Clarifying the Fundamentals of HTTP

Jeffrey C. Mogul
Compaq Computer Corporation Western Research Laboratory
250 University Avenue, Palo Alto, CA 94301
Jeffrey.Mogul@Compaq.com
JeffMogul@ACM.org
+1 (650) 617-3304

Abstract

The simplicity of HTTP was a major factor in the success of the Web. However, as both the protocol and its uses have evolved, HTTP has grown complex. This complexity results in numerous problems, including confused implementors, interoperability failures, difficulty in extending the protocol, and a long specification without much documented rationale.

Many of the problems with HTTP can be traced to unfortunate choices about fundamental definitions and models. This paper analyzes the current (HTTP/1.1) protocol design, showing how it fails in certain cases, and how to improve these fundamentals. Some problems with HTTP can be fixed simply by adopting new models and terminology, allowing us to think more clearly about implementations and extensions. Other problems require explicit (but compatible) protocol changes.

Categories: C.2.2 [**Computer-communication Networks**]: Network Protocols---*Applications*
Terms: Design
Keywords: HTTP, protocol design

This is a preprint of a paper to appear in *Proceedings of the Eleventh International World Wide Web Conference*, May, 2002, Honolulu, Hawaii. The final paper will be formatted differently.

Clarifying the Fundamentals of HTTP

Abstract

The simplicity of HTTP was a major factor in the success of the Web. However, as both the protocol and its uses have evolved, HTTP has grown complex. This complexity results in numerous problems, including confused implementors, interoperability failures, difficulty in extending the protocol, and a long specification without much documented rationale.

Many of the problems with HTTP can be traced to unfortunate choices about fundamental definitions and models. This paper analyzes the current (HTTP/1.1) protocol design, showing how it fails in certain cases, and how to improve these fundamentals. Some problems with HTTP can be fixed simply by adopting new models and terminology, allowing us to think more clearly about implementations and extensions. Other problems require explicit (but compatible) protocol changes.

1. Introduction

HTTP appears to be a very simple protocol, and its simplicity was a major factor in the success of the Web. We have learned, however, that HTTP is more complex than it first looked. Partly, this is because the actual complexity increased as the protocol evolved; partly, because the environment in which HTTP is applied has become more complex (especially with the introduction of intermediaries such as caches); and partly, because the protocol always was more complicated than it seemed.

The complexity of HTTP causes many problems, not the least of which is a lengthy (176-page) specification without much explicit rationale for the design decisions. As a result, many implementors have been unable to understand how to combine features in ways not specifically addressed in the document. The natural consequences of this confusion are interoperability failures, limited support for useful but subtle features, and repetitive discussions on mailing lists.

Albert Einstein is supposed to have said ‘‘Everything should be as simple as possible, but not simpler’’ [8]. Protocol designers should keep this in mind; decisions to leave things out of a protocol, in search of apparent simplicity, can create actual complexity as people try to bend the protocol to solve hard problems. Much of the complexity of HTTP/1.1 stems not from the length of its specification, but rather what was left unsaid.

We can trace many of the problems with HTTP to what, in hindsight, were incompletely considered choices about fundamental definitions and models. The deepest problems lie with the protocol’s lack of a clean underlying data model, but HTTP/1.1 also failed to resolve significant problems with extensibility.

In particular, HTTP forces implementations to infer certain pieces of information that should and could be explicit in the protocol messages. Inference is seldom a good mechanism to ensure reliable interoperation, and prevents certain useful extensions because implementations depend on behavior not actually defined in the specification.

In this paper, I analyze certain fundamentals of the current (HTTP/1.1) protocol design, to show where and how it fails. I also suggest some ways to fix both the protocol and the underlying concepts. While some of the problems with HTTP do require explicit (but compatible) protocol changes, others can be fixed simply by adopting new models and terminology. This should allow us to think more clearly about how to implement or extend the protocol.

1.1. Why Write This Paper, and Why Now?

For about six years, I have been involved in the process of designing the HTTP/1.1 revision of the protocol [10], particularly the aspects related to caching, as well as several subsequent extensions intended to improve the utility of HTTP caches. These design efforts have often been difficult because HTTP lacks a clear and consistent data type model for the primitive structures of the protocol itself.

It is quite unlikely that these conceptual problems could have been addressed, or even foreseen, when HTTP was first designed. The protocol and its uses have co-evolved, in ways which the original designers could not have predicted. But with several years of deployment experience with the Web, we can now see many problems quite clearly.

However, we have learned from the slow transition between HTTP/1.0 and HTTP/1.1 [18] just how difficult it could be to replace HTTP with a new protocol. HTTP is the protocol that we have, and it would be pointless to propose tossing it out and starting over.

The goals of this paper, then, are:

- **Start with the existing HTTP protocol**: The installed base of HTTP systems must constrain any proposals.
- **Identify the problems**: From our experience with HTTP, we can recognize sets of related problems with the current models.
- **Get the concepts crisp and right**: Provide a common terminology that both reflects reality and allows for unambiguous interpretations.
- **Create guidance for**: (1) Implementation decisions where the existing specification is ambiguous; (2) future extensions to HTTP; and (3) designers of future new protocols, who can learn from the HTTP experience.
- **Judiciously suggest new *tagging* mechanisms to add to HTTP**: Almost all of the problems identified in this paper can be addressed by providing explicit information in the protocol, instead of forcing implementations to guess.

It may seem paradoxical to try to change the definitions and even the specification of HTTP without making changes that are incompatible with the installed base. However, in many cases the ‘‘missing’’ aspects of the specification are inescapably implied by the logical consequences of what is already there. Our task is thus to unearth these consequences, rather than to invent new specification details from scratch.

1.2. The Particular Importance of Caching

HTTP is a network protocol, but it is also the basis of a large and complex distributed system. HTTP has clients, servers, intermediate relays (‘‘proxies’’), and the possibility of caches at all points. It can be difficult to design the caching mechanisms in a distributed system, especially if one hopes to make the caches ‘‘semantically transparent’’ (that is, caching should be invisible to the end systems except for its effect on performance).

HTTP acquired its caching mechanisms by accretion. The first mechanisms quickly supported significant benefits from simple cache implementations, but also opened the door for an array of confusing feature interactions. During the HTTP/1.1 design process, this lead to a debate between people who viewed aggressive caching as vitally important to the health of the Internet, and people who viewed it as potentially dangerous to the semantic integrity of the Web. In fact, it should be possible to design a caching system that guarantees semantic transparency to Web interactions while still eliminating nearly all truly excess costs. But, this requires a more rigorously defined caching design than has evolved for HTTP. Our failure to get that design right is largely a consequence of a conceptual faultline between ‘‘protocol designers’’ and ‘‘distributed system designers,’’ and a failure to meld the expertise of both camps.

Most of the topics discussed in this paper ultimately reflect a need for clear definitions and explicit information, in order to support safe, aggressive caching. Ambiguity is the enemy of caching, because it forces the use of inferences, which reduces the opportunities for truly safe caching, and makes it nearly impossible to compose independently developed caching designs.

2. HTTP's existing data type model

Analysis of a data-oriented protocol such as HTTP should start with an understanding of the protocol's data type model. By this, I mean the various data types on which the protocol operates. This paper is concerned with the data types acted on and interpreted by the protocol, and *not* with the higher-level types that are transported via HTTP, but which are opaque to the protocol itself.

2.1. Resources

HTTP requests always specify a URI (Uniform Resource Identifier). Thus, every HTTP request represents an attempted operation on at least one ‘‘resource,’’ and every HTTP response message conveys something about the result of that attempt. The HTTP specification's definition of the term ‘‘resource’’ is circular (‘‘A network data object or service that can be identified by a URI’’, but ‘‘[URIs] are simply formatted strings which identify ... a resource’’), a problem for someone else to unravel.

To support features such as multilingual documents, an HTTP resource may have multiple ‘‘variants.’’ Each variant of a given resource is expected to represent the same conceptual thing, but (e.g.) the French and Chinese versions of a document might have nothing visibly in common. The Request-URI is not sufficient to identify a unique variant; other request header fields called ‘‘selecting headers’’ (e.g., Accept-Language) might be involved. (It is convenient to treat an unvarying resource as having exactly one ‘‘variant.’’) Variants introduce their own set of problems, later briefly covered in section 7.1.

2.2. Messages

An HTTP message consists of protocol-visible header information, followed by an optional body (an opaque sequence of bytes). The body of an HTTP message has one of these higher-level types, described by an orthogonal ‘‘content-type’’ system that HTTP inherited from MIME [12]. The HTTP message headers can (should) convey application-level content-type tags for this body, such as ‘‘image/jpeg’’ or ‘‘text/html’’, but HTTP *per se* does not concern itself with the interpretation of content-types.

2.3. Entities and Entity Tags

Relatively early in its history, the HTTP protocol adopted a number of concepts from MIME (Multipurpose Internet Mail Extensions) [12]. In particular, MIME uses the term ‘‘entity’’ to refer to

> [the] MIME-defined header fields and contents of either a message or one of the parts in the body of a multipart entity. The specification of such entities is the essence of MIME.

HTTP adopted the term and defines it similarly, as

> The information transferred as the payload of a request or response. An entity consists of metainformation in the form of entity-header fields and content in the form of an entity-body ...

For comparison, a dictionary definition for ‘‘entity’’ is ‘‘something that has separate and distinct existence and objective or conceptual reality’’ [21].

HTTP/1.1 introduced ‘‘entity tags,’’ used to validate cache entries. The server generates entity tags, which are strings whose content is opaque to the client. The server may send an entity tag in a response. Later, a client wishing to validate a cache entry for this response (that is, to check whether the cache entry is coherent with what the server would send in response to a new request) simply returns this entity tag string to the server. If it matches the current entity-tag, then the server can respond with a ‘‘Not Modified’’ message instead of sending the entire entity.

3. Problems with the current model

Consider the result of the simplest HTTP operation, a GET on a URI with exactly one ‘‘variant.’’ What is the data type of the result? Is it an entity?

The problem with the attempted analogy between MIME messages and HTTP data types is that it assumes the message is the central concern.

In MIME, an email protocol, the message is indeed central. Every MIME entity (email message) is fully contained within a single SMTP-layer message. Further, in an email system, the entity (email message) truly is ‘‘something that has separate and distinct existence’’: the email message actually leaves the sender and travels to the receiver.

In HTTP, however, resources are central, and messages are not nearly as central as they are in MIME. For one thing, the resource (upon which the HTTP request operates) does not itself travel from server to client. This is especially true of dynamic resources, such as CGI scripts or database query systems. Also, HTTP allows the transmission of subrange of the bytes of a result, or of just the metainformation without the associated body, so the result might span several HTTP-layer messages. Therefore, what HTTP calls an entity cannot be said to have a separate and distinct existence; it is merely an ephemeral, and perhaps partial, representation of one aspect of a resource.

HTTP therefore has reasonably well-defined terms and concepts for resources and messages, but no clearly defined term to describe the result of applying an operation to a resource. In other words, what do we call ‘‘the result of successfully applying a simple HTTP GET request to a given resource variant at a given point in time’’?

This could appear to be merely a quibble about terminology. In fact, however, the lack of such a term, and the failure to recognize the importance of the concept, has led to a number of difficult problems. I will discuss three in detail: how to specify HTTP caching, how to consistently deal with partial results, and how to categorize HTTP header fields.

3.1. How to Specify HTTP Caching

What does an HTTP cache store? This might seem to be a trivial question, but a clear answer (or the lack of clear answer) has a profound effect on how to specify and implement HTTP caches.

There are some things that clearly aren’t stored by an HTTP cache. A cache does not store the actual requested ‘‘resource’’: resources themselves do not transit the network (think of a stock-quote resource, for example).

Neither does a cache store ‘‘objects’’ in the sense of object-oriented programming; while HTTP supports multiple methods applied to many resources, HTTP caches can currently only respond to GET methods.

And clearly a cache cannot, in general, store a Web ‘‘page’’ or ‘‘document,’’ since these are often composites of multiple resources with differing cachability properties.

During the design of HTTP/1.1, we debated whether HTTP caches stored ‘‘responses’’ or ‘‘values.’’ Most other kinds of computer caches store values; for example, a CPU cache entry might store the value of a memory line, and a file cache page might store the value of a disk block. But an HTTP resource might respond differently to different requests, so it was hard to define what the ‘‘value’’ of a resource is.

Instead, HTTP caches are currently defined as storing ‘‘response messages.’’ In other words, an HTTP cache entry does not store what a resource is; it stores what the resource says. As a result, it is difficult to define precisely what an HTTP cache must do in many circumstances, since the same resource could say two different things in response to two apparently identical requests. HTTP/1.1 includes a mechanism (the Vary header) which allows a server to tell a cache the ‘‘selecting headers’’ that the result of a request depends on, in addition to the Request-URI. However, the Vary mechanism cannot deal with even fairly simple generalizations, often requiring a fallback to non-caching operation.

The lack of a clear formal specification for caching causes implementors to make guesses, based on fuzzy ideas of what is ‘‘good’’ (e.g., minimizing network traffic). This leads to non-interoperability, because content providers cannot predict what caches do.

As an aside, we also lack consistent, rigorous shared definitions for terms such as ‘‘cache hit’’ and ‘‘cache miss’’ as applied to HTTP. In traditional (CPU or file system) caches, a reference is simply either a hit or a miss, but because HTTP allows conditional requests (e.g., using the If-Modified-Since header), we have the potential for an in-between case: the cache cannot satisfy a request without contacting the origin server, but we might still be able to avoid transferring a response body. And because HTTP caches do not guarantee coherency, a ‘‘hit’’ might or might not yield the right answer. Nobody as yet has proposed a standard taxonomy of HTTP cache hits and misses, although many papers have described private, partial taxonomies (Dilley gives one of the best [5]).

3.2. Consistent Handling of Partial Results

HTTP originally assumed that a result would be carried in a single response message. HTTP/1.1, however, introduced the possibility for a client to request a partial result (or ‘‘range’’), thus allowing a full result to be transmitted using a series of messages. It also recognized the possibility that a client might never want more than a subset of the entire result (e.g., a chapter from a PDF) file, and so might plausibly cache a partial result.

The entity-based data model, unfortunately, does not entirely support partial results.

For example, consider a cache that already stores the first half of a result. When the second half of the result arrives, we would like the cache to unite the two halves into a single cache entry. But this entry could no longer be treated as containing any specific ‘‘response’’ sent by the server, so the caches-store-responses view becomes untenable. In fact, the HTTP/1.1 specification had to include special rules for composing multiple responses into one.

The creation of a cache entry by combining multiple responses introduces the possibility of erroneous assembly, so we would like a means to check end-to-end integrity for the entire result. HTTP defines a Content-MD5 header, inherited from MIME, which carries a digest of the entity (message) body. But this is useless for checking the integrity of a result assembled from partial responses, because each digest only covers a single part.

The situation becomes more complex when combining ranges with compression. HTTP/1.1 allows compression either as an end-to-end ‘‘content-coding,’’ or as a hop-by-hop ‘‘transfer-coding.’’ Generally, the end-to-end approach is more efficient. The specification defines a content-coding as a transformation on an entity; that is, both the input and output is of type ‘‘entity.’’

Suppose a client requests both a compression content-coding and a byte range (e.g., bytes 1-1000). In what order should the server perform the byte-range selection and the compression (which presumably changes the byte numbering)? The specification does not explicitly resolve this ambiguity. However, there is an implicit reason why the compression cannot be done after the range selection: there would then be no way to consistently choose the point where the entity tag (see section 2.3) is assigned.

We can deduce this point from two constraints:

1. An entity tag must be assigned before the range selection. Otherwise, a client trying to assemble a full result from two or more ranges (in multiple messages) could not match the entity tags to test cache coherency.
2. The specification must allow an entity tag to be assigned after the application of a content-coding, because it already allows the server to store its data in a pre-encoded form (and thus to require the entity tag to be assigned prior to any content coding would make all existing servers non-compliant).

Since the entity tag must be assigned prior to range selection but after the application of content-coding, range selection cannot precede content-coding. Otherwise, we would have to accept inconsistent rules about when to assign the entity tag. (This chain of logic serves as an example of a point made in section 1.1: some of the incompletely specified aspects of HTTP may be deduced as necessary consequences of the existing specification.)

Now consider the extension of HTTP to support ‘‘delta encoding’’, in which the server transmits the differences between the client’s cache entry and what the server would currently return for a full response [15]. When we first tried to define this extension [25] we thought delta encoding should be treated as just another form of content-coding, since it resembles compression. But it more closely resembles range selection, in that it transmits partial content that must be combined with an existing cache entry. Trying to treat delta encoding as a content-coding turned out to create extremely complex rules for handling entity tags and cache entries. We also found it difficult to define how a client could ask for delta encoding, ranges, and compression in various orders.

Other proposed HTTP extensions, such as rsync [31] and cache-based compaction [4], share the definitional problems first seen with delta encoding.

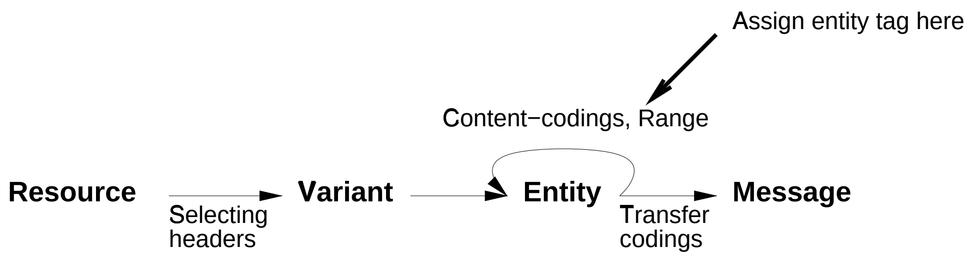

Figure 3-1: Original HTTP message generation pipeline model

We can visualize the situation by depicting the HTTP message generation model as a pipeline, as shown in figure 3-1. In this figure, datatypes (bold-faced terms) are transformed via processing steps (arrows) such as variant selection, application of content-codings, range selection, etc. The figure shows that the server must assign an entity tag at a point not associated with a specific stage in the pipeline. The model includes a cycle, where an entity can be both input and output for either range selection or the application of content-codings. This cycle is what leads to the apparent need for complex rules to define how delta encoding works.

The situation with content-codings is further complicated by the pragmatic distinction between on-the-fly encoding, in which the server applies a content-coding at the moment it transmits a normally unencoded resource, and as-stored encoding, in which the native (on-disk) form of a resource is already compressed. HTTP treats both of these cases identically, yet a naively-implemented server might apply range selection before on-the-fly encoding but not before as-stored encoding.

3.3. Categorization of Headers

The HTTP/1.1 specification distinguishes between ‘‘general headers ... which do not apply to the entity being transferred’’ (e.g., Cache-Control, Date), ‘‘response headers ... additional information about the response’’ (e.g., ETag, Age), and ‘‘entity headers ... metainformation about the entity-body or, if no body is present, about the resource identified by the request’’ (e.g., Expires). Note that ‘‘Cache-control: max-age’’ and Expires are categorized differently, even though they have essentially the same function. There is also a confusion here between bodies and resources -- bodies expire, but resources don’t.

The specification also lumps all extension headers (that is, headers that might be defined in the future) as entity headers, even though plausible extensions could introduce new headers that, as with existing general headers, ‘‘do not apply to the entity being transferred.’’

These flaws in the specification not only can confuse implementors; they also limit extensibility, since a proxy might not properly handle an extension header that it cannot correctly characterize. Extension designers must consider how naive or pre-existing proxies and caches might confound a proposed extension design.

3.4. Summary of the Problems

The current HTTP model leaves caching hard to specify, makes it difficult to cleanly classify HTTP headers, leads to confusion about when to assign entity tags, and makes it very difficult to consistently deal with both partial results and compression.

The complexities and ambiguities of the model create traps for unwary implementors and for designers of new protocol extensions.

4. A better data type model

All of these problems can be solved by adding one new data type to the HTTP model, which I have called the ‘‘instance’’:

> The entity that would be returned in a full response to a GET request, at the current time, for the selected variant of the specified resource, with the application of zero or more content-codings, but without the application of any instance manipulations.

In this revised model, the input to a content-coding transformation is the selected variant of the requested resource, and the output is an instance. (It is convenient to treat the no-content-coding case as the application of an identity content-coding.)

The instance is then used as the input to a series of zero or more ‘‘instance manipulations,’’ which can include range selection, delta encoding, and compression. The result of the series of instance manipulations (possibly just the identity function) is an HTTP entity. The specification for delta encoding [24] now extends HTTP both to allow the server to list the instance manipulations applied in a response, using a new header named IM, and to allow the client to list a set of acceptable instance manipulations, using a new header named A-IM. The A-IM header also allows the client to specify the ordering if the server applies multiple instance manipulations.

In the new model, it becomes clear that the entity tag is assigned to the instance, because it must be assigned prior to any instance manipulations. It is clearly not assigned to the entity (and would better have been called an ‘‘instance tag’’).

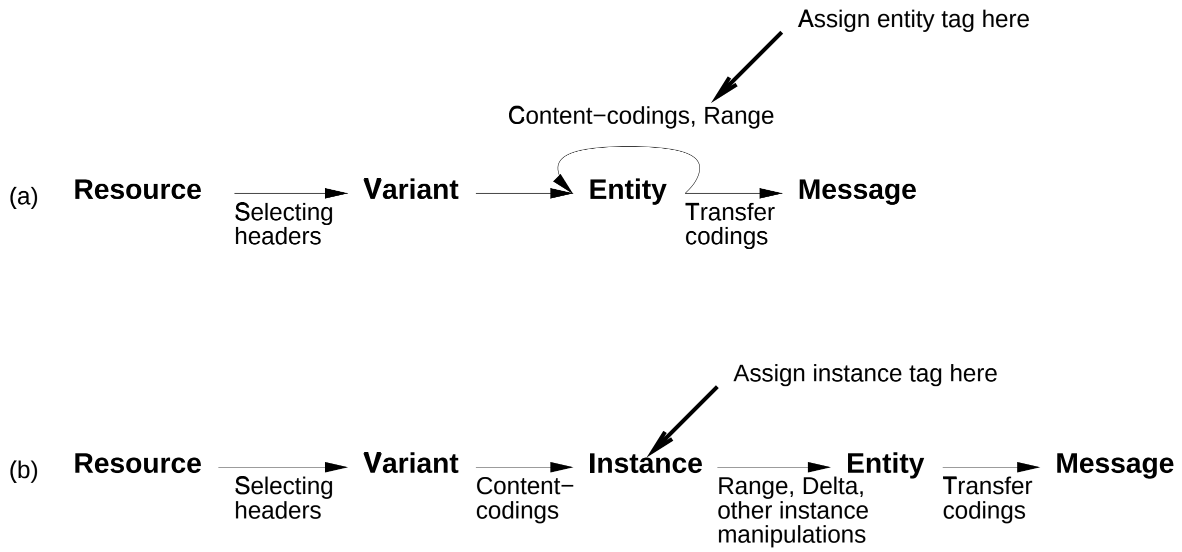

Figure 4-1: HTTP message generation pipelines: (a) old model, (b) new model

Figure 4-1 depicts the differences between the old and new models. In the new model (figure 4-1(b)), there are no cycles, and there is a clear relationship between processing stages (e.g., applying a Content-coding) and links in the graph. Also, while the old model requires the entity tag to be assigned between two different steps that both result in entities, the new model clearly associates the assignment of an entity tag with a specific stage in the pipeline.

It might seem redundant to allow compression either as a content-coding or as an instance manipulation. In retrospect, this simplifies the distinction between on-the-fly compression (best described as an instance manipulation) and the server's use of pre-compressed files (in which compression naturally would be done prior to the assignment of an entity tag).

Table 4-1 shows how various operations that may be included in the message generation pipeline now have well-defined and unique input and output data types. The ''HTTP protocol elements'' column shows just a subset of the relevant protocol elements.

Operation	**HTTP protocol elements**	**Input type**	**Output type**
Resource selection	URL	(none)	Resource
Variant selection	''Selecting headers''	Resource	Variant
Apply content-coding (e.g., compression)	Content-encoding	Variant	**Instance**
Apply instance manipulation (e.g., Range selection, Delta encoding, compression, etc.)	IM, Content-Range	**Instance**	Entity
Apply hop-by-hop transfer-coding (e.g., compression, chunking, trailers)	Transfer-Encoding	Entity	Message

Table 4-1: Input and output data types for message generation operations

It is now also clear what an HTTP cache does: it stores instances (rather than entities or responses). This is the only point in the message generation pipeline where one can implement cache coherency through the use of entity tags. Of course, in the many cases where no instance manipulation or transfer-coding has been used, there is no practical distinction between ''instance,'' ''entity,'' and ''response,'' but more generally the three data types are distinct.

A cache entry might, in some cases, contain only part of an instance: for example, after an aborted transfer or after Range request. However, because a cache entry is always associated with a specific instance, it is clear how the cache should combine multiple partial responses for that instance.

4.1. Better Header Classifications

Section 3.3 discussed the distinction that the HTTP/1.1 specification makes between general, response, and entity headers, arguing that the choices are often confusing.

The new model makes it easier to categorize HTTP headers. We can create the new category of ''instance headers,'' for instance-specific meta-information. Many headers currently classified as entity headers (such as Content-Language, Content-Type, and Last-Modified) or response headers (Etag and perhaps others) are associated with a specific instance, and should be called instance headers.

We can also use the new categories ''resource header'' and ''variant header'' for fields that pertain to those datatypes, and ''outer header'' for fields pertaining only to the message (HTTP already defines the term ‘‘message-header’’). The category ‘‘connection header’’ pertains to aspects of the transport connection (which is orthogonal to all other categories) and ‘‘server header’’ applies to aspects of the server *per se*.

Field name	**RFC2616 category**	**New category**
Accept-Ranges	response-header	resource-header
Age	response-header	outer-header
Allow	entity-header	resource-header
Cache-Control	general-header	instance-header
Connection	general-header	conn-header
Content-Encoding	entity-header	instance-header
Content-Language	entity-header	variant-header?
Content-Length	entity-header	**entity-header**
Content-Location	entity-header	variant-header?
Content-MD5	entity-header	**entity-header**
Content-Range	entity-header	**entity-header**
Content-Type	entity-header	variant-header
Date	general-header	instance-header
ETag	response-header	instance-header
Expires	entity-header	instance-header
Last-Modified	entity-header	instance-header
Location	response-header	resource-header
Proxy-Authenticate	response-header	resource-header
Retry-After	response-header	server-header?
Server	response-header	server-header
Trailer	general-header	outer-header
Transfer-Encoding	general-header	outer-header
Upgrade	general-header	conn-header
Vary	response-header	variant-header
Via	general-header	outer-header
WWW-Authenticate	response-header	resource-header
Warning	general-header	outer-header?

Table 4-2: Header field classifications

Table 4-2 classifies many of the HTTP/1.1 header fields according to both the existing (RFC2616) categorization and this recategorization. (The table excludes fields used only in request messages, although perhaps those fields could be similarly categorized.) Some entries in the ‘‘New’’ column are shown with question marks, since the existing specification is not always precise about the necessary distinctions. (Note that the Date header is listed as an instance-header, but is also required in HTTP responses that do not pertain to specific instances.)

In hindsight, it might have been wise to explicitly tag each HTTP header with its category, thus allowing implementations (especially proxies) to properly handle all headers without knowing their specifications. For example, all instance header names would start with ‘‘I-’’ (I-Content-Language, I-Tag, etc.) Such a scheme could still be adopted for all future definitions of HTTP headers.

This recategorization does not necessarily change the actual protocol. However, it can help to simplify some of the arcane specification rules.

One such set of rules cover ‘‘hop-by-hop,’’ ‘‘non-modifiable,’’ and ‘‘end-to-end’’ headers. The listings of these headers in RFC2616 appear arbitrary, but (if one ignores the possibility of transcoding by proxies), the set of hop-by-hop headers roughly matches the sets of outer-headers and connection-headers in table 4-2, and the set of non-modifiable headers roughly matches the set of instance-headers and variant-headers. Arguably, the mismatches reflect errors in the lists given by RFC2616. Unfortunately, a few loose ends persist: the resource-header category does not map cleanly to either hop-by-hop or non-modifiable; the Content-Length header requires complex special treatment in HTTP/1.1 (because of compatibility issues with older versions); and the question of which headers can be modified by a transcoding proxy creates significant complexity.

4.2. Can We Simplify the Specification?

In section 1, I alluded to the length and the complexity of the HTTP specification. One of the anonymous reviewers asked whether the new five-stage data model would allow the specification to be broken up into independent layers; perhaps the requirements for a cache could be defined entirely in terms of the instance data type, since caches store instances.

This does not seem possible for the existing HTTP protocol. For example, caching may involve not only instance headers (such as Expires) but also an outer header (Age, which is modified on every hop) and perhaps certain variant headers (if the cache participates in content negotiation) and entity headers (such as Content-Range, if the cache uses Range requests). This resistance to strict layering might be the result of HTTP’s history, reflecting design both by evolution and before the construction of this model, but it might reflect intrinsic complexity (‘‘Everything should be as simple as possible, but not simpler’’). It might also imply that the proposed five-stage model could be improved upon.

However, one could, as an exercise, construct an ‘‘HTTP-prime’’ protocol designed to strictly fit a layered model. Such a protocol would have a connection layer, a hop-by-hop message layer (including transfer-codings), a caching layer (including instance manipulations and naming mechanisms for resources), a resource layer (dealing with operations on resources), and a content-typing layer (for use by higher levels). Support for content negotiation and variants, at least as they are currently treated in HTTP, would probably still create cross-layer issues (because variants affect both naming and content-typing).

5. Data access model

The Web, almost from its beginning, has supported not just access to static ‘‘documents,’’ but also interactive information retrieval (e.g., search engines, street maps), information modification (e.g., discussion forums, Web logs), and active operations with real-world side effects (e.g., ordering pet food, or buying stock in pets.com). However, we seem unable to shake the habit of using the term ‘‘document’’ to apply generically to the object of Web requests. Many research papers use ‘‘document’’ (or ‘‘page’’), when a more accurate term would be ‘‘resource,’’ ‘‘response,’’ or (as in section 4) ‘‘instance.’’ Even the HTTP/1.1 specification often uses the term ‘‘document’’ in place of more precise terms, and without any formal definition.

HTTP needs a *data access model* describing both *what* kinds of data items can be accessed (e.g., static documents, product order forms, etc.) and *how* data is accessed (e.g., reading, updating, or more complex modifications). Although earlier versions of the specification [9] called HTTP ‘‘object-oriented,’’ it is not, but no complete alternative model has yet been defined. Most people have only a hazy idea, based on the uses they have already seen for HTTP.

This ambiguity about the data access model is reflected in the protocol itself. The distinction between static documents and other kinds of resources is at best implicit, and often impossible even to infer from the protocol messages. For example, one can infer that if a POST method is not rejected by the server, then the URI involved is not simply an immutable static document. But it is impossible to make that kind of inference by observing a server’s response to a GET request.

When we think solely of static documents, we ignore several aspects of other kinds of Web resources. For example:

- **What update operations are allowed?** Some resources accept update operations, such as PUT or POST, but most only accept GET.
- **Is the resource mutable?** Regardless of whether the resource allows a client to update it, it might still be subject to change. Many, if not most, Web pages do change over time [6]. However, some documents are immutable (such as IETF RFCs, whose text is immutable by definition, although some sites provide additional formatting that may be mutable). Mutability has implications for cache consistency and for automated clients, such as search-engine crawlers.
- **Do operations on the resource have side effects?** Reading a static document may have no hidden consequences, but many other HTTP operations do have side effects. Some are insignificant (such as updating a page’s ‘‘hit counter’’). Significant side effects can include effects on the physical or financial world (e.g., operating a remote control, or buying a stock); on other HTTP-accessible resources (e.g., posting a message to a discussion forum); on other online resources (e.g., submitting a paper to a conference), or even on the client browser (e.g., updating the bookmarks or infecting the client with a virus).
- **Are operations on the resource idempotent?** Users often retry a specific HTTP operation (i.e., send exactly the same request message more than once). This is often done to work around an error, such as a dropped TCP connection, anticipating that repeating the request will yield the *same* result. It is also often done to update the user’s view of a resource (such as a sports score), in the hope that repeating the request will yield a *new* result. Thus, the ‘‘Reload’’ button may have a distinctly different meaning depending on whether a request is idempotent or not. Although browsers often warn the user about potentially dangerous repetitions (‘‘repost form data?’’), this does not address the root problem: does the resource support the operation that the user intends?
- **Does the URI name a specific or generic target?** A Web site might employ a naming structure in which the binding between a URI and the underlying resource is specific; for example, the URI might denote the October 3, 2001 front page of the *Podunk Times*. Or the site might employ a generic binding: the newspaper’s URI might denote ‘‘today’s’’ front page (and when you read this, ‘‘today’’ will no longer be October 3, 2001). Both naming structures can be useful, but the choice can affect, for example, the approach taken to automatically archiving the contents of remote sites. There is no reliable way to discover which structure a site or resource uses.

While most of these issues may seem irrelevant to human users, they are more significant to automated clients, especially to intermediaries such as proxies and caches. Something that might be ‘‘obvious from context’’ to an intelligent human is usually not obvious to software, unless it is made explicit.

5.1. Access-model Support in HTTP/1.1

HTTP includes several mechanisms to ameliorate the lack of an explicit data access model. By convention, for example, the GET method ‘‘SHOULD NOT have the significance of taking an action other than retrieval,’’ but the specification goes on to say that ‘‘some dynamic resources consider [generating side-effects on a GET method] a feature’’ [10] (section 9.1.1). It is not clear if one can rely, in practice, on this aspect of GETs. The specification also requires certain methods to be idempotent; this requirement might be widely abused.

HTTP includes an ‘‘Allow’’ response header which lists the allowable methods (such as PUT). Typical implementations, such as Apache, send this only in response to an unallowed method (a reasonable optimization, since relatively few resources accept methods other than GET.)

HTTP also supports the use of the ‘‘Location’’ header, returned in a response to a POST method that creates a new resource (e.g., adding a message to a discussion forum). This helps to expose the relationship between resources. It is not clear what a server should send if a POST results in the creation of more than one resource, since the specification of Location allows only one such URI per response. Moreover, one cannot know prior to performing the POST request whether the method will create a new resource.

Generally, however, the mechanisms currently supported by HTTP tend to act as directives rather than as labels. For example, HTTP allows a server to defeat caching for responses if the resource’s semantics do not permit caching, but provides no way to label a resource as immutable. The use of directives rather than labels makes it harder to introduce new, unanticipated services at intermediaries and clients.

We also need to make a clean separation between access-model issues that relate to caching, and those that do not. Existing rules (both in the specification and in folklore) that restrict response caching based on inferences about the access model (e.g., responses to URLs containing ‘‘?’’ are not cachable) are both too weak (to prevent some cache coherency failures), and too strict (they can forbid caching when it is safe). If we had an explicit and useful data access model, we would not need, for example, to confuse caching-related labels with URL query syntax.

5.2. More Explicit Labeling

At least one proposal has been made to add access-model labels to HTTP. RFC2310 [14] defines a ‘‘Safe’’ response header. This would indicate whether the request (such as a POST) could be ‘‘repeated automatically without asking for user confirmation.’’ Note that this label applies to a specific request, not to the resource, since it is possible that some POST requests to a given resource are repeatable, while others are not. It might be feasible to treat the Safe header as a label on a given instance of the resource, although the consequences of this choice are not entirely clear.

However, a Safe header could not always be treated as applying to all other instances of a resource, since the semantics of the application behind the resource might not allow this. This suggests that a labeling scheme needs to distinguish between resource labels and instance labels.

HTTP needs a more general scheme for attaching a variety of access-model labels to instances or resources. For example, a resource might be labeled ‘‘immutable’’ (instances, as a consequence of the term’s definition, are always immutable). Resources or instances could be labeled as idempotent. And by analogy with the programming-language concepts of ‘‘lvalues’’ and ‘‘rvalues’’, a resource could be labeled as ‘‘assignable,’’ meaning that it can accept PUT methods (already supported in HTTP using the ‘‘Allow: PUT’’ header).

To conserve bandwidth, such a labeling scheme should use a compact representation. While HTTP traditionally has used human-readable encodings for protocol elements, this is neither necessary nor appropriate for labels meant primarily for automated interpretation. For example, a hypothetical response header such as:

```
Labels: R:IAD, I:S
```

would indicate that the (R)esource is (I)mmutable, (A)ssignable, and i(D)empotent, while the (I)instance is (S)afe. (The actual encoding format is best left to a standards group.)

5.3. Static vs. Dynamic Resources

We have historically made a distinction between ‘‘static’’ resources, for which a Web server simply returns the contents of an existing file, and ‘‘dynamic’’ resources, generated by a process at the time the request is received. Certainly for server implementors, this is an important distinction. But in the context of the HTTP protocol, it appears to have been a red herring. Clients and proxies should not care how the server comes up with the bits that make up an instance; they should care only what the server means by those bits.

So while conventional practice, especially in client or proxy cache implementations, is to treat cautiously anything that might be a dynamic resource (e.g., a URL containing a ‘‘?’’ or the string ‘‘cgi-bin’’), this would not be necessary if HTTP instances were explicitly labeled with sufficient detail. (With minor exceptions, RFC2616 does not require special handling for dynamic resources; the distinction is primarily folklore.)

It might indeed be useful for a client or proxy to know whether a particular instance is expensive for the server to generate. We could introduce an instance label stating the server’s time to regenerate the instance. This might, for example, affect a cache replacement policy. However, this measure is orthogonal to the static/dynamic axis (i.e., a ‘‘static’’ disk access might be more expensive than a short ‘‘dynamic’’ thread execution). In fact, it makes little sense that, in current practice, responses requiring the most server costs to regenerate are exactly the ones that caches do not store.

5.4. Resources vs. Pages

I complained earlier that people often use the term ‘‘document’’ or ‘‘page’’ when they really mean ‘‘resource’’ or ‘‘instance.’’ This split, between the terms used by specification writers and the terms used by almost everyone else, while problematic, partially reflects reality rather than simple ignorance. While HTTP would have been far more complicated if it had directly supported multi-resource pages, instead of leaving that to a higher layer, browsers and their users ultimately care about the page rendered, not its atomic parts.

HTTP’s lack of support for page-oriented operations can affect users in ways that complicate their interactions with the Web. For example, since all HTTP caching mechanisms are defined with respect to resources (or instances), not pages, HTTP provides no mechanism either to ensure that all cached elements of a page are consistent, or even to detect when they are inconsistent. In principle, this could lead to corrupted pages, although in practice the problem seems quite rare. It also means that when one uses the ‘‘Reload’’ button to refresh one image on a page, one cannot avoid reloading all of the images.

More problematic, and confusing, is the interaction between caching and browser ‘‘history’’ mechanisms (which support the ‘‘Back’’ and ‘‘Forward’’ buttons). A history mechanism should lead you to the page view you actually saw in the past, not to a current (and hence cache-coherent) view of the document. (For more discussion of history mechanisms, see section 7.3.) But because most browsers use their caches to store history data, one can end up seeing the old HTML with new images, or vice versa.

The WebDAV extensions to HTTP [13] provide facilities for managing collections of resources in the context of distributed content authoring, but do not appear to address the problem of maintaining the relationship between a page and its resources in a typical browsing application.

6. HTTP extensibility

The Web spread so widely and rapidly largely because its main components (HTTP and HTML) emphasize interoperation. The Web has managed to evolve to support new features only because these same components are easily extended. Interoperability and extensibility can work at cross purposes: too much extension can damage interoperability; too much emphasis on interoperability can freeze out valuable extensions.

The extension mechanisms in HTTP have been the subject of significant discussion and several false starts. While HTTP has always had one powerful extension mechanism (the requirement that implementations must ignore headers that they don’t understand, without generating errors), the protocol lacks support for complex extensions.

The traditional HTTP extension mechanism has been for the client to send a header indicating its support for a feature, such as response data compression, and the server to use that feature only upon receipt of such a header. For example, a client could send ‘‘Accept-Encoding: gzip, compress’’, allowing the server to use ‘‘Content-Encoding: gzip’’ in its response.

Early HTTP clients sent long lists of their capabilities in Accept* headers. This seemed wasteful, and in current practice, browsers omit many ‘‘obvious’’ content-types, such as text/plain and text/html (although the HTTP specification does not include any such default values). Server implementors, in turn, have learned to key their responses based on the User-Agent header, rather than the Accept headers, for lack of any other information. (Or, they use Javascript that tries to tailor the HTML to the specific browser.)

Therefore, efficiency considerations further complicate the problem: an expressive extension mechanism becomes disused because it costs too much, to be replaced by a cheaper mechanism that can severely limit interoperability.

Similarly, while a server could send extra headers in its responses to indicate what extensions it supports (e.g., the existing Accept-Ranges header), this is inefficient for a server supporting a wide variety of extensions.

6.1. Looking for an Extension Mechanism

HTTP needs an extension mechanism that explicitly and unambiguously identifies the capabilities of implementations, and that is efficient both in its use of bytes and its use of network round trips. Several mechanism have already been proposed; do any of them meet these requirements?

6.1.1. Protocol version numbers

One obvious approach would be to use a protocol version number to indicate implementation capabilities. In fact, the HTTP/1.1 specification assumes that certain capabilities are associated with the version number. Unfortunately, many HTTP implementations send meaningless version numbers, either because they were deployed before the specification was finished, or (for some proxies) because they incorrectly forward version numbers from incoming messages [26].

Even if we could rely on correct implementations, the use of protocol version numbers to indicate feature support does not match the way by which HTTP gains features. Many of the warts in the specification resulted from the co-evolution of the applications people invented to exploit HTTP, the client and server implementation innovations added to support those inventions, and the protocol features created to rationalize those innovations. The HTTP/1.1 specification [10] is a fairly arbitrary snapshot in this evolutionary sequence, not an intrinsically stable point.

One could increment the version number frequently enough to capture the rate at which features are proposed. But this imposes a partial ordering on capabilities (or else a version-*X* system cannot make any assumptions about a version-*X+1* system). Such a partial ordering would be too burdensome for extension designs.

6.1.2. Explicit extension naming

HTTP/1.1 introduced an OPTIONS method to ‘‘request information about the communication options available on the request/response chain identified by the Request-URI.’’ There has never been a clear specification of what information OPTIONS returns, how it is encoded, or how it names optional features. (Some implementations simply send an ‘‘Allow’’ header, which only lists the methods supported, not other optional features.)

In fact, the central issue in designing a general HTTP extension mechanism is how to name extensions. One series of proposals culminated in RFC2774 [27], which never entered the IETF Standards Track. In this model, each extension is named using a URI such as ‘‘http://example.com/extension’’, which allows each extension-designer to control a private name. (This approach depends on the stability of the organization owning the DNS name in the URI.) RFC2774 provides additional mechanisms for reserving names for message headers and for allowing multiple, independently-defined extensions to co-exist in one message. In this approach, introduction of a new extension does not depend on a standardization process.

6.1.3. RFC numbers as extension names

One alternative is to consider only how to add those extensions created by a centralized standards body. Although this set is potentially much smaller than could be supported by RFC2774, it is still not adequately supported by HTTP/1.1. The traditional HTTP approach (send a feature-specific header and see if you get something relevant back) can discover if two implementations both implement a given header name, but it cannot guarantee that they agree on what the header *means*.

Josh Cohen, Scott Lawrence, and I proposed [23] a simple mechanism to resolve this problem: the use of IETF RFC numbers as the extension name space. The IETF ensures that RFCs are well-specified and immutable, an RFC number is relatively compact, and the IETF appears to be a stable naming authority. We also proposed adding a ‘‘Compliance’’ header to assert compliance with elements of these name spaces, as well as a ‘‘Non-compliance’’ header for proxies along a path to indicate a lack of end-to-end support.

In such a declaratory (rather than negotiation-based) approach, one risks sending lengthy Compliance headers, listing lots of extension identifiers. However, while there might be many registered extensions, in practice most implementations would support one of a relatively small number of distinct subsets of extensions. Each subset would be a sort of ‘‘profile’’ (a term often used to describe an agreed-upon set of protocol features). The Compliance header could therefore be used to list subsets, using another compact and centrally-managed namespace. Or, one could avoid centralization by sending a hash value based on the elements of a subset, falling back to a negotiation mechanism to transfer the actual list of extensions if the subset has not been seen before (this approach was proposed by Klyne and Masinter for abbreviation of ‘‘feature sets’’ [17], although not ultimately adopted).

6.2. Summary: Extension Mechanisms

From the discussion above, one can crudely divide HTTP extension proposals into three categories:

- **Trial and error:** send an extension-specific header and see if you get something useful back.
- **Negotiate:** ask the other end what it supports, then choose the best option.
- **Declare capabilities or profile:** always say what extensions you support, and let the other end decide whether to exploit them.

The first (trial-and-error) approach is informal, but widely supported. Currently, HTTP has no formal extension mechanism; the negotiation-based approaches have proved too complex for most tastes, while many people dislike centralized name spaces. However, the centralized-name-space approach seems simpler to specify and understand.

7. Other issues

Space constraints do not permit comprehensive discussion of many other unresolved or unclear aspects of HTTP. Here I briefly describe a few of these issues, to show how they tie in with the rest of this paper.

7.1. Variants

A truly ‘‘World-Wide’’ Web must support the use of many natural languages and character sets. One of the most prominently proclaimed features in the first public draft of the HTTP specification [2] was the ‘‘the negotiation of data representation, allowing systems to be built independently of the development of new advanced representations.’’ One goal behind this ‘‘content negotiation’’ mechanism is to allow a single URL to automatically serve the same content in the appropriate natural language for any user. Content negotiation in the current specification can also apply to other dimensions, including content-encoding or presentation issues such as display screen size.

The use of content negotiation means that a given URL is not simply a name for a specific piece of data. (The translation of a text in one language to a different language clearly is not now, and may never be, an automatable one-to-one mapping; think of how hard it is to translate puns and other wordplay.) Instead, HTTP defines the term ‘‘variant’’; a given URL might have multiple variants. The content negotiation mechanism is used to select the best variant of a URL, given the preferences of the user (client) and the content-provider (server).

Variants create immense complications for almost all of the issues discussed in this paper, especially caching. Space does not permit even a minimal discussion of how one might bring some clarity to variants, and I’m not sure that anyone knows how to do that yet.

7.2. Regularizing the Use of Intermediaries

Much of the Internet’s success depends on placing sophisticated processing at end hosts, not in the infrastructure. This ‘‘end-to-end argument’’ [30] is the object of veneration, debate, and some critical analysis [3], as the use of intermediary systems becomes more prevalent.

The Web not only supports intermediaries, but in some ways depends on them for its success. HTTP directly specifies the behavior of proxies, especially for caching, but they are also widely used for access control, transcoding, server availability, and the deployment of new protocol features.

While the HTTP specification treats proxies as explicit agents, it also assumes (for the most part) that their role is semantically transparent. However, many newer intermediary functions (such as transcoding to support the use of handheld clients) can radically change forwarded content, a feature that HTTP does not really grapple with. At the same time, many proxies take transparency to an extreme, making themselves invisible to end systems. This can lead to confusion when errors occur.

HTTP needs a more regular and comprehensive approach to intermediaries. The Internet Architecture Board has issued some policy recommendations in this area [16], and elsewhere I have suggested a new approach to proxy-based transcoding [22].

7.3. Protocol Support for User Interface Concerns

We like to think that there is a clear boundary between the HTTP protocol and the user interface of an HTTP implementation: user-interface concerns are outside the scope of the protocol design. In one way, this separation is intrinsic; some HTTP clients have no user interface. But most applications of HTTP involve an interactive human user, and in reality, the boundary between user-interface concerns and protocol concerns cannot be ignored.

The existing specification already places some constraints on the user interface. HTTP/1.1 recommends (but does not require) that user agents display Warning headers [10] (section 14.46), and requires that a user agent can be configured never to send Cookier headers [19] (section 6.1). The Secure HTTP specification requires browsers to ‘‘provide a visual indication of the security of the transaction’’ [29] (section 6.3.1), typically displayed as a lock icon. However, these constraints are phrased timidly, as if this were inappropriate for a protocol specification.

As a result, we are again stuck in a situation where service designers are forced to rely on inferences about poorly specified user-interface features, instead of on explicit protocol support.

For example, RFC2616 makes a clear delineation between client caches and client history mechanisms (‘‘back’’ and ‘‘forward’’ buttons) [10] (section 13.13), which were discussed in section 5.4. Most popular browsers violate this part of the specification, primarily for implementation expedience. However, browser implementors have also faced pressure to make history entries obey cache-related HTTP directives (contrary to the specification), such as ‘‘Cache-control: no-store’’, in order to meet certain (possibly misguided) expectations about security. And certain pages warn ‘‘Do not use the back button,’’ putting the onus on the user to avoid semantic confusion caused by the history mechanism.

A better approach would be for the HTTP protocol to provide explicit support for any necessary server control over history functions, rather than overloading the cache-related protocol features or burdening users.

8. Related work

Most work on HTTP, and on other Web protocols and data formats, has focused on solving specific problems, usually in the context of standards committees. Academic researchers, on the other hand, have generally taken the protocol as a given, not treating it as worthy of direct study. However, a few people have taken a step back to look at the larger protocol design issues.

Fielding and Taylor developed an idealized model for interactions in the Web [11], which is more abstract than the discussion of this paper. They point out that HTTP fails to match their model, and mention the failure to sufficiently distinguish between various types of HTTP headers. Their model does not address the detailed issues of developing a data type model. They write that ‘‘entity, instance, or variant’’ are ‘‘less precise names’’ for what they call a ‘‘representation,’’ apparently ignoring the problem that trying to subsume these terms under the more generic ‘‘representation’’ obscures useful distinctions.

Eastlake has written on how protocol designers can take either a ‘‘protocol’’ or ‘‘document’’ point of view during the design process [7]. He implies that one needs to honor both points of view. In particular, a purely ‘‘document’’ view can lead to the omission of important details.

Baker attempts to define an abstract model of how HTTP methods affect the state of resources [1]. In his model, ‘‘static’’ resources are containers for a single (non-composite) immutable piece of data. Resources accepting PUT but not POST are containers for single, mutable pieces of data. Certain resources accepting POST are containers for composites of several data items. Using this model, Baker proposes new descriptions for HTTP methods that, while consistent with the existing method specifications, are intended to be more easily understood. One problem with Baker’s model is that most ‘‘static’’ Web resources are not immutable [6], which implies that his model needs elaboration.

The efforts listed above, and this paper, have been aimed at improving the existing HTTP design. Several groups have suggested a clean-slate redesign, under names such as ‘‘HTTP Next Generation (HTTP-NG)’’ [28]. None of these efforts have born fruit; the existing HTTP design, albeit flawed, works well enough to discourage revolutionary changes.

9. Future work

A short paper such as this one cannot include all of the conceptual problems afflicting HTTP now, or as it evolves further. I have no idea how to solve many of these problems. Here I suggest some areas for future work.

The improved data type model that I have described in this paper has been developed while working on the existing protocol, and on a few extensions that have received a lot of scrutiny. The model should be further tested on additional HTTP extensions (e.g., pending work on coherent caching [20], or the metadata mechanisms used in some CDNs to track the validity or mutability of information), to ensure that it is robust enough to support extensions that I have not considered.

One impetus for the improved model was to clarify how compression and byte-range selection could be composed. It would be useful to test whether the model is helpful in enabling implementors to understand, without having to consult protocol experts, how to compose other features in ways not explicitly described in the HTTP specification.

My goals in writing this paper explicitly excluded replacing HTTP with a clean-slate design. However, attempting such a design, while preserving as much as possible of the existing flavor of HTTP, might be an illuminating exercise. Section 4.2 sketched how one might convert HTTP to a more cleanly layered design. Section 5.2 sketched how one might add more explicit labeling to the data access model. Section 6.1.3 showed how a central authority could be a simple way to name extensions. A full clean-slate design would certainly include other changes.

10. Summary

In this paper I have attempted to show the need for more rigorous fundamental models for HTTP, and I have sketched in some of the details. While it might not be possible to compatibly resolve all the problems with the existing protocol, such an effort would provide guidance to designers of a follow-on protocol.

I have made several specific recommendations:

- **HTTP needs a clean and consistent data type model:** By thinking in terms of a message-generation pipeline with well-defined stages and processing steps, we can clarify many issues of HTTP, especially caching, the handling of partial results, and the categorization of header fields.
- **HTTP needs an explicit ‘‘instance’’ data type:** One cannot construct a consistent message-generation pipeline without introducing a new data type. Because HTTP caches store instances (rather than entities or messages), this change greatly simplifies many protocol concepts.
- **HTTP needs a clear data access model**: We need a framework to discuss what kinds of data items HTTP operates upon, and what operations it can apply.
- **Resources and instances should carry explicit access-model labels**: Explicit labeling, rather than heuristic inferences, allows automated clients and caches to correctly deal with mutability, side effects, idempotence, and response-generation costs.
- **Create a simple name space for implementations to declare sets of supported extensions**: Use ‘‘profiles’’ to limit the overhead of supporting many extensions.

More generally, I have tried to show how careful consideration of the existing HTTP protocol can reveal regularities, or near-regularities, that should be exploited to improve our understanding, implementations, and extension designs.

11. Conclusions

If we have a clearer idea of how to think about HTTP, shouldn't we be able to simplify the protocol? Given the goal of working within the constraints imposed by the installed base, we cannot actually remove features from the existing design.

We should, however, be able to clarify the protocol specification. This might not actually shorten the specification, since the clarification effort (which would be a major undertaking) might reveal ambiguities that need additional treatment. Interoperability suffers far more damage from ambiguity than from verbosity.

We should certainly expect any proposal for a future extension design to explain either how it fits into a consistent design model for HTTP, or how the model can be consistently extended to support the extension.

Many in the HTTP community have resisted formality at this level, either because they think it unnecessary or because they expect it to be too confining. My belief is that the lack of rigor and clarity in the specification stifles innovation. Just as CPU designers are freed to innovate at the implementation level when they are sure that the instruction set architecture is rigorously defined, HTTP implementors (and extension designers) will gain freedom if the protocol is unambiguous.

Acknowledgments

I have benefited from many helpful comments from Mark Baker, Eric Brewer, Terence Kelly, Balachander Krishnamurthy, and Koen Holtman, as well as from the anonymous reviewers.

12. References

Note: several of the references below are ''Internet Drafts.'' By IETF policy,

> Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

While these restrictions are reasonable as part of the IETF standards process, in the context of establishing priority for an idea, it would be unethical to fail to cite a relevant Internet-Draft. The reader should be aware that some of drafts cited below may have been superseded, and that expired drafts are often hard to obtain.

[1] Mark Baker. *An Abstract Model for HTTP Resource State*. Internet-Draft draft-baker-http-resource-state-model-01.txt, IETF, November, 2001. This is a work in progress. http://www.ietf.org/internet-drafts/draft-baker-http-resource-state-model-01.txt.

[2] Tim Berners-Lee. *Hypertext Transfer Protocol (HTTP)*. Internet Draft draft-ietf-iiir-http-00.txt, IETF, November, 1993. This is a work in progress. ftp://ftp.std.com/obi/Networking/WWW/draft-ietf-iiir-http-00.txt.

[3] M. Blumenthal and D. Clark. Rethinking the design of the Internet: The end to end arguments vs. the brave new world. *ACM Trans. Internet Technology* 1(1):70-109, August, 2001. http://www.ana.lcs.mit.edu/anaweb/PDF/Rethinking_2001.pdf.

[4] Mun Choon Chan and Thomas Woo. Cache-based Compaction: A New Technique for Optimizing Web Transfer. In *Proc. IEEE Infocom '99*, pp. 117-125. New York, NY, March, 1999.

[5] John Dilley. The Effect of Consistency on Cache Response Time. *IEEE Network* 14(3):24-28, May/June, 2000. http://www.comsoc.org/ni/private/2000/may/Dilley.html.

[6] 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. Symposium on Internet Technologies and Systems*, pp. 147-158. USENIX, Monterey, CA, December, 1997. www.usenix.org/publications/library/proceedings/usits97/douglis_rate.html.

[7] Donald E. Eastlake 3rd. *Protocol versus Document Points of View*. Internet-Draft draft-eastlake-proto-doc-pov-04.txt, IETF, September, 2001. This is a work in progress. http://www.ietf.org/internet-drafts/draft-eastlake-proto-doc-pov-04.txt.

[8] Albert Einstein. Widely attributed quotation. (Various forms of this quotation are attributed to Einstein).

[9] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2068, HTTP Working Group, January, 1997. http://www.ietf.org/rfc/rfc2068.txt.

[10] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616, HTTP Working Group, June, 1999. ftp://ftp.isi.edu/in-notes/rfc2616.txt.

[11] Roy T. Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. In *Proc. 22nd International Conf. on Software Engineering*, pp. 407-416. Limerick, Ireland, June, 2000. http://www.acm.org/pubs/citations/proceedings/soft/337180/p407-fielding/.

[12] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC 2045, Network Working Group, November, 1996.

[13] Y. Goland, E. Whitehead, Jr, A. Faizi, S. Carter, D. Jensen. *HTTP Extensions for Distributed Authoring -- WEBDAV*. RFC 2518, IETF, February, 1999. ftp://ftp.isi.edu/in-notes/rfc2518.txt.

[14] Koen Holtman. *The Safe Response Header Field*. RFC 2310, IETF, April, 1998. ftp://ftp.isi.edu/in-notes/rfc2310.txt.

[15] Barron C. Housel and David B. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. In *Proc. 2nd Annual Intl. Conf. on Mobile Computing and Networking*, pp. 108-116. ACM, Rye, New York, November, 1996.

[16] Internet Architecture Board. *IAB Architectural and Policy Considerations for OPES*. Internet Draft draft-iab-opes-01.txt, IETF, October, 2001. This is a work in progress. http://www.ietf.org/internet-drafts/draft-iab-opes-01.txt.

[17] Graham Klyne and Larry Masinter. *Identifying composite media features*. Internet Draft draft-ietf-conneg-feature-hash-03.txt, IETF CONNEG Working Group, July, 1999. This is a work in progress. http://www1.ics.uci.edu/pub/ietf/http/draft-ietf-conneg-feature-hash-03.txt.

[18] Balachander Krishnamurthy and Martin Arlitt. PRO-COW: Protocol Compliance on the Web. In *Proc. USENIX Symposium on Internet Technology and Systems*, pp. 109-122. San Francisco, CA, March, 2001. http://www.research.att.com/~bala/papers/usits01.ps.gz.

[19] David M. Kristol and Lou Montulli. *HTTP State Management Mechanism*. RFC 2965, IETF, October, 2000. http://www.ietf.org/rfc/rfc2965.txt.

[20] Dan Li, Pei Cao, and Mike Dahlin. *WCIP: Web Cache Invalidation Protocol*. Internet Draft draft-danli-wrec-wcip-01.txt, IETF, March, 2001. This is a work in progress. http://www.ietf.org/internet-drafts/draft-danli-wrec-wcip-01.txt.

[21] Merriam-Webster. *Webster's Seventh New Collegiate Dictionary*. G. & C. Merriam Co., Springfield, MA, 1963.

[22] Jeffrey C. Mogul. Server-Directed Transcoding. *Computer Communications* 24(2):155-162, February, 2001. http://www.terena.nl/conf/wcw/Proceedings/S1/S1-4.ps.

[23] Jeffrey Mogul, Josh Cohen, and Scott Lawrence. *Specification of HTTP/1.1 OPTIONS messages*. Internet Draft draft-ietf-http-options-02.txt, HTTP Working Group, August, 1997. This is a work in progress. http://www1.ics.uci.edu/pub/ietf/http/draft-ietf-http-options-02.txt.

[24] J. C. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. *Delta encoding in HTTP*. RFC 3229, IETF, January, 2002.

[25] 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*, pp. 181-194. ACM SIGCOMM, Cannes, France, September, 1997.

[26] Jeffrey C. Mogul, Roy T. Fielding, Jim Gettys, Henrik Frystyk Nielsen. *Use and Interpretation of HTTP Version Numbers*. RFC 2145, HTTP Working Group, May, 1997. http://www.ietf.org/rfc/rfc2145.txt.

[27] Henrik Frystyk Nielsen, Paul J. Leach, and Scott Lawrence. *An HTTP Extension Framework*. RFC 2774, IETF, February, 2000. ftp://ftp.isi.edu/in-notes/rfc2774.txt.

[28] Henrik Frystyk Nielsen, Mike Spreitzer, Bill Janssen, and Jim Gettys. *HTTP-NG Overview: Problem Statement, Requirements, and Solution Outline*. Internet Draft draft-frystyk-httpng-overview-00.txt, IETF, November, 1998. This is a work in progress. http://www.w3.org/Protocols/HTTP-NG/1998/11/draft-frystyk-httpng-overview-00.

[29] Eric Rescorla and Allan M. Schiffman. *The Secure HyperText Transfer Protocol*. RFC 2660, IETF, August, 1999. http://www.ietf.org/rfc/rfc2660.txt.

[30] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Trans. Computer Systems* 2(4):277-288, November, 1984. http://www.ana.lcs.mit.edu/anaweb/PDF/saltzer_reed_clark_e2e.pdf.

[31] Andrew Tridgell and Paul Mackerras. *The rsync algorithm*. Technical Report TR-CS-96-05, Dept. of Computer Science, Australian National University, June, 1996. http://cs.anu.edu.au/techreports/1996/TR-CS-96-05.html.