b, HM98a] was separated from the HTTP/1.1 specification.

## 4 Bandwidth optimization

Network bandwidth is almost always limited. Both by intrinsically delaying the transmission of data, and through the added queueing delay caused by congestion, wasting bandwidth increases latency. HTTP/1.0 wastes bandwidth in several ways that HTTP/1.1 addresses. A typical example is a server's sending an entire (large) resource when the client only needs a small part of it. There was no way in HTTP/1.0 to request partial objects. Also, it is possible for bandwidth to be wasted in the forward direction: if a HTTP/1.0 server could not accept large requests, it would return an error code *after* bandwidth had already been consumed. What was missing was the ability to negotiate with a server and to ensure its ability to handle such requests before sending them.

### 4.1 Range requests

A client may need only part of a resource. For example, it may want to display just the beginning of a long document, or it may want to continue downloading a file after a transfer was terminated in mid-stream. HTTP/1.1 *range requests* allow a client to request portions of a resource. While the range mechanism is extensible to other units (such as chapters of a document, or frames of a movie), HTTP/1.1 supports only ranges of bytes. A client makes a range request by including the `Range` header in its request, specifying one or more contiguous ranges of bytes. The server can either ignore the `Range` header, or it can return one or more ranges in the response.

If a response contains a range, rather than the entire resource, it carries the 206 (Partial Content) status code. This code prevents HTTP/1.0 proxy caches from accidentally treating the response as a full one, and then using it as a cached response to a subsequent request. In a range response, the `Content-Range` header indicates the offset and length of the returned range, and the new `multipart/byteranges` MIME type allows the transmission of multiple ranges in one message.

Range requests can be used in a variety of ways, such as:

1. to read the initial part of an image, to determine its geometry and therefore do page layout without loading the entire image.

2. to complete a response transfer that was interrupted (either by the user or by a network failure); in other words, to convert a partial cache entry into a complete response.

3. to read the tail of a growing object.

Some of these forms of range request involve cache conditionals. That is, the proper response may depend on the validity of the client's cache entry (if any).

For example, the first kind (getting a prefix of the resource) might be done unconditionally, or it might be done with an `If-None-Match` header; the latter implies that the client only wants the range if the underlying object has changed, and otherwise will use its cache entry.

The second kind of request, on the other hand, is made when the client does not have a cache entry that includes the desired range. Therefore, the client wants the range only if the underlying object has *not* changed; otherwise, it wants the full response. This could be accomplished by first sending a range request with an `If-Match` header, and then repeating the request without either header if the first request fails. However, since this is an important optimization, HTTP/1.1 includes an `If-Range` header, which effectively performs that sequence in a single interaction.

Range requests were originally proposed by Ari Luotonen and John Franks [FL95], using an extension to the URL syntax instead of a separate header field. However, this approach proved less general than the approach ultimated used in HTTP/1.1, especially with respect to conditional requests.

### 4.2 Expect and 100 (Continue)

Some HTTP requests (for example, the `PUT` or `POST` methods) carry request bodies, which may be arbitrarily long. If, the server is not willing to accept the request, perhaps because of an authentication failure, it would be a waste of bandwidth to transmit such a large request body.

HTTP/1.1 includes a new status code, 100 (Continue), to inform the client that the request body should be transmitted. When this mechanism is used, the client first sends its request headers, then waits for a response. If the response is an error code, such as 401 (Unauthorized), indicating that the server does not need to read the request body, the request is terminated. If the response is 100 (Continue), the client can then send the request body, knowing that the server will accept it.

However, HTTP/1.0 clients do not understand the 100 (Continue) response. Therefore, in order to trigger the use of this mechanism, the client sends the new `Expect` header, with a value of `100-continue`. (The `Expect` header could be used for other, future purposes not defined in HTTP/1.1.)

Because not all servers use this mechanism (the `Expect` header is a relatively late addition to HTTP/1.1, and early "HTTP/1.1" servers did not implement it), the client must not wait indefinitely for a 100 (Continue) response before sending its request body. HTTP/1.1 specifies a number of somewhat complex rules to avoid either infinite waits or wasted bandwidth. We lack sufficient experience based on deployed implementations to know if this design will work efficiently.

### 4.3 Compression

One well-known way to conserve bandwidth is through the use of data compression. While most image formats (GIF, JPEG, MPEG) are precompressed, many other data types used in the Web are not. One study showed that aggressive use of additional compression could save almost 40% of the bytes sent via HTTP [MDFK97]. While HTTP/1.0 included some support for compression, it did not provide adequate mechanisms for negotiating the use of compression, or for distinguishing between end-to-end and hop-by-hop compression.

HTTP/1.1 makes a distinction between content-codings, which are end-to-end encodings that might be inherent in the native format of a resource, and transfer-codings, which are always hop-by-hop. Compression can be done either as a content-coding or as a transfer-coding. To support this choice, and the choice between various existing and future compression codings, without breaking compatibility with the installed base, HTTP/1.1 had to carefully revise and extend the mechanisms for negotiating the use of codings.

HTTP/1.0 includes the `Content-Encoding` header, which indicates the end-to-end content-coding(s) used for a message; HTTP/1.1 adds the `Transfer-Encoding` header, which indicates the hop-by-hop transfer-coding(s) used for a message.

HTTP/1.1 (unlike HTTP/1.0) carefully specifies the `Accept-Encoding` header, used by a client to indicate what content-codings it can handle, and which ones it prefers. One tricky issue is the need to support "robot" clients that are attempting to create mirrors of the origin server's resources; another problem is the need to interoperate with HTTP/1.0 implementations, for which `Accept-Encoding` was poorly specified.

HTTP/1.1 also includes the `TE` header, which allows the client to indicate which transfer-codings are acceptable, and which are preferred. Note that one important transfer-coding, `Chunked`, has a special function (not related to compression), and is discussed further in Section 6.1.

## 5 Network connection management

HTTP almost always uses TCP as its transport protocol. TCP works best for long-lived connections, but the original HTTP design used a new TCP connection for each request, so each request incurred the cost of setting up a new TCP connection (at least one round-trip time across the network, plus several overhead packets). Since most Web interactions are short (the median response message size is about 4 Kbytes [MDFK97]), the TCP connections seldom get past the "slow-start" region [Jac88] and therefore fail to maximize their use of the available bandwidth.

Web pages frequently have embedded images, sometimes many of them, and each image is retrieved via a separate HTTP request. The use of a new TCP connection for each image retrieval serializes the display of the entire page on the connection-setup latencies for all of the requests. Netscape introduced the use of parallel TCP connections to compensate for this serialization, but the possibility of increased congestion limits the utility of this approach.

To resolve these problems, Padmanabhan and Mogul [PM95] recommended the use of *persistent connections* and the *pipelining* of requests on a persistent connection.

## 5.1 The Connection header

Before discussing persistent connections, we address a more basic issue. Given the use of intermediate proxies, HTTP makes a distinction between the end-to-end path taken by a message, and the actual hop-by-hop connection between two HTTP implementations.

HTTP/1.1 introduces the concept of hop-by-hop headers: message headers that apply only to a given connection, and not to the entire path. (For example, we have already described the hop-by-hop `Transfer-Encoding` and `TE` headers.) The use of hop-by-hop headers creates a potential problem: if such a header were to be forwarded by a naive proxy, it might mislead the recipient.

Therefore, HTTP/1.1 includes the `Connection` header. This header lists all of the hop-by-hop headers in a message, telling the recipient that these headers must be removed from that message before it is forwarded. This extensible mechanism allows the future introduction of new hop-by-hop headers; the sender need not know whether the recipient understands a new header in order to prevent the recipient from forwarding the header.

Because HTTP/1.0 proxies do not understand the `Connection` header, however, HTTP/1.1 imposes an additional rule. If a `Connection` header is received in an HTTP/1.0 message, then it must have been incorrectly forwarded by an HTTP/1.0 proxy. Therefore, all of the headers it lists were also incorrectly forwarded, and must be ignored.

The `Connection` header may also list *connection-tokens*, which are not headers but rather per-connection boolean flags. For example, HTTP/1.1 defines the token `close` to permit the peer to indicate that it does not want to use a persistent connection. Again, the `Connection` header mechanism prevents these tokens from being forwarded.

## 5.2 Persistent Connections

HTTP/1.0, in its documented form, made no provision for persistent connections. Some HTTP/1.0 implementations, however, use a `Keep-Alive` header (described in [Fie95]) to request that a connection persist. This design did not interoperate with intermediate proxies (see Section 19.6.2 of [FGM+98]); HTTP/1.1 specifies a more general solution.

In recognition of their desirable properties, HTTP/1.1 makes persistent connections the default. HTTP/1.1 clients, servers, and proxies assume that a connection will be kept open after the transmission of a request and its response. The protocol does allow an implementation to close a connection at any time, in order to manage its resources, although it is best to do so only after the end of a response.

Because an implementation may prefer not to use persistent connections if it cannot efficiently scale to large numbers of connections or may want to cleanly terminate one for resource-management reasons, the protocol permits it to send a `Connection: close` header to inform the recipient that the connection will not be reused.

### 5.3 Pipelining

Although HTTP/1.1 encourages the transmission of multiple requests over a single TCP connection, each request must still be sent in one contiguous message, and a server must send responses (on a given connection) in the order that it received the corresponding requests. However, a client need not wait to receive the response for one request before sending another request on the same connection. In fact, a client could send an arbitrarily large number of requests over a TCP connection before receiving any of the responses. This practice, known as pipelining, can greatly improve performance [NGBS$^+$97]. It avoids the need to wait for network round-trips, and it makes the best possible use of the TCP protocol.

## 6 Message transmission

HTTP messages may carry a body of arbitrary length. The recipient of a message needs to know where the message ends. The sender can use the `Content-Length` header, which gives the length of the body. However, many responses are generated dynamically, by CGI [CGI98] processes and similar mechanisms. Without buffering the entire response (which would add latency), the server cannot know how long it will be and cannot send a `Content-Length` header.

When not using persistent connections, the solution is simple: the server closes the connection. This option is available in HTTP/1.1, but it defeats the performance advantages of persistent connections.

### 6.1 The Chunked transfer-coding

HTTP/1.1 resolves the problem of delimiting message bodies by introducing the `Chunked` transfer-coding. The sender breaks the message body into chunks of arbitrary length, and each chunk is sent with its length prepended; it marks the end of the message with a zero-length chunk. The sender uses the `Transfer-Encoding: chunked` header to signal the use of chunking.

This mechanism allows the sender to buffer small pieces of the message, instead of the entire message, without adding much complexity or overhead. All HTTP/1.1 implementations must be able to receive chunked messages.

The `Chunked` transfer-coding solves another problem, not related to performance. In HTTP/1.0, if the sender does not include a `Content-Length` header, the recipient cannot tell if the message has been truncated due to transmission problems. This ambiguity leads to errors, especially when truncated responses are stored in caches.

### 6.2 Trailers

Chunking solves another problem related to sender-side message buffering. Some header fields, such as `Content-MD5` (a cryptographic checksum over the message body), cannot be computed until after the message body is generated. In HTTP/1.0, the use of such header fields required the sender to buffer the entire message.

In HTTP/1.1, a chunked message may include a *trailer* after the final chunk. A trailer is simply a set of one or more header fields. By placing them at the end of the message, the sender allows itself to compute them after generating the message body.

The sender alerts the recipient to the presence of message trailers by including a `Trailer` header, which lists the set of headers deferred until the trailer. This alert, for example, allows a browser to avoid displaying a prefix of the response before it has received authentication information carried in a trailer.

HTTP/1.1 imposes certain conditions on the use of trailers, to prevent certain kinds of interoperability failure. For example, if a server sends a lengthy message with a trailer to an HTTP/1.1 proxy that is forwarding the response to an HTTP/1.0 client, the proxy must either buffer the entire message or drop the trailer. Rather than insist that proxies buffer arbitrarily long messages, which would be infeasible, the protocol sets rules that should prevent any critical information in the trailer (such as authentication information) from being lost because of this problem. Specifically, a server cannot send a trailer unless either the information it contains is purely optional, or the client has sent a `TE: trailers` header, indicating that it is willing to receive trailers (and, implicitly, to buffer the entire response if it is forwarding the message to an HTTP/1.0 client).

### 6.3 Transfer-length issues

Several HTTP/1.1 mechanisms, such as Digest Access Authentication (see Section 9.1), require end-to-end agreement on the length of the message body; this is known as the *entity-length*. Hop-by-hop transfer-codings, such as compression or chunking, can temporarily change the *transfer-length* of a message. Before this distinction was clarified, some earlier implementations used the `Content-Length` header indiscriminately.

Therefore, HTTP/1.1 gives a lengthy set of rules for indicating and inferring the entity-length of a message. For example, if a non-identity transfer-coding is used (so the transfer-length and entity-length differ), the sender is not allowed to use the `Content-Length` header at all. When a response contains multiple byte ranges, using `Content-Type: multipart/byteranges`, then this self-delimiting format defines the transfer-length.

## 7 Internet address conservation

Companies and organizations use URLs to advertise themselves and their products and services. When a URL appears in a medium other than the Web itself, people seem to prefer "pure hostname" URLs; i.e., URLs without any path syntax following the hostname. These are often known as "vanity URLs," but in spite of the implied disparagement, it's unlikely that non-purist users will abandon this practice, which has led to the continuing creation of huge numbers of hostnames.

IP addresses are widely perceived as a scarce resource (pending the uncertain transition to IPv6 [DH95]). The Domain Name System (DNS) allows multiple host names to be bound to the same IP address. Unfortunately, because the original designers of HTTP did not anticipate the "success disaster" they were enabling, HTTP/1.0 requests do not pass the hostname part of the request URL. For example, if a user makes a request for the resource at URL `http://example1.org/home.html`, the browser sends a message with the *Request-Line*

```
GET /home.html HTTP/1.0
```

to the server at `example1.org`. This prevents the binding of another HTTP server hostname, such as `exampleB.org` to the same IP address, because the server receiving such a message cannot tell which server the message is meant for. Thus, the proliferation of vanity URLs causes a proliferation of IP address allocations.

The Internet Engineering Steering Group (IESG), which manages the IETF process, insisted that HTTP/1.1 take steps to improve conservation of IP addresses. Since HTTP/1.1 had to interoperate with HTTP/1.0, it could not change the format of the Request-Line to include the server hostname. Instead, HTTP/1.1 requires requests to include a `Host` header, first proposed by John Franks [Fra94], that carries the hostname. This converts the example above to:

```
GET /home.html HTTP/1.1
Host: example1.org
```

If the URL references a port other than the default (TCP port 80), this is also given in the `Host` header.

Clearly, since HTTP/1.0 clients will not send `Host` headers, HTTP/1.1 servers cannot simply reject all messages without them. However, the HTTP/1.1 specification requires that an HTTP/1.1 server must reject any HTTP/1.1 message that does not contain a `Host` header.

The intent of the `Host` header mechanism, and in particular the requirement that enforces its presence in HTTP/1.1 requests, is to speed the transition away from assigning a new IP address for every vanity URL. However, as long as a substantial fraction of the users on the Internet use browsers that do not send `Host`, no Web site operator (such as an electronic commerce business) that depends on these users will give up a vanity-URL IP address. The transition, therefore, may take many years. It may be obviated by an earlier transition to IPv6, or by the use of market mechanisms to discourage the unnecessary consumption of IPv4 addresses.

# 8 Error notification

HTTP/1.0 defined a relatively small set of sixteen status codes, including the normal 200 (OK) code. Experience revealed the need for finer resolution in error reporting.

## 8.1 The Warning header

HTTP status codes indicate the success or failure of a request. For a successful response, the status code cannot provide additional advisory information, in part because the placement of the status code in the *Status-Line*, instead of in a header field, prevents the use of multiple status codes.

HTTP/1.1 introduces a `Warning` header, which may carry any number of subsidiary status indications. The intent is to allow a sender to advise the recipient that something may be unsatisfactory about an ostensibly successful response.

HTTP/1.1 defines an initial set of `Warning` codes, mostly related to the actions of caches along the response path. For example, a `Warning` can mark a response as having been returned by a cache during disconnected operation, when it is not possible to validate the cache entry with the origin server.

The `Warning` codes are divided into two classes, based on the first digit of the 3-digit code. One class of warnings must be deleted after a successful revalidation of a cache entry; the other class must be retained with a revalidated cache entry. Because this distinction is made based on the first digit of the code, rather than through an exhaustive listing of the codes, it is extensible to `Warning` codes defined in the future.

## 8.2 Other new status codes

There are 24 new status codes in HTTP/1.1; we have discussed 100 (Continue), 206 (Partial Content), and 300 (Multiple Choices) elsewhere in this paper. A few of the more notable additions include

- 409 (Conflict), returned when a request would conflict with the current state of the resource. For example, a PUT request might violate a versioning policy.
- 410 (Gone), used when a resource has been removed permanently from a server, and to aid in the deletion of any links to the resource.

Most of the other new status codes are minor extensions.

# 9 Security, integrity, and authentication

In recent years, the IETF has heightened its sensitivity to issues of privacy and security. One special concern has been the elimination of passwords transmitted "in the clear." This increased emphasis has manifested itself in the HTTP/1.1 specification (and other closely related specifications).

## 9.1 Digest access authentication

HTTP/1.0 provides a challenge-response access control mechanism, *Basic authentication*. The origin server responds to a request for which it needs authentication with a `WWW-Authenticate` header that identifies the authentication *scheme* (in this case, "Basic") and *realm*. (The realm value allows a server to partition sets of resources into "protection spaces," each with its own authorization database.)

The client (user agent) typically queries the user for a username and password for the realm, then repeats the original request, this time including an `Authorization` header that contains the username and password. Assuming these credentials are acceptable to it, the origin server responds by sending the expected content. A client may continue to send the same credentials for other resources in the same realm on the same server, thus eliminating the extra overhead of the challenge and response.

A serious flaw in Basic authentication is that the username and password in the credentials are unencrypted and therefore vulnerable to network snooping. The credentials also have no time dependency, so they could be collected at leisure and used long after they were collected. *Digest access authentication* [FHBH+97, FHBH+98] provides a simple mechanism that uses the same framework as Basic authentication while eliminating many of its flaws. (Digest access authentication, being largely separable from the HTTP/1.1 specification, has developed in parallel with it.)

The message flow in Digest access authentication mirrors that of Basic and uses the same headers, but with a scheme of "Digest." The server's challenge in Digest access authentication uses a nonce (one-time) value, among other information. To respond successfully, a client must compute a checksum (MD5, by default) of the username, password, nonce, HTTP method of the request, and the requested URI. Not only is the password no longer unencrypted, but the given response is correct only for a single resource and method. Thus, an attacker that can snoop on the network could only replay the request, the response for which he has already seen. Unlike with Basic authentication, obtaining these credentials does not provide access to other resources.

As with Basic authentication, the client may make further requests to the same realm and include Digest credentials, computed with the appropriate request method and request-URI. However, the origin server's nonce value may be time-dependent. The server can reject the credentials by saying the response used a stale nonce and by providing a new one. The client can then recompute its credentials without needing to ask the user for username and password again.

In addition to the straightforward authentication capability, Digest access authentication offers two other features: support for third-party authentication servers, and a limited message integrity feature (through the `Authentication-Info` header).

## 9.2 Proxy authentication

Some proxy servers provide service only to properly authenticated clients. This prevents, for example, other clients from stealing bandwidth from an unsuspecting proxy.

To support proxy authentication, HTTP/1.1 introduces the `Proxy-Authenticate` and `Proxy-Authorization` headers. They play the same role as the `WWW-Authenticate` and `Authorization` headers in HTTP/1.0, except that the new headers are hop-by-hop, rather than end-to-end. Proxy authentication may use either of the Digest or Basic authentication schemes, but the former is preferred.

A proxy server sends the client a `Proxy-Authenticate` header, containing a *challenge*, in a 407 (Proxy Authentication Required) response. The client then repeats the initial request, but adds a `Proxy-Authorization` header that contains *credentials* appropriate to the challenge. After successful proxy authentication, a client typically sends the same `Proxy-Authorization` header to the proxy with each subsequent request, rather than wait to be challenged again.

## 9.3 Protecting the privacy of URIs

The URI of a resource often represents information that some users may view as private. Users may prefer not to have it widely known that they have visited certain sites.

The `Referer` [*sic*] header in a request provides the server with the URI of the resource from which the request-URI was obtained. This gives the server information about the user's previous page-view. To protect against unexpected privacy violations, the HTTP/1.1 specification takes pains to discourage sending the `Referer` header inappropriately; for example, when a user enters a URL from the keyboard, the application should not send a `Referer` header describing the currently-visible page, nor should a client send the `Referer` header in an insecure request if the referring page had been transferred securely.

The use of a `GET`-based HTML form causes the encoding of form parameters in the request-URI. Most proxy servers log these request-URIs. To protect against revealing sensitive information, such as passwords or credit-card numbers, in a URI, the HTTP/1.1 specification strongly discourages the use of `GET`-based forms for submitting such data. The use of `POST`-based forms prevents the form parameters from appearing in a request-URI, and therefore from being logged inappropriately.

## 9.4 Content-MD5

MIME's `Content-MD5` header contains the MD5 digest of the entity being sent[MR95]. The HTTP/1.1 specification provides specific rules for the use of this header in the Web, which differ slightly from its use in MIME (electronic mail). The sender may choose to send `Content-MD5` so the recipient can detect accidental changes to the entity during its transmission. `Content-MD5` is a good example of a header that a server might usefully send as a trailer.

Clearly the `Content-MD5` value could easily be spoofed and cannot serve as a means of security. Also, because `Content-MD5` covers just the entity in one message, it cannot be used to determine if a full response has been successfully reassembled from a number of partial (range) responses, or whether response headers have been altered.

## 9.5 State management

HTTP requests are stateless. That is, from a server's perspective, each request can ordinarily be treated as independent of any other. For Web applications, however, state can sometimes be useful.

For example, a shopping application would like to keep track of what is in a user's "shopping basket," as the basket's contents change over the course of a sequence of HTTP requests.

Netscape introduced "cookies" [Net] in version 1.1 of their browser as a state management mechanism. The IETF subsequently standardized cookies in RFC2109 [KM97]. (The cookie specification is another example of how HTTP can be extended by a separate specification without affecting the core protocol. Cookie support is optional in servers and user agents, although some Web-based services will not work in their absence.)

The basic cookie mechanism is simple. An origin server sends an arbitrary piece of (state) information to the client in its response. The client is responsible for saving the information and returning it with its next request to the origin server. RFC2109 and Netscape's original specification relax this model so that a cookie can be returned to any of a collection of related servers, rather than just to one. The specifications also restricts for which URIs on a given server the cookie may be returned. A servers may assign a lifetime to a cookie, after which it is no longer used.

Cookies have both privacy and security implications. Because their content is arbitrary, cookies may contain sensitive application-dependent information. For example, they could contain credit card numbers, user names and passwords, or other personal information. Applications that send such information over unencrypted connections leave it vulnerable to snooping, and cookies stored at a client system might reveal sensitive information to another user of (or intruder into) that client.

RFC2109 proved to be controversial, primarily because of restrictions that were introduced to protect privacy. Probably the most controversial of these has to do with "unverifiable transactions" and "third-party cookies." Consider this scenario.

1. The user visits `http://www.example1.com/home.html`.
2. The returned page contains an `IMG` (image) tag with a reference to `http://ad.example.com/adv1.gif`, an advertisement.
3. The user's browser automatically requests the image. The response includes a cookie from `ad.example.com`.
4. The user visits `http://www.exampleB.com/home.html`.
5. The returned page contains an `IMG` tag with a reference to `http://ad.example.com/adv2.gif`.
6. The user's browser automatically requests the image, sending the previously received cookie to `ad.example.com` in the process. The response includes a new cookie from `ad.example.com`.

Privacy advocates, and others, worried that:

- the user receives, in step 3, a ("third-party") cookie from `ad.example.com`, a site she didn't even know she was going to visit (an "unverifiable transaction"); and
- that first cookie gets returned to `ad.example.com` in the second image request (step 6).

If a `Referer` header is sent with each of the image requests to `ad.example.com`, then that site can begin to accumulate a profile of the user's interests from the sites she visited, here `http://www.example1.com/home.html` and `http://www.exampleB.com/home.html`. Such an advertising site could potentially select advertisements that are likely to be interesting to her. While that profiling process is relatively benign in isolation, it could become more personal if the profile can also be tied to a specific real person, not just a persona. For example, this might happen if the user goes through some kind of registration at `www.example1.com`.

RFC2109 sought to limit the possible pernicious effects of cookies by requiring user agents to reject cookies that arrive from the responses to unverifiable transactions. RFC2109 further stated that user agents could be configured to accept such cookies, provided that the default was *not* to accept them. This default setting was a source of concern for advertising networks (companies that run sites like `ad.example.com` in the example) whose business model depended on cookies, and whose business blossomed in the interval between when the specification was essentially complete (July, 1996) and the time it appeared as an RFC (February, 1997). RFC2109 has undergone further refinement [KM98] in response to comments, both political and technical.

## 10 Content Negotiation

Web users speak many languages and use many character sets. Some Web resources are available in several *variants* to satisfy this multiplicity. HTTP/1.0 included the notion of *content negotiation*, a mechanism by which a client can inform the server which language(s) and/or character set(s) are acceptable to the user.

Content negotiation has proved to be a contentious and confusing area. Some aspects that appeared simple at first turned out to be quite difficult to resolve. For example, although current IETF practice is to insist on explicit character set labeling in all relevant contexts, the existing HTTP practice has been to use a default character set in most contexts, but not all implementations chose the same default. The use of unlabeled defaults greatly complicates the problem of internationalizing the Web.

HTTP/1.0 provided a few features to support content negotiation, but RFC1945 [BLFF96] never uses that term and devotes less than a page to the relevant protocol features. The HTTP/1.1 specification specifies these features with far greater care, and introduces a number of new concepts.

The goal of the content negotiation mechanism is to choose the best available representation of a resource. HTTP/1.1 provides two orthogonal forms of content negotiation, differing in where the choice is made:

1. In *server-driven* negotiation, the more mature form, the client sends hints about the user's preferences to the server, using headers such as `Accept-Language`, `Accept-Charset`, etc. The server then chooses the representation that best matches the preferences expressed in these headers.

2. In *agent-driven* negotiation, when the client requests a varying resource, the server replies with a 300 (Multiple Choices) response that contains a list of the available representations and a description of each representation's properties (such as its language and character set). The client (agent) then chooses one representation, either automatically or with user intervention, and resubmits the request, specifying the chosen variant.

Although the HTTP/1.1 specification reserves the `Alternates` [HM98b] header name for use in agent-driven negotiation, the HTTP working group never completed a specification of this header, and server-driven negotiation remains the only usable form.

Some users may speak multiple languages, but with varying degrees of fluency. Similarly, a Web resource might be available in its original language, and in several translations of varying faithfulness. HTTP introduces the use of *quality values* to express the importance or degree of acceptability of various negotiable parameters. A quality value (or *qvalue*) is a fixed-point number between 0.0 and

1.0. For example, a native speaker of English with some fluency in French, and who can impose on a Danish-speaking office-mate, might configure a browser to generate requests including

```
Accept-Language: en, fr;q=0.5, da;q=0.1
```

Because the content-negotiation mechanism allows qvalues and wildcards, and expresses variation across many dimensions (language, character-set, content-type, and content-encoding) the automated choice of the "best available" variant can be complex and might generate unexpected outcomes. These choices can interact with caching in subtle ways; see the discussion in Section 3.4.

Content negotiation promises to be a fertile area for additional protocol evolution. For example, the HTTP working group recognized the utility of automatic negotiation regarding client implementation features, such as screen size, resolution, and color depth. The IETF has created the Content Negotiation working group to carry forward with work in the area.

# 11 Pending proposals

Although the HTTP working group will disband after the publication of the HTTP/1.1 specification, there are numerous pending proposals for further improvements. Here we sketch a few of the more significant ones.

The HTTP/1.1 specification placed a strong emphasis on extensibility but was not able to resolve this issue entirely. Although it has not yet achieved the status of a working group, one effort has been trying to define a general extensibility mechanism [NLL98].

As we noted in Section 10, the Content Negotiation working group is working on proposals to better define content negotiation [HM98b, HM98a] and feature tags.

Several researchers have observed that when Web resources change (thus invalidating cache entries), they usually do not change very much [DFKM97, MDFK97]. This suggests that transmitting only the differences (or *delta*) between the current resource value and a cached response, rather than an entire new response, could save bandwidth and time. Two of us, in conjunction with several other people, have proposed a simple extension to HTTP/1.1 to support *delta encoding* [MKD+99].

In today's Web, content is widely shared, but it mostly flows in one direction, from servers to clients. The Web could become a medium for distributed updates to shared content. The IETF's World Wide Web Distributed Authoring and Versioning (WEBDAV) working group is in the process of defining HTTP extensions to enable this vision [SVWD98].

# 12 Observations

HTTP/1.1 differs from HTTP/1.0 in numerous ways, both large and small. While many of these changes are clearly for the better, the protocol description has tripled in length, and many of the new features were introduced without any real experimental evaluation to back them up. The HTTP/1.1 specification also includes numerous irregularities for compatibility with the installed base of HTTP/1.0 implementations.

This increase in complexity complicates the job of client, server, and especially proxy cache implementors. It has already led to unexpected interactions between features, and will probably lead to others. We do not expect the adoption of HTTP/1.1 to go entirely without glitches. Fortunately, the numerous provisions in HTTP/1.1 for extensibility should simplify the introduction of future modifications.

## Acknowledgments

We would like to thank the anonymous referees for their reviews.

## References

[ASA+95] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. In *Proceedings of the International World Wide Web Conference*, December 1995. `http://ei.cs.vt.edu/~succeed/WWW4/WWW4.html`.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, HTTP Working Group, May 1996.

[CGI98] CGI: Common Gateway Interface, March 1998. `http://www.w3.org/CGI/`.

[Cis] Cisco cache engine. `http://www.cisco.com/warp/public/751/cache`.

[CLF98] HTTP-WG Mailing list archives, Sept-Nov 1994-1998. `http://www.ics.uci.edu/pub/ietf/http/hypermail/`.

[DFKM97] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: A live study of the World Wide Web. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 147–158, December 1997. `http://www.usenix.org/events/usits97`.

[DH95] S. Deering and R. Hinden. Internet Protocol Version 6. RFC 1883, IETF, Dec 1995.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variation via on-demand dynamic transcoding. In *Proc. ASPLOS VII*, pages 160–170, Cambridge, MA, October 1996.

[FGM+98] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet Draft draft-ietf-http-v11-spec-rev-06.txt, HTTP Working Group, November 18 1998. `ftp://ftp.ietf.org/internet-drafts/draft-ietf-http-v11-spec-rev-06.txt`; will be issued as an RFC prior to WWW8.

[FHBH+97] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, and L. Stewart. An Extension to HTTP: Digest Access Authentication. RFC 2069, IETF, January 1997. `ftp://ftp.ietf.org/internet-drafts/draft-ietf-http-authentication-03.txt`; this is a work in progress.

[FHBH+98] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. Internet Draft draft-ietf-http-authentication-03.txt, IETF, September 1998. `ftp://ftp.ietf.org/internet-drafts/draft-ietf-http-authentication-03.txt`; this is a work in progress.

[Fie95] Roy T. Fielding. Keep-alive notes, Oct 10 1995.
http://www.ics.uci.edu/pub/ietf/http/hypermail/1995q4/0063.html.

[FL95] John Franks and Ari Luotonen. Byte ranges – formal spec proposal, May 17 1995.
http://www.ics.uci.edu/pub/ietf/http/hypermail/1995q2/0122.html.

[Fra94] John Franks. Two proposals for HTTP/2.0, November 16 1994.
http://www.ics.uci.edu/pub/ietf/http/hypermail/1994q4/0019.html.

[HM98a] Koen Holtman and Andrew H. Mutz. HTTP Remote Variant Selection Algorithm – RVSA/1.0. RFC 2296, HTTP Working Group, March 1998.

[HM98b] Koen Holtman and Andrew H. Mutz. Transparent Content Negotiation in HTTP. RFC 2295, HTTP Working Group, March 1998.

[IC97] Arun Iyengar and Jim Challenger. Improving Web server performance by caching dynamic data. In *Proc. USENIX Symp. on Internet Technologies and Systems*, December 1997.
http://www.usenix.org/events/usits97.

[Ink] Inktomi traffic server.
http://www.inktomi.com/products/traffic/.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *Proc. 1988 SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314–329, Stanford, CA, August 1988.

[KLM97] Thomas M. Kroeger, Darrel D.E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Symposium on Internet Technology and Systems*. USENIX Association, December 1997.
http://www.usenix.org/publications/library/proceedings/usits97/kroeger.html.

[KM97] D.M. Kristol and L. Montulli. HTTP State Management. RFC 2109, HTTP Working Group, February 1997.

[KM98] D.M. Kristol and L. Montulli. HTTP State Management. Internet Draft draft-ietf-http-state-man-mec-10.txt, HTTP Working Group, July 1998.
http://portal.research.bell-labs.com/~dmk/cookie-3.6.txt; this is a work in progress.

[Mar97] James Marshall. HTTP Made Really Easy.
http://www.jmarshall.com/easy/http/, 1997.

[Mas98] Larry Masinter. Implementation report for HTTP/1.1 to Draft Standard, September 30 1998.
http://www.ietf.org/IESG/http1.1-implementations.txt.

[MDFK97] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proc. SIGCOMM '97 Conference*, pages 181–194, Cannes, France, September 1997. ACM SIGCOMM.

[MFGN97] Jeffrey C. Mogul, Roy T. Fielding, Jim Gettys, and Henrik Frystyk Nielsen. Use and Interpretation of HTTP Version Numbers. RFC 2145, HTTP Working Group, May 1997.

[MKD+99] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, and A. van Hoff. Delta encoding in HTTP. Internet Draft draft-mogul-http-delta-01.txt, IETF, February 1999. `ftp://ftp.ietf.org/internet-drafts/draft-mogul-http-delta-01.txt`; this is a work in progress.

[MR95] J. Myers and M. Rose. The Content-MD5 Header Field. RFC 1864, IETF, October 1995.

[Net] Netscape. Persistent client state HTTP cookies. `http://www.netscape.com/newsref/std/cookie_spec.html`.

[NG98] Henrik Frystyk Nielsen and Jim Gettys. HTTP/1.1 Feature List Report Summary, July 1998. `http://www.w3.org/Protocols/HTTP/Forum/Reports/`.

[NGBS+97] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proc. SIGCOMM '97*, Cannes, France, September 1997.

[NLL98] H. Frystyk Nielsen, P. Leach, and Scott Lawrence. HTTP Extension Framework for Mandatory and Optional Extensions. Internet-Draft draft-frystyk-http-extensions-01.txt, IETF, November 1998. `ftp://ftp.ietf.org/internet-drafts/draft-frystyk-http-extensions-01.txt`; this is a work in progress.

[PM95] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28(1/2):25–35, December 1995.

[SVWD98] J. Slein, F. Vitali, E. Whitehead, and D. Durand. Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web. RFC 2291, WEBDAV Working Group, February 1998.

[TMW97] K. Thompson, G. J. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, November/December 1997.

[Yap97] Ken Yap. A Technical Overview of the New HTTP/1.1 Specification. In *Proc. Australian World Wide Web Technical Conference*, Brisbane, Queensland, Australia, May 1997. `http://www.dstc.edu.au/aw3tc/papers/yap/`.


**Vitae**

Balachander Krishnamurthy is a researcher at AT&T Labs–Research in Florham Park, NJ, and can be reached at bala@research.att.com.

Jeffrey C. Mogul received an S.B. from the Massachusetts Institute of Technology in 1979, an M.S. from Stanford University in 1980, and his PhD from the Stanford University Computer Science Department in 1986. Dr. Mogul has been an active participant in the Internet community, and is the author or co-author of several Internet Standards; most recently, he has contributed extensively to the HTTP/1.1 specification. Since 1986, he has been a researcher at the Compaq (formerly Digital) Western Research Laboratory, working on network and operating systems issues for high-performance computer systems, and on improving performance of the Internet and the World Wide Web. He is a member of ACM, Sigma Xi, and CPSR, and was Programm Committee Chair for the Winter 1994

USENIX Technical Conference, and for the IEEE TCOS Sixth Workshop on Hot Topics in Operating Systems.

Address for correspondence: Compaq Computer Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, California, 94301 (mogul@pa.dec.com)

David M. Kristol is currently a Member of Technical Staff at Bell Laboratories, Lucent Technologies, in the Information Sciences Research Center, where he now does research in security and electronic commerce on the Internet. Previously, he worked on formal specifications for communications protocols. For six years he was responsible for the C compilers for UNIX(r) System V as a member of Bell Labs's Unix support organization, and he was the principal developer of the System V ANSI C compiler. He joined Bell Laboratories in 1981. Earlier, Kristol worked at Massachusetts Computer Associates, where he developed a mass spectrometry data system, and at GenRad, Inc., where he developed automatic test equipment systems. He received BA and BSEE degrees from the University of Pennsylvania, Philadelphia, and MS and ME degrees (in Applied Mathematics) from Harvard University.

His email address is: dmk@bell-labs.com.