David M. Kristol
AT&T Bell Laboratories

INTERNET DRAFT
<draft-kristol-http-extensions-00.ps>
January 1995
Expires July 1995

A Proposed Extension Mechanism for HTTP

This is author's draft 2.9. The previously available author's draft was 1.3.

1. ABSTRACT

HTTP, the hypertext transfer protocol, underpins the World-Wide Web (WWW). As the Web has grown, pressures have mounted to add a variety of facilities to HTTP. Some of the new features that have been proposed include: keep-alive, packetized data, compression, security, and payment. This memo offers an alternative: well-defined hooks in a slightly modified HTTP framework that make it possible to add extensions to the basic protocol in a way that will retain compatible behavior between clients and servers, yet allow both clients and servers to discover and use extended capabilities. The goal is to use HTTP as just a transport mechanism, leaving other, higher-level (session) activities to extensions.

2. MOTIVATION

One virtue of HTTP is that it is easy to modify: just add more request or response headers. Unrecognized headers will be ignored by agents (client or server) that don't understand them. Why is this approach unacceptable? The following paragraphs will attempt to justify a different approach.

2.1 Generality

A common, well-defined framework by which to introduce extensions reduces the danger of uncontrolled proliferation of incompatible extensions. Vendors that want to add extensions can do so in a predictable way. Client and server software will be better able to predict what kinds of headers they will encounter.

2.2 Simplicity and Modularity

The server and client architectures should be kept simple. If extensions can be recognized easily, it becomes possible to posit the following architecture. A client (server) comprises a core part. Extensions are handed off to an *extensions manager*, which dispatches extension requests to the appropriate handler, such as a *security manager*, *payment manager*, etc. If the interface between the core part and the extensions manager, and between the extensions manager and the individual extension managers, is well-defined, the core part of a client (server) can be quite ignorant of what is actually being done by the various extensions. Thus this architecture leads to a highly modular design into which it is possible to ''plug'' new extensions, while the core part remains simple. Vendors of extensions could supply matched plug-in parts for them to client and server implementors, to incorporate them in their products.

2.3 Recognizable Extensions

The scheme proposed here makes it easy to identify requests for extensions quickly. Requests for, or acceptance of, extensions is signaled by `Extension:` request/response headers. Because they are easily identified, a caching server can recognize headers for extensions and store them as part of the cached information.

(Caching requires further consideration. It may be necessary for caching servers not to cache information that was obtained using extensions, since those extensions often entail security or payment.)

Using the HTTP version number to determine what extensions are present is a bad idea. Extensions are often disjoint, and clients and servers should be able to ‘‘mix and match’’ the ones they can support. The HTTP version number is too crude a discriminant and should be reserved for true changes to the base protocol. The extension mechanism proposed here merely uses HTTP for transport.

2.4 Who’s in Charge?

HTTP headers are likely to require approval from IANA. Thus, using multiple headers for different extensions may impede new applications of the World-Wide Web. It is easier to get two headers approved that enable a whole range of extensions, and it carves out a hunk of name space that can be controlled separately, possibly by W3C (World-Wide Web Consortium).

2.5 Wrapping Better Than MIME

Wrapping, described below, has capabilities that MIME cannot supply. Specifically, security may require that the actual request be obscured. The `WRAPPED` method in this proposal makes it possible to encrypt the actual request. A snooper would see only the `WRAPPED` request method and the extension header (with, presumably, enough information to describe how to decrypt the actual request).

3. CONCEPTS

The proposed extension mechanism has two fundamental concepts: *wrapping* and *negotiation*.

3.1 Wrapping

Wrapping implies that a core set of information has information added before and after it. In some cases the information added may comprise just headers. The core information may itself be changed as well, such as when it is encrypted. The information added as the pre-wrapper must convey enough information to the recipient so it can unwrap the information. Either a client or server can do the wrapping, although I assume that the server more often does the wrapping.

3.2 Negotiation

Before a sender wraps information, it must be sure the receiver can unwrap it. Thus the two parties must negotiate what kind of wrapping is desirable. Therefore a (prospective) receiver (*initiator*) tells a sender (*responder*) what forms of wrapping it accepts, and whether they are acceptable always, or sometimes. The sender responds with a suitably wrapped response. ‘‘One-of’’ wrappings must always be used, but the sender can choose at will from a group of such wrappings. ‘‘Sometimes’’ wrappings may be used at the discretion of the responder.

3.3 Recursion

Wrappings can be recursive. To give but one example, a core message might be wrapped thus: packetize(security(payment(core))). The notation implies that, given a core message, first a payment wrapping is applied, followed by a security wrapper, followed by packetization. The receiver must unwrap the message from outside in, i.e., packetization first.

4. NOTATIONAL CONVENTIONS

This proposal uses the notational conventions of the (draft) specification, *Hypertext Transfer Protocol — HTTP/1.0*, Berners-Lee, Fielding, Frystyk Nielsen.

5. EXTENSIONS TO HTTP 1.0

The proposed extension mechanism requires small changes in the existing HTTP, to add two methods and two request headers.

5.1 Methods

5.1.1 GETEXT get extensions

The `HEAD` method in HTTP only provides a limited amount of information about how the server would respond to a `GET` request. `GETEXT` provides information about how the server would respond (at least for extensions) for any method.

```
<getext>      ::=     "GETEXT" <url> "HTTP/1.0"
<url>         ::=     *
              /       <path part of URL>
```

A client sends a request with the `GETEXT` method to a server to learn what extensions the server supports.

The `GETEXT` method may have an optional `Extension:` request header with the following form:

```
<getext-hdr>  ::=     "Extension:" "HTTP/Method" "method=" <method>
<method>      ::=     "GET" / "PUT" / "POST" / ...
```

Semantics

The server reports the extensions as response headers, identical to the request headers that are described in the next section, that apply. There are four cases, depending on whether an explicit <url> is present and whether an `Extension:` header is present. In each case the result is the intersection set of extensions for the method(s) and URL(s) specified by the `GETEXT` request.

<url> ::= *, no `Extension:` request header
: The server returns the set of `Extension:` response headers that apply to any URL and any method on the server.

<url> ::= *, `Extension:` request header present
: The server returns the set of `Extension:` response headers that would apply if the server got a request comprising the <method> in the `Extension:` request header and any URL.

<url> ::= URL, no `Extension:` request header
: The server returns the set of `Extension:` response headers that would apply if the server got a request comprising any acceptable method applied to the selected URL.

<url> ::= URL, `Extension:` request header present
: The server returns the set of `Extension:` response headers that would apply if the server got a request comprising the <method> in the `Extension:` request header and the specified URL.

5.2.1 WRAPPED wrapped request

```
<wrapped>     ::=     "WRAPPED" "*" "HTTP/1.0"
```

A client sends a request with this method to tell a server that the request is wrapped. The request headers (next section) specify exactly how the request is wrapped. The server must peel the wrappings one layer at a time until it encounters a normal request, which it can then process. The value of what would normally be the URL field, shown above as *, is ignored.

5.3 Request Headers

Two new request headers specify extensions. Their syntax and semantics are given below.

5.3.1 Extension: Header

```
<extension>     ::=   "Extension:"  <cat-class> [<av-pairs>]
<cat-class>     ::=   <category> "/" <class>
<category>      ::=   <alpha-numeric string>        ; kind of generic extension
<class>         ::=   <alpha-numeric string>        ; class within generic extension
<av-pairs>      ::=   <av-pair> [";" <av-pair>]
<av-pair>       ::=   <attr> "=" <value>
<attr>          ::=   <alpha-numeric string>
<value>         ::=   <alpha-numeric string>
```

The Extension: request header may wrap, RFC822-style, onto multiple lines.

Semantics

<category> describes the generic extension, for example *security*, *payment*. (Should the list of extensions be controlled by W3C?) A responder that receives an unrecognized <category> responds with an error if the <av-pair> `required=oneof` is present and ignores the header otherwise.

<class> specifies a member of the <category>. For example, <category> `payment` might have <class>s `Visa`, `MasterCard`, `Ecash`, etc. If a <category> has no <class>s, `*` must be used as the <class>.

Attribute-value pairs (<av-pairs>) are optional. The only attribute that is defined for all <category>s is `required`, with possible values `oneof` and `sometimes`. If the `required` attribute is present for an `Extension:` header, it must be part of the first <av-pair>.

The algorithm for choosing among Extension: headers is described in **NEGOTIATION**, below.

5.4.1 Extension-Order: Header

```
<extension-order>     ::=     "Extension-Order:" 1#<cat-class>
```

The `Extension-Order:` header provides a way for a client or server to specify the order in which extensions must be applied. (MIME headers are unordered by definition.)

5.5 Wrapped Response

A wrapped response comprises a status code of 207 (*Wrapped*), a single `Extension:` response header that describes the outermost wrapping, and a blank line. The body of the response comprises the wrapped response. The wrapping must be defined in such a way that the body can be unwrapped to produce a new response, complete with status line, response header line(s), blank line, and new body. The specifics of that wrapping are outside the scope of this document and are specific to a given extension. The resulting inner response may also be a wrapped response, in which case the unwrapping occurs recursively.

A response code of 409 by a server denotes a *Wrapping Required* response. The response headers specify both the type and order of wrapping that the server requires from the client. (Note that in this case the server acts as an initiator and should provide `required=` <av-pair>s for the `Extension:` response headers it returns.)

5.5.1 Response Headers An `Extension:` response header is identical in form and content to a corresponding request header. A responder's header does not contain a `required=` <av-pair>, because the presence of the `Extension:` header means the extension request has been honored, whether it was optional or mandatory.

6. NEGOTIATION

This section describes in more detail the negotiation process. It describes the roles of *initiators* and *responders* and how they should use and respond to `Extension:` and `Extension-Order:` headers. Either clients or servers may be *initiators* or *responders*.

[This first attempt will undoubtedly be mushy.]

6.1 Initiators

An initiator informs a responder of extension capabilities that are either mandatory or optional to use. Each capability is described by an `Extension:` (request or response) header. Mandatory extensions must have `required=oneof` as the first <av-pair> of the header. If no `required=` <av-pair> is present in an initiator's header, the default is `required=sometimes`.

An initiator can advertise to a responder many <class>s for a particular <category>, and many extension <category>s. Based on the negotiation selection algorithm (below), a responder may (and sometimes must) choose among them. How the initiator chooses which extensions to advertise is outside the scope of this proposal.

6.2 Responders

A responder informs an initiator of which extensions it has selected. If the initiator has specified multiple <class>s for a particular <category>, the responder must choose to honor either one or none of the choices. When the initiator has specified mandatory extensions (`required=oneof`) for a <category>, the responder must choose one. For optional extensions (`required=sometimes`), the responder informs the initiator of which ones, if any, it has chosen to use.

What happens when a responder cannot honor a mandatory extension depends on whether it is a client or server. A client responder would typically inform a user that it cannot complete a request because it lacks some capability that the server requires (such as authentication or a suitable payment method). A server responder would return an error response code to the client, preferably with an informative HTML message for the client to display to describe the failure.

6.3 Negotiation Selection Algorithm

A responder uses the negotiation selection algorithm to choose among possible <class>s in a given <category>, and to define the order in which extensions in different <category>s are applied.

For expository purposes, assume each <cat-class> part of an `Extension-Order:` header is assigned an integer index from 1 (first) to N (last), based on its lexical order in the header. If there is no `Extension-Order:` header, the indexes are assigned arbitrarily by the responder. Note that if there is only one `Extension:` header, the `Extension-Order:` header is redundant.

6.3.1 Selection All the extensions in the same <category> (call them *category-options*) are treated as a group. If any *category-option* in the group has the <av-pair> `required=oneof` (a mandatory category), then any *category-option* in the group that does not have `required=oneof` is discarded. The responder then considers each remaining *category-option* in turn, choosing the one it prefers from the group. Note that at this point all the members of the group must have the same `required=` value, either `oneof` or `sometimes`. (How the responder chooses the one ‘‘it prefers’’ is an implementation issue, outside the scope of this proposal.)

If the responder cannot support any of the *category-options* for a mandatory category, it responds with an error, as indicated earlier. For an optional category, if the responder cannot support, or chooses not to honor, an extension, the responder simply does not apply the extension.

6.3.2 Order After the responder has identified the extensions it will honor, based on the initiator's `Extension:` headers, it applies them in the order of the index numbers of the honored <cat-class>s.

6.3.3 Exceptional Conditions

1. If an `Extension-Order:` header is present, any `Extension:` header to be considered in the selection process must have its <cat-class> listed by the `Extension-Order:` header.

2. If more than one `Exception:` header has the same <cat-class>, the responder should reject the request.

3. If a <cat-class> appears in an `Exception-Order:` header, and there is no `Exception:` header with the same <cat-class>, the responder should behave as though the <cat-class> were omitted from the `Exception-Order:` header.

7. EXAMPLE 1: SIMPLE AUTHENTICATION

The Basic authentication scheme can be recast in the framework of the HTTP extension mechanism. Assume the requested (local) URL on the server is /foo/bar.

7.1 Client -> Server

```
GET /foo/bar HTTP/1.0
Accept:     text/html, ...
[blank line]
```

7.2 Server -> Client [Server is initiator]

```
HTTP/1.0 409 Wrapping Required
Date: Wed, 28 Dec 1994 10:59:13 GMT
Server: HTD/0.9
MIME-version: 1.0
Extension: Security/Basic required=oneof;Realm="Demonstration"
```

7.3 Client -> Server [Client is responder]

```
WRAPPED * HTTP/1.0
Extension: Security/Basic data=ZG1rOnBhc3N3b3Jk
[blank line]
====
GET /foo/bar HTTP/1.0
Accept:     text/html, ...
====
```

(Note: pretend ==== is used to ‘‘wrap’’ the inner request.)

7.4 Server -> Client

```
HTTP/1.0 200 OK
Date: Wed, 28 Dec 1994 10:59:15 GMT
Server: HTD/0.9
MIME-version: 1.0
Content-type: text/html
Last-modified: Thu, 17 Nov 1994 19:35:21 GMT
Content-length: 3729

[body]
```

8. EXAMPLE 2: PAYMENT AND ENCRYPTION

This more extended example demonstrates how the extension mechanism can be used to support both security and payment. In the example, the payment manager relies on an existing security manager to provide the encryption that secures a credit card number.

8.1 Client -> Server

```
GET /foo/bar HTTP/1.0
Accept:     text/html, ...
[blank line]
```

8.2 Server -> Client [Server is initiator]

The server requires both payment and security (encryption). The payment information must be supplied first, followed by the encryption. Note that there is no need for an actual ‘‘wrapping’’ by the payment piece.

```
HTTP/1.0 409 Wrapping Required
Date: Wed, 28 Dec 1994 10:59:13 GMT
Server: HTD/0.9
MIME-version: 1.0
Extension: Security/SimpleDES required=oneof;
            keyname=OpenSesame; nonce=873955
Extension: Security/SHTTP required=oneof;
            (S-HTTP parameters)
Extension: Payment/MasterCharge required=oneof;
            cost=2; units=USD; mult=0
Extension: Payment/GreenCard required=oneof;
            cost=2; units=USD; mult=0
Extension: Payment/American_Excess required=oneof;
            cost=25; units=USD; mult=-1
Extension-Order: Security/SHTTP,
            Security/SimpleDES,
            Payment/MasterCharge,
            Payment/American_Excess,
            Payment/GreenCard
[blank line]
```

8.3 Client -> Server [Client is responder]

Client provides server with payment information. It chooses (presumably with the user's help) which of the three payment methods, MasterCharge, GreenCard, American_Excess, to use. It also chooses between the two security methods. Note that the ‘‘wrapping’’ for the payment part just comprises adding a header. The security piece does wrap the request, however.

```
WRAPPED * HTTP/1.0
Extension: Security/SimpleDES keyname=OpenSesame; nonce=873955
[blank line]
====
GET /foo/bar HTTP/1.0
Accept:     text/html, ...
Extension: Payment/GreenCard cost=2; units=USD; mult=0;
            cardno=11112223333444
====
```

Pretend `====` is used to ‘‘wrap’’ the inner request, and that the inner request is encrypted using simple DES with a key named OpenSesame, a shared secret between the client and server. The wrapping could just as well use MIME multipart syntax, but that's a decision to be made by the particular extension manager.

8.4 Server -> Client

Server accepts payment and returns a receipt to the client. The server could, optionally, wrap the response to protect the receipt.

```
HTTP/1.0 200 OK
Date: Wed, 28 Dec 1994 11:01:56 GMT
Server: HTD/0.9
MIME-version: 1.0
Content-type: text/html
Last-modified: Thu, 17 Nov 1994 19:35:21 GMT
Content-length: 3729
Extension: Payment/GreenCard cost=2; units=USD; mult=0;
            Receipt="You paid $2.  Thank you for using AT&T."

[body]
```

9. EXAMPLE 3: GETEXT

This example shows how the `GETEXT` method might be used.

9.1 Client -> Server

```
GETEXT /foo/bar HTTP/1.0
Extension: HTTP/Method method=GET
[blank line]
```

9.2 Server -> Client

```
HTTP/1.0 200 OK
Date: Wed, 28 Dec 1994 10:59:13 GMT
Server: HTD/0.9
MIME-version: 1.0
Extension: Security/SimpleDES required=oneof;
            keyname=OpenSesame; nonce=873955
Extension: Security/SHTTP required=oneof;
            (S-HTTP parameters)
Extension: Payment/MasterCharge required=oneof;
            cost=2; units=USD; mult=0
Extension: Payment/GreenCard required=oneof;
            cost=2; units=USD; mult=0
Extension: Payment/American_Excess required=oneof;
            cost=25; units=USD; mult=-1
Extension-Order: Security/SHTTP,
            Security/SimpleDES,
            Payment/MasterCharge,
            Payment/American_Excess,
            Payment/GreenCard
[blank line]
```

10. <category> `HTTP` Extensions

The proposed mechanism can be used for extensions to HTTP itself, as well as to what might be considered ‘‘outboard’’ matters, like security and payment. Such extensions would comprise the <category> `HTTP`. Two extensions to HTTP/1.0 have been frequently discussed. Both can be accommodated by this proposed mechanism.

10.1 Packetization `HTTP/Packetize`

Packetization divides a message into hunks of not more than a fixed size. It is well-suited for transmitting the output of a script for which the total size of the message is unknown when the script begins to run.

<packet-hdr>	::=	"Extension:" "HTTP/Packetize"
		"required=sometimes" [<size>]
<size>	::=	";" "size=" <positive, non-zero decimal number>

The initiator (usually a client) announces its willingness to accept responses in packetized form, with packets no larger than <size>, but packetization is optional. The initiator must supply <size>.

The responder (usually a server) announces that it plans to use packetization in its response by returning a similar `Extension:` header, omitting the <size> and `required=`.

I have omitted details of what a packetized response looks like. A typical implementation produces packets that comprise a decimal byte count, followed by that many bytes (line terminators included). A zero or negative count signifies end of data.

10.2 Keep-connection `HTTP/KeepConnection`

One of the performance defects of HTTP is that clients often open many connections to the same server to get images for a document they just fetched. Connections are expensive in time and resources. It would be better to pass follow-up requests over an already-open connection.

<keep-conn>	::=	"Extension:" "HTTP/KeepConnection"
		"required=sometimes" [<timeout>]
<timeout>	::=	";" "timeout=" <timeout in seconds>

A client (initiator) would signal its ability and willingness to hold a connection open by passing this request header. The <timeout> advises the server how long it would like the connection held open.

A server (responder) would indicate it had honored the request by returning the same header, minus the `required=` part. The server can respond with a different <timeout> to say how long it plans to hold open the connection, but that information is strictly advisory.

Note that, as a practical matter, `HTTP/KeepConnection` must be used with `HTTP/Packetize`, because in some instances (e.g., output from scripts), the server doesn't know how long its output to the client will be, and it currently signals the end of data by closing the connection. Packetization lets a server pass back data to a client in manageable chunks, rather than buffer the entire response and send a correct `Content-length` header.

11. ACKNOWLEDGEMENTS

Jeff Hostetler, Spyglass, Inc., originally suggested the idea of ‘‘wrapping’’ to me.

12. AUTHOR'S ADDRESS

David M. Kristol
AT&T Bell Laboratories
600 Mountain Ave. Room 2A-227
Murray Hill, NJ 07974
Phone: (908) 582-2250
FAX: (908) 582-5809
Email: dmk@research.att.com