HTTP Working Group — Rohit Khare
INTERNET-DRAFT — World Wide Web Consortium
<draft-khare-http-pep-01.ps> — Massachusetts Institute of Technology
Expires: August 22, 1996 — February 22, 1995

PEP: An Extension Mechanism for HTTP/1.1

Status of this memo

This Internet Draft is also available as W3 Consortium Working Draft `WD-http-pep-950216`. A current version is available at `http://www.w3.org/pub/WWW/TR/WD-http-pep`. This is the second public version, third author's draft.

Abstract

PEP is a system for HTTP clients, servers, and proxies to reliably reason about custom extensions to HTTP. Traditionally, HTTP agents offer extended behavior by adding new message headers, by private agreement. PEP has features for expressing the scope, strength, and ordering of such extensions. PEP also includes an extensible negotiation framework which can not only detect features, but agree on parameters and values as well.

Table of Contents

1. Introduction
2. Concepts
    2.1 PEP Architecture
        2.1.1 Naming
        2.1.2 Addressing
        2.1.3 Negotiation
        2.1.4 Processing
    2.2 Terminology
3. Operation
    3.1 Scenarios
    3.2 Interpretation
    3.3 Negotiation
    3.4 Relaying PEP Messages
    3.5 Deployment Issues
        3.5.1 Relationship to HTTP/1.1
        3.5.2 Protocol Naming and Specification
4. Notation
    4.1 Header Fields
    4.2 Content Codings
    4.3 Status Codes
5. Usage
    5.1 Strength & Scope
    5.2 Payments Negotiation
    5.3 Security Processing
6. Security Considerations
    6.1 Information Leakage
    6.2 Trusting PEP Headers
    6.3 Protocol Interaction Effects
    6.4 Protocol Substitution Effects
    6.5 Negotiation Scope
7. Development Path
8. Acknowledgements
9. References
10. Author’s Address
Appendix A: Minimum Compliance
    A.1 Operating as a Server
    A.2 Operating as a Client
    A.3 Operating as a Proxy
Appendix B: Implementation Model

1. Introduction

HTTP messages, like most applications of RFC 822 [7], can be extended with additional header fields. However, this provides no guidance to HTTP agents on whether to strip the header, or to act on a header, and if acted upon, in what order, and so on. Furthermore, multiple extensions may use conflicting header field names.

Protocol extensions are a higher-level abstraction. They can specify any associated header lines and also provide guidance on each of the above decisions. PEP is an extension protocol for HTTP that captures such information about protocol extensions (hence ‘PEP’: Protocol Extension Protocol).

Using PEP, HTTP agents can interoperate correctly with locally-unknown protocol extensions and negotiate a set of common protocol extensions. PEP adds to HTTP the extensibility lessons learned from ESMTP (extension naming) [15,17], IPv6 (unknown-option disposition) [8], and Telnet (option negotiation) [18].

2. Concepts

PEP is a tool for deploying applications which can be superimposed on HTTP transactions. PEP provides certain core functionality within standard HTTP/1.1: naming, addressing, negotiation, and processing.

First, though, a brief introduction to the headers of PEP for HTTP/1.1. PEP uses the `Protocol:` and `Protocol-Request:` headers to indicate which protocol extensions the current message conforms to and which are requested for subsequent messages, respectively.

> The `Protocol:` header indicates that the HTTP agent generating the message is acting in accordance with the given protocol extension in constructing the message, and the receiver, according to the strength and scope of the extension, may be required to understand the same protocol, or report an error.

Similarly:

> The `Protocol-Request:` header indicates that the HTTP agent generating the message, according to the strength and scope of the extension, requires the receiver to conform to the given protocol extension in constructing replies or subsequent access to specified resources, or report an error.

In many cases, a protocol extension will correspond to a software module; for example, the "`rot13`" protocol is an agreement to shift all characters by 13 positions, one which can be implemented by a filter process. Since applying several extensions in sequence corresponds to a pipeline, PEP reuses an existing HTTP header, `Content-Encoding:` to encode ordering information.

2.1 PEP Architecture

The PEP architecture provides the key elements for reliably extending HTTP. Section 3, ‘‘Operation’’, discusses how these facilities modify agent behavior in more detail.

Note that PEP is completely symmetric; none of the facilites below distinguish between ‘‘client’’ and ‘‘server’’ agents.

2.1.1 Naming

PEP identifies protocol extensions in two contexts: describing a message which employs a protocol, and in describing an request to employ a protocol. Each, in turn, is separated into a protocol identifier, strength and scope configuration, and parameters describing how the protocol is being used.

The protocol identifier is a pointer to a specification of the protocol itself: the full URI of a resource describing the extension. This takes the thinking of ESMTP [15] -- encouraging proliferation of standardized, named extensions -- one step further, by using the web to dynamically register names.

Although URIs can be longer than traditional names, they have many advantages for this application. URIs imply registration of protocols by associating them with the owner of a domain name. It also eliminates the transition phase between experimental and standardized use, the so-called ‘‘`x-` Problem.’’

A protocol specification, in addition to the ‘‘usual’’ material defining the meaning and structure of its payload, can also specify parameters that specify how the protocol is being used (e.g. modes, generic parameters, data types, etc). .

2.1.2 Addressing

While HTTP is a host-to-host protocol, an HTTP transaction is not; it may be routed through proxies, caches, and gateways. PEP offers a mechanism to select which agents are involved in processing each protocol instance or request.

PEP uses a standard configuration attribute, "`scope`", to indicate which HTTP agents are required to pay attention to an instance or request. `Scope` has three values: `conn`ection, `route`, and `origin`. The concept is largely similar to option processing in IPv6 [8], particularly the importance of proper error reporting by agents which are in scope, but cannot handle the option/extension.

Connection

This scope is addressed to the next HTTP agent; each protocol instance or request so labeled must be handled and then removed before passing the resulting message onward. The rationale is similar to the `Connection:` header in Section 10.9 of HTTP/1.1.

Route

This scope is addressed to all HTTP agents in the transaction, to set up synchronous tunnel-type extensions.

Origin

This scope is addressed exclusively to the opposite endpoint (the ‘‘origin server’’ or ‘‘origin client’’). No intermediate agents are allowed to act upon or modify such protocol instances or requests -- unless such an agent is explicitly authorized to act for the origin (e.g. a content-filtering firewall).

Furthermore, PEP explicitly binds protocol requests to specific resource prefixes, which allows requests to be addressed to parts of URI space.

2.1.3 Negotiation

Before an HTTP transaction begins, none of the agents involved can be fully informed of the other agents' capabilities. PEP provides a framework for advertising capabilities and selecting interoperable sets of protocol extensions. Telnet option negotiation [18] is the direct inspiration for the symmetric negotiation model PEP uses. PEP, though, adds a twist, borrowed from S-HTTP [19] of controlling the process by explicitly encoding the "`strength`" of a request.

PEP negotiation handles protocol requests by name (Section 2.1.1, ''Naming'') and by strength. This allows agents to explictly require, reject, or optionally accept particular protocols and parameter constraints.

The only source of asymmetry in HTTP is that the client always moves first. Once the client has listed a set of protocol requests, the server can choose, according to its own preferences, which protocol extensions it will apply, and which of its own to request. Furthermore, the server can indicate applicable requests with URI prefixes to forestall client round-trips upon accessing those resources.

Note that the negotiation proceeds not just on the name, but on the requested parameters, akin to ''subnegotiation'' in Telnet. A particular protocol specification can specify how to reconcile requests and compute responses. For example, consider the `simple-cipher` extension, which defines one parameter, `key-length`. The client can request one range, the server another, and the servers' response would be computed by choosing a value from the intersection.

Finally, agents may choose to respond not with the requested extension, but with an extension considered to be equivalent. This kind of compatibility test allows a *generic* request to be answered by a *specific* instance.

2.1.4 Processing

PEP is designed to accomodate multiple extensions of a single HTTP message. If an extension must be evaluated in a certain order, the protocol instance can define an "`enc`" attribute, whose value can be cited as part of the `Content-Encoding:` pipeline. Protocol extensions that are order-independent need not specify an "`enc`", and are evaluated in arbitrary order after those cited in the pipeline.

PEP does not include a mechanism to order negotiation requests (i.e. ''only accept A after B''), as [13] does.

2.2 Terminology

This following terms have specific meaning in the context of this document. Section 1.3 of the HTTP/1.1 specification [6] defines additional useful terms.

attribute
: a bag whose name is either a complete URI, a token defined in a protocol specification, or a token defined by this document.

bag
: a list consisting of a name and zero or more items, which are tokens, strings, or other bags. See Section 4.1, ''Header Fields'' for the *<bag>* syntax.

connection
: ("`conn`") The scope which addresses the recipient of a message, i.e. the ''next hop''

in the chain of HTTP agents.

encoding
: ("`enc`") An encoding is a process that read or writes the message (header and/or body). The `Content-Encoding:` header is an ordered list of encodings to apply; in addition to the methods defined in HTTP/1.1, an encoding may refer to any protocol instance "`enc`" value.

for list
: ("`for`") In a protocol request, a list of URI prefixes the requirement applies to.

module
: Many protocols will imply complementary processing, which is implemented by a module. A module can be used as a stage of a processing pipeline. Note that a single module could implement several protocols, separately or in combination.

optional
: ("`opt`") A strength value indicating that the associated protocol is optional. As an attribute of an instance, it means that an agent may elide corresonding processing. As an attribute of an request, it means that a reply or subsequent request may be created in accordance with this protocol.

origin
: ("`origin`") The scope which addresses the endpoints of an HTTP transaction; the origin server and the origin client (user). This scope includes proxies trusted to act on behalf of the actual origin.

parameters
: ("`params`") A list of attributes bound to particular protocol instance or protocol request. Attributes are defined in protocol specifications, though they may use differing syntax in protocol instances for negotiable constraints and in protocol instances for configuration values.

PEP-capable HTTP agent
: Any process that communicates according to HTTP with PEP. Any HTTP agent that sends messages with HTTP version number 1.1 or higher must be PEP-capable (See Section 3.5, ‘‘Deployment Issues’’).

PEP-enhanced HTTP message
: Any HTTP message that conforms to the this specification, *viz.* use of `Protocol:`, `Protocol-Request:` or extended `Content-Encoding:`.

protocol
: A convention for communication between two or more parties relating the syntax, sequence and semantics of the communication between them.

protocol instance
: A bag whose name is a protocol identifier, and whose items specify strength and scope values, associated header field names, and protocol-specific attributes. A protocol instance occurs in a "`Protocol:`" header of a message.

protocol identifier

an absolute URI [3,4] for a protocol specification. See Section 3.5.2, ‘‘Protocol Specification Content’’.

protocol request
A bag whose name is a protocol identifier, and whose items specify requested strength and scope values, associated header field names, and protocol-specific attribute constraints. A protocol request occurs in a "`Protocol-Request:`" header of a message.

protocol specification
A resource that describes a protocol, i.e. it defines messages, associated semantics, and possible compatibility with other specifications.

refused
("`ref`") A strength value indicating that a reply or subsequent request message must not be created in accordance with this protocol request.

required
("`req`") A strength value indicating that the associated protocol is required. As an attribute of an instance: an agent must not elide corresponding processing. As an attribute of an request: a reply or subsequent request message must be created in accordance with this protocol or an equivalent.

route
("`route`") The scope which addresses every HTTP agent in a transaction from the current agent to the origin.

scope
("`scope`") In an HTTP transaction, the set of HTTP agents being addressed: the next hop ("`conn`"), the subsequent chain ("`route`"), or the endpoints ("`origin`").

strength
("`str`") As an attribute of an protocol instance: whether an agent may ("`opt`") or may not ("`req`") ignore the protocol instance. As an attribute of a protocol request: whether a reply or subsequent request is required ("`req`"), prohibited ("`ref`"), or allowed ("`opt`") to include a compatible protocol instance.

strip
To strip a protocol instance or request, an agent must remove the bag describing it, and each of the header fields listed in that bag’s `header` attribute.

3. Operation

This section is an operational guide to PEP semantics. Section 4, ‘‘Notation’’, includes a formal presentation of the syntax and status codes.

When a PEP-capable HTTP agent receives a PEP-enhanced message, it will parse the various headers, store negotiation data away for later use, and decide which protocols to ‘‘invoke’’ and, if relaying the messsage, what data to strip from the message. This section covers each of these phases in detail.

3.1 Scenarios

There are a few different modes of operation. Here is a quick example of two parties attempting to use the `http://pep.w3.org/trackIt` protocol --

The client can demand that the server initiate use of `trackIt`:

1. The client issues a request: "`Protocol-Request: {http://pep.w3.org/trackIt {str req}}`"
   1. The server can reply using `trackIt`: "`Protocol: {http://pep.w3.org/trackIt {str req}}`"
   2. The server can deny the request: "`521 Protocol Extension Not Implemented`"

The client can unilaterally assume the server supports `trackIt`:

1. The client issues a request using `trackIt`: "`Protocol: {http://pep.w3.org/trackIt {str req}}`"
   1. The server can continue the `trackIt`-enchanced conversation: "`Protocol: {http://pep.w3.org/trackIt {str req}}`"
   2. The server can reject `trackIt`: "`422 Protocol Extension Refused`"
   3. The server may not know `trackIt` at all: "`421 Protocol Extension Unknown`"

3.2 Interpretation

First, parse each of the three PEP headers: `Protocol:`, `Protocol-Request:`, and `Content-Encoding:`. General parsing problems at this point raise error 420, `Bad Protocol Extension Request`, or its equivalent for a client.

For each protocol instance and request, the response will depend on the strength and scope. Second, select only those which are ‘‘in scope’’ (for the origin, everything; otherwise only "`conn`" and "`route`").

Third, for each protocol instance and request, the agent must return the error codes below if its strength is "`req`"; error reporting is discretionary for "`opt`" or "`ref`".

- If it refers to a *protocol identifier* whose specification or implementation is not available, Error 421, `Protocol Extension Unknown`.
- If it refers to a *protocol identifier* and *parameters* combination which is refused, Error 422, `Protocol Extension Refused`.
- If the *parameters* are not acceptable (e.g. not parseable, out-of-bounds, not acceptable by local implementation), Error 423, `Bad Protocol Extension Parameters`.
- If it has "`conn`" or "`route`" *scope* and was sent by an HTTP agent below version 1.1, Error 520, `Protocol Extension Error`. (see Section 3.5, ‘‘Deployment Issues’’).
- If it refers to a *protocol identifier* whose implementation is not available (nor any equivalent), Error 521, `Protocol Extension Not Implemented`.
- If it refers to a *protocol identifier* and *parameters* combination that are correct, but not acceptable to the server, Error 522, `Protocol Extension Parameters Not`

`Acceptable`. This might only be determined after the agent attempts processing.

Each protocol instance that ends up in scope, required (or optional but elected by the agent), must be evaluated. If it has an "`enc`" attribute cited in `Content-Encoding:`, it must be evaluated in that order, otherwise in any order after decoding is complete. Note that some extensions may not modify the message contents, so multiple extensions could be evaluated in parallel.

3.3 Negotiation

Each of the protocol requests received in an HTTP message are binding upon certain URIs. When preparing a reply or request involving some resource R, the agent should merge its preferences against the known policy statements concerning R to choose which protocol extensions to employ.

If there is no compabtible set, a server may be forced to reply with Error 520, `Protocol Extension Error`, akin to the use of Error 406, `None Acceptable`, for content negotiation.

3.4 Relaying PEP Messages

A proxy, gateway, firewall, or other non-origin HTTP agent will relay HTTP response and reply messages. When relaying a PEP-enhanced message to another HTTP agent:

1. Select protocol instances that must be evaluated and do so (Section 3.2, ‘‘Interpretation’’)
2. Before relaying, strip all instances and requests in "`conn`" scope.
3. Before relaying, optionally strip optional instances and requests in "`route`" scope. This is suggested if the agent cannot handle the particular protocol extension or configuration.

3.5 Deployment Issues

PEP is designed to tolerate relaying through non-PEP-aware HTTP agents. There is only one PEP-compatibility issue: detecting a non-PEP-aware relay which passes a PEP message containing protocol instances or requests it would have been required to act upon.

- Report error 420 if an HTTP message before 1.1 contains protocol instances or requests which in "`conn`" or "`route`" scope that are "`req`" strength.

For experimental purposes, PEP-compatbility is equated with HTTP/1.1. A trusted PEP HTTP/1.1 <--> HTTP/1.0 proxy could deploy PEP applications to the HTTP/1.0 installed base.

3.5.1 Relationship to HTTP/1.1

HTTP/1.1 defines a number of new constructs that PEP relies upon and/or integrates with:

Wrapping

PEP can enhance wrapped messages, and PEP-enhanced messages can be wrapped. Only the outermost headers are consulted for PEP features, so wrapped PEP messages should not include connection or route scope directives.

Options

This new method returns the methods and other properties of the specified URI, or of the entire server if the URI is "`*`". A server should reply with the various `Protocol-Request:` policies it supports for the server or the resource.

Tunneling

PEP can be used to set up a tunnel (e.g. a secure channel protocol). Note that any HTTP agent acting as a tunnel in a transaction, by definition, cannot act upon PEP directives in the encapsulated traffic.

Chunked Transfer Encoding

This transfer encoding allows agents to manipulate streaming, unknown-length data. Protocol extensions adapted to streaming will operate cleanly above this layer, but some extension modules may force PEP-aware agents to buffer the entire data stream before completing processing.

3.5.2 Protocol Naming and Specification

PEP, unlike many Internet applications, does not cite a central registration authority: protocol extensions are simply URIs. This allows anyone who can control a long-lived URI to create a protocol without any further registration. There are several potentially useful properties of this scheme:

Appropriate Authority

This scheme was chosen to make the process of creating and disseminating useful protocols as straightforward as possible. It avoids the problems inherent with central registration including editorial decisions, assurance of support, trademark conflict, international naming and legal variances, and so on.

The application designer can independently choose the appropriate authority for each extension. It is expected that Web developers, industry consortia, and possibly IANA will each become authorities for various extensions. Note that the compatibility clause in Section 2.1.3, ‘‘Negotiation’’, makes cross-endorsement of compatible extensions possible.

Appropriate Lifetime

Protocols, once specified, should be forward-compatible. A protocol can be extended (e.g. new attributes), but designers should not redefine existing syntax or semantics. If incompatible additions or changes to a particular protocol are needed, a new protocol should be created. Protocols can easily use naming conventions that identify revisions. For example, the URI of a protocol can include the version number of the protocol, such as `http://pep.w3.org/trackIT/v1.0` or use a `{ver 1.0}` parameter.

Protocols should be long-lasting. This means that the URI of a protocol identifier should be reliably resolvable for the foreseeable future. Designers should consider the longetivity of the server and domain when specifying a URI; it will be embedded in PEP software modules and compatibility expressions. One technique is to create a unique host name for all protocol identifiers within an organization, such as `http://pep.w3.org/`.

Note that this does *not* mean that particular extensions should be supported indefinitely. The role of PEP is to catalyze just such experimentation and ‘‘creative destruction.’’ Nevertheless, to make PEP-enhanced HTTP messages meaningful, even in the future, protocol definitions should be immutable and available.

Packages

Future clients and servers that support many protocol extensions could package their default settings into a single protocol. ‘Packages’ are protocols that consist only of a list of protocol requests. Thus, `http://pep.w3.org/LMB9.0b7` would be shorthand for the list of extensions supported by W3C’s Line Mode Browser, version 9.0 beta 7.

Similarly, some large-scale PEP applications, such as W3C’s Security Extension Architecture [14], will declare "top-level" protocols, so agents can simply request `http://pep.w3.org/SEA` to indicate an understanding of the entire SEA hierarchy.

Protocol Specification Content

A protocol identifier is a URI that resolves into a human-readable text description of the extension, ideally a specification. If the scheme is HTTP, then the media types of "`text/plain`" or "`text/html`" must be human-readable. Any other media type can be used for machine-readable purposes.

For example, the description of a ‘package’ is nothing more than a series of headers, so its readable representation would be "`message/rfc822`". Downloadable code could be made available in several formats. A syntax could also be developed to encode metainformation such as compatibility assertions.

4. Notation

PEP-related syntax is specified here relative to the definitions and rules of the HTTP/1.0 [5], HTTP/1.1 [6], and the relative URL specification [11].

4.1 Header Fields

PEP defines two new general header fields, `Protocol:` and `Protocol-Request:`, and extends the meaning of a third, `Content-Encoding:`.

```
/* Added to General Header rule, Sec 4.3 of HTTP/1.1 */
Protocol         = "Protocol" ":" 1#bag
Protocol-Request = "Protocol-Request" ":" 1#bag

/* Following rules are copied from Section 2.2 of HTTP/1.1 */
bag              = "{" bagname 1*LWS *bagitem "}"
bagname          = token ¦ URI
bagitem          = bag ¦ token ¦ quoted-string

word             = token ¦ quoted-string
token            = 1*<any CHAR except CTLs or tspecials>
tspecials        = "(" ¦ ")" ¦ "<" ¦ ">" ¦ "@"
                 ¦ "," ¦ ";" ¦ ":" ¦ "\" ¦ <">
                 ¦ "/" ¦ "[" ¦ "]" ¦ "?" ¦ "="
                 ¦ "{" ¦ "}" ¦ SP  ¦ HT

quoted-string    = ( <"> *(qdtext) <"> )
qdtext           = <any CHAR except <"> and CTLs but including LWS>
```

Each instance or request bag has several top-level attributes defined:

`{`**`<protocol-identifier>`**` {`**`scope`**` (origin ¦ conn ¦ route)} {`**`str`**` (opt ¦ req ¦ ref)} {`**`enc`**` <token>} {`**`headers`**` *<token>} {`**`params ...`**`} {`**`for`**` *<uri>} }`

protocol-identifier
: The well-known protocol extension URI.

scope
: *scope* is one of "`conn`", "`route`", or "`origin`" ; the default is "`origin`".

str
: *strength* is one of "`req`", "`ref`" or "`opt`" ; the default is "`opt`". ["`ref`" only allowed on requests]

enc
: *encoding* is a unique `token` ; the default is undefined [only allowed on instances]

headers
: A list of associated *headers*; the default is the empty set.

params
: A list of *parameters* configuring the instance or request, according to the protocol specification; the default is an empty list.

for
: a list of relative or absolute URI prefixes which a protocol request is binding upon; the default, only when the *for* list is empty, is the request URI or `Location:` response header.

Note: Multiple occurrences of PEP-defined attribute names in a list ("`{enc foo}...{enc bar}`") may yield undefined behavior.

4.2 Content Codings

The only tokens describing content codings in HTTP/1.1 are "`gzip`" and "`compress`". Other content-coding tokens may be selected from the "`enc`" attribute of protocol instances. See Sections 10.10, 3.5, and Appendix C.3, ‘‘Introduction of Content-Encoding’’, from the HTTP/1.0 specification [5] for more details.

4.3 Status Codes

PEP defines several new status codes for HTTP replies. Note that the HTTP/1.0 specification [5] states in Section 6.1.1:

> The first digit of the `Status-Code` defines the class of response. The last two digits do not have any categorization role.

Informally, PEP distinguishes PEP-dependent `x2z` response codes.

200 Class
: `220 Uses Protocol Extensions`

400 Class
: `420 Bad Protocol Extension Request`
: `421 Protocol Extension Unknown`

    422 Protocol Extension Refused
    423 Bad Protocol Extension Parameters

500 Class

    520 Protocol Extension Error
    521 Protocol Extension Not Implemented
    522 Protocol Extension Parameters Not Acceptable

Each of 400 and 500 class responses may include entity bodies with an explanation of the error, and an indication of whether the problem is temporary or permanent. Though it is correct to use a 220 response if any PEP feature is used in a reply, it should be used only when a PEP feature is required for correct handling -- it affects the cacheability of the response (Appendix A.3, ‘‘Operating as a Proxy’’).

5. Usage Examples

This section presents a high-level overview of some possible PEP applications.

5.1 Strength & Scope

These are reasonable examples for each of the (strength, scope) combinations:

PICS Labels

(`origin`, `opt`) The only parties that have to agree to transmit PICS labels [16] are the endpoints, and compliance is optional.

rot-13

(`origin`, `req`) The only parties that have to agree to transmit `rot-13` data labels are the endpoints, but compliance is required.

keep-alive

(`conn, req ¦ opt`) The only parties that have to agree to keep alive an HTTP connection are the immediate two hosts, but compliance may be required. [12]

SSL tunnel

(`route`, `req`) To set up a secure channel, every agent on the path is required to cooperate.

5.2 Payments Negotiation

Consider an electronic payments scenario where a merchant wishes to agree with a customer upon a common set of payment instruments, [9, 10]. The payment system vendors agree to a common extension, called `http://pep.w3.org/Payment`, and agree to describe each of their HTTP extensions (client wallets, merchant cash registers, proxy brokers, etc) using some commonly defined `params`, such as `cost`, `acct`, etc.

This example uses currently fictitious domain names `MyLocalBank.com`, `CreditCentral.com`, and `CashMachine.com` as examples of how different financial institutions can participate in a single electronic payment framework.

At any point, a wallet or cash register can emit a protocol request, in several currencies and account types. It can also refuse an option, particularly useful as a feedback mechanism. Here, the client specifically offers to use MyLocalBank’s `DebitCard2.1` instrument, but also asks the merchant to list other available options:

```
GET /Catalog HTTP/1.1
Protocol-Request: {http://pep.w3.org/Payment}, \
   {http://MyLocalBank.com/DebitCard2.1 {params {acct Checking}}}

220 Uses Protocol Extensions
Content-Type: text/html
Protocol-Request: {http://www.CreditCentral.com/TypeF {for /}},\
 {http://pep.CashMachine.com/e$ {params {cost {USD} {JPY}}} {for /}},\
 {http://MyLocalBank.com/DebitCard2.1 {str ref} {for /}},
```

The merchant is stating, ‘‘Here’s the catalog, but we refuse MyBank’s `DebitCard`s (including the `Checking` account type); We will accept `TypeF` CreditCentral transactions or CashMachines’s `e$` in US dollars or Japanese yen -- and this applies everywhere on our site’’.

Later, the merchant might show the contents of the shopping cart to the client with a ‘‘Pay’’ button. The cart page will provide very specific parameters about the purchase for each system. At this point, the merchant wants to force payment info to be sent. In particular, when the user access the URI behind the ‘‘Pay’’ button, `/CheckoutRegister`, the merchant wants to ask the client for an application of a payment protocol rather than another protocol request:

```
GET /MyCart HTTP/1.1

220 Uses Protocol Extensions
Content-Type: text/html
Protocol-Request:
    {http://pep.w3.org/Payment {str req} {for /CheckoutRegister}}, \
    {http://www.CreditCentral.com/TypeF \
            {params {cost {CaD 15.07} {USD 11.42}} {merchant 69}}
            {for /CheckoutRegister}},\
    {http://pep.CashMachine.com/e$ \
            {params {cost {USD 11.00} {JPY 1200.}}} \
            {for /CheckoutRegister}}
```

By *requiring* a response conformant with `http://pep.w3.org/Payment`, the client must generate instances of its deisred instruments on its subsequent request to the ‘‘Pay’’ URI (but not, say for the ‘‘Cancel’’ button or any other link).

```
POST /CheckoutRegister HTTP/1.1
Protocol: {http://pep.CashMachine.com/e$ \
    {params {cost {USD 11.00}} {acct 73} {persona "Anonymous"}}}

200 OK
Content-Type: application/x-e$
...
```

The client then chooses to reply using a protocol it believes is compatible with the generic `Payment` protocol. It applies an instance of `e$`, binding parameters for its particular fields; the server replies with MIME content that launches the helper app which drives the `e$` system.

This is an elementary example of this system; further interactions and negotiation would highlight the flexibility and extensibility of PEP.

5.3 Security Processing

PEP is not just for applications which piggyback new headers on HTTP. It is also intended to signal new encodings and pipelined reprocessing of message contents. HTTP security, for example, uses such notions [19, 14]. Consider the following request-response pair:

```
  WRAPPED * HTTP/1.1
  Protocol: {http://pep.w3.org/SEA/Signature/RSA-MD5 \
     {params {key-id RSA://khare@w3.org/hexhexhex} {sig hexhexhex}}}
  Protocol-Request: {http://pep.w3.org/SEA/Key-Exchange {str req}}
  Protocol-Request: {http://pep.w3.org/SEA/Encryption {str req}}

+ GET /MyProfile HTTP/1.1
+ ...

  220 Uses Protocol Extensions
  Protocol: {http://pep.w3.org/SEA/Signature/RSA-MD5 \
     {params {key-id RSA://server@foo.org/hexhexhex} {sig hexhexhex}}}
  Protocol: {http://pep.w3.org/SEA/Key-Exchange/RSA \
     {params {key-id RSA://khare@w3.org/hexhexhex} \
             {session-key hexhexhex} {enc key-exch3}} {str req}}
  Protocol: {http://pep.w3.org/SEA/Encryption/IDEA \
     {params {message-key "key-exch3"} {enc enc7}} {str req}}
  Content-Type: text/html
  Content-Encoding: key-exch3, enc7, gzip

+ encrypted(compressed(HTML)
```

The client is stating ‘‘I am making a signed request for my account data; and I require the server to encrypt the data (and tell me the key) if it chooses to reply.’’ The server, at its discretion, verifies the signature, unwraps the request, and returns, a signed, exchanged, encrypted, compressed resource back. Like the payment example, this is only the tip of the iceberg; there can be negotiation over which cryptosystems are available, in what strength, and with which identities.

In particular, the server’s reply indicates that first, the recipient is required to evaluate the RSA key exchange module, which generates the "`key-exch3`" message key. The next stage of the pipeline is IDEA decryption, followed by Gzip decompression. At this point, any protocols not cited in `Content-Encoding:` can be executed; in this case, verifying the server’s signature.

In security applications, it is essential that authors be able to make assertions such as ‘‘If you fetch your account records from Foo’s server, be sure to demand a signature.’’ Since that policy corresponds to a PEP `Protocol-Request:` header, it can be directly embedded in to HTML as just one more HTTP metainformation header. Such integration would also allow more specific binding of policies to objects within documents; the example in Section 5.2, ''Payment Negotiation'' might specify the `Payment` requirement along with the ''Pay'' button itself, rather than in the HTTP response to `/MyCart`. See [14] for more details on PEP integration with HTML.

6. Security Considerations

There are several security issues PEP implementors must be aware of, especially when deploying security protocol extensions. Fundamentally, PEP emphasizes flexibility, which is at odds with principles of secure design. See [14] for further analysis of PEP-based security solutions.

Separately, PEP encourages a plug-in software architecture for HTTP agents. There are manifold risks to executing untrusted or marginally-trusted code, especially if sensitive data is passed into such modules. These are not PEP-specific risks, but are of importance to any implementor.

6.1 Information Leakage

Typically, a PEP-compliant implementation will read `Content-Encoding:` and create a processing pipeline for each module.

> ''The information passing between processing stages should be considered sensitive.''

For example, one module may compute a shared session key, and pass it inband (in the clear) to the next stage, an encipherment protocol that protects the key.

Implementations should carefully protect intermediate data flows. Consider: controlling access to pipe endpoints, avoiding writing to disk, and wiping clean all memory buffers after use.

Implementors should be particularly careful on platforms that do not provide secure interprocess communication.

Finally, protocol-extension designers may wish to specify that implementations should handle several protocol instances with a single module; in the example above, a single module that generated the session key and the ciphertext.

6.2 Trusting PEP Headers

HTTP messages travel in the clear; messages using PEP are no different.

> ''Do not trust the integrity of PEP headers without proof. ''

Unless PEP headers are being read from an authenticated channel or a wrapped, signed or encrypted message, PEP headers are unreliable. An attacker can modify, remove, or add protocol requests, instances, and encoding order.

The consequences of such a man-in-the-middle attack include denial-of-service, since two parties that actually have facilities to communicate can each end up believing that the other does not.

In general, the risks here can be limited if implementors apply reasonable sanity checks: e.g. don't send sensitive data in the clear, scrutinize the order and plausibility of the modules to be run, and so on.

6.3 Protocol Interaction Effects

Protocol extension designers must be very careful about interactions with other protocols.

> ''Protocol extensions considered safe individually can be dangerous in combination or reordered.''

For example, a digital-signature protocol and an encryption protocol are both separately correct operations to execute; but it is a well-known cryptographic protocol design error [1,2] to allow signature *after* encryption. PEP, as specified in this document, does not include a mechanism to express explicit ordering constraints. The SEA security architecture provides its own guidance to implementors separately, by defining ''recipes'' of safe, allowable combinations.

Separate protocols may also interpret data in conflicting ways, or offer contradictory modes of operation. Repeated application may also be an error, e.g. `rot-13(rot-13(text))`.

6.4 Protocol Substitution Effects

The language of the PEP specification is carefully formulated to allow agents to reply using protocol extensions that an agent believes to be equivalent.

> ''Any protocol request may be satisfied using a different protocol the originator believes to be interchangeable.''

Implementors can use this technique to deploy new technology, or to make generic requests (''I require a `Signature` on the response''...''OK, it has been signed with the `FooBar` algorithm''). There may be security risks in trusting the counterparty's beliefs: the other agent might believe that cleartext is no different from ciphertext, for example.

6.5 Negotiation Scope

HTTP is a stateless protocol, and PEP does not modify that. As a result, agents cannot rely on negotiating within a fixed context; each request may be considered anew.

> ''PEP for HTTP/1.1 cannot require all agents to maintain common beliefs about capabilities.''

In particular, there is no way for servers to enforce ''refused'' semantics on clients. The client may state its preferences (including what it refuses) and the server must reply according to those preferences, since the server is ''fully informed'' at the time the request is received. The reverse is not true: when the server replies, along with its preferences, the client is not obligated to ''remember'' this information.

For example, if a server refuses clear text POST to a certain URI, there is no basis to assume that a client will not, in fact, attempt to POST clear text.

There is a similar risk associated with the for list, which allows to make protocol requests for resources matched in a different administrative domain, or unanticipated matches. The solution is the same: any false claims are refuted by the policy statement on the actual resource when it is accessed.

7. Development Path

The W3 Consortium is actively pursuing PEP specification and deployment. Reference implementations will be freely available from W3C, as will protocol extension modules for a wide variety of applications, including PICS [16]. W3C's Security and Payments Working Groups have been involved with PEP since July 1995, and are developing PEP-based solutions.

Please contact `khare@w3.org` with any questions, comments, or concerns in this area.

8. Acknowledgements

This specification makes heavy use of the grammar, constructs, and style of HTTP/1.0 and HTTP/1.1. Thanks to Roy T. Fielding for his work on those documents, and for his input to PEP.

After the first draft was released, several people came forward with excellent suggestions, but Paul Hoffman of the Internet Mail Consortium in particular put considerable time and effort into helping refine the PEP model and clean up the text of this document.

The W3 Consortium technical staff at MIT have put significant design effort into this proposal: Tim Berners-Lee, Dan Connolly, Jim Gettys, Phillip Hallam-Baker, Jim Miller, Henrik Frystyk Nielsen, and Dave Raggett.

Allan Schiffman of Terisa Systems helped clarify the logic and power of PEP security. Finally, credit is due to Dave Kristol, whose original ''A Proposed Extension Mechanism for HTTP'' Internet Draft directly led to the PEP proposal presented here.

9. References

[1]

M. Abadi and R. Needham. "Prudent Engineering Practice for Cryptographic Protocols." Digital Systems Research Center: Report 125, Digital, June 1994.

[2]

R. Anderson and R. Needham. "Robustness principles for public key protocols." ftp://ftp.cl.cam.ac.uk/users/rja14/robustness.ps.Z [in proceedings of Crypto '95], Cambridge University Computer Laboratory, 1995.

[3]

Berners-Lee, T., "Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", RFC 1630, CERN, June 1994.

[4]

Berners-Lee, T., Masinter, L., and M. McCahill, Editors, "Uniform Resource Locators (URL)", RFC 1738, CERN, Xerox Corporation, University of Minnesota, December 1994.

[5]

T. Berners-Lee, R. Fielding, and H. Frystyk Nielsen, "Hypertext Transfer Protocol

-- HTTP/1.0". Internet Draft W3 Consortium/MIT, UC Irvine, W3 Consortium/MIT, October 1995 (Work in Progress).

[6]

T. Berners-Lee, R. Fielding, and H. Frystyk Nielsen, "Hypertext Transfer Protocol -- HTTP/1.1". Internet Draft W3 Consortium/MIT, UC Irvine, W3 Consortium/MIT, November 1995 (Work in Progress).

[7]

D. H. Crocker. "Standard for the Format of ARPA Internet Text Messages." STD 11, RFC 822, UDEL, August 1982.

[8]

S. Deering, R. Hinden, Editors, "Internet Protocol, Version 6 (IPv6) Specification", Internet Draft, June 1995 (Work in Progress).

[9]

D. Eastlake, B. Boesch "Universal Payment Preamble", Internet Draft CyberCash, 6 November 1995 (Work in Progress).

[10]

D. Eastlake, "Application Level Internet Payment Syntax", Internet Draft CyberCash, 29 January 1996 (Work in Progress).

[11]

R. Fielding. "Relative Uniform Resource Locators." RFC 1808 , UC Irvine, June 1995.

[12]

A. Hopmann, "HTTP Session Extension", Internet Draft, July 1995 (Work in Progress).

[13]

D. Kristol, "A Proposed Extension Mechanism for HTTP", Internet Draft, January 1995 (Work in Progress, Expired).

[14]

R. Khare. "SEA: A Security Extension Architecture for HTTP/1.x." W3C Working Draft , W3 Consortium, February 1996 (Work In Progress).

[15]

J. Klensin, N. Freed, M. Rose, E. Stefferud, and D. Crocker. "SMTP Service Extensions." RFC 1869. MCI, Innosoft, Dover Beach Consulting, Network Management Associates, Brandenburg Consulting, November 1995.

[16]

J. Miller. "Label Syntax and Communication Protocols." Internet Draft , W3 Consortium/PICS, November 1995 (Work In Progress).

[17]

J. Postel. "Simple Mail Transfer Protocol." STD 10, USC/ISI, August 1982.

[18]

J. Postel, J. Reynolds, "Telnet Protocol specification." STD 8, RFC 854, USC/ISI, May 1983

[19]

E. Rescorla, and A. Schiffman, "The Secure Hypertext Transfer Protocol", Internet Draft, July 1995 (Work in Progress).

10. Author's Address

Rohit Khare
Technical Staff, W3 Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, U.S.A.
Tel: +1 (617) 253 5884
Fax: +1 (617) 258 8682
Email: `khare@w3.org`
Web: http://www.w3.org/People/Khare

Appendices

These appendices are provided for informational purposes only -- they do not form a part of the PEP specification.

Appendix A: Minimum Compliance

This section is a complete description of the behavior expected of PEP-compliant HTTP/1.1 agents. Essentially, a minimally compliant agent without any extensions in place need only generate the appropriate errors.

There is one subtle detail to mention with respect to all three contexts: handling PEP messages relayed through non-PEP-compliant agents (Section 3.5, ‘‘Deployment Issues’’). The rules below handle this case, but systems that actually implement one or more PEP extensions will have to detect separately when a requirement was actually directed at a previous agent, not the current agent.

It should be noted that parsing the *<bag>* data structure is a simple affair (less than 200 lines of C, for example). Error reporting is not much more difficult.

A.1 Operating as a Server

Parse PEP Headers. Parse the bags in the `Protocol:` and `Request-Protocol:` headers. The only fields that need to be kept are the *protocol identifier*, *scope*, and *strength*, and whether the bag represents an *instance* or *request*. Return Error 420 if there are any difficulties at this stage.

Return Error 521. For each *request* or *instance*, if the *strength* is `req`, return an Error 521. Explanatory text could be generated for each error raised.

Proceed. If there are no required *instances*s, the message can be understood without further processing -- simply ignore unknown encoding tokens in `Content-Encoding:`.

A.2 Operating as a Client

Parse PEP Headers. Parse the bags in the `Protocol:` header; The only fields that need to be kept are the *protocol identifier*, *scope*, *strength*, and *encoding*. Raise a parsing error if there are any difficulties at this stage.

Report Unknown Extensions. For each *instance*, if the *strength* is `req`, notify the user that the message cannot be handled, or discard the message. Explanatory text could be generated for each error raised. Additionally, it is useful to flag required *requests*, since the inability to satisfy such demands could cause the failure of subsequent request messages.

Proceed. If there are no required *instance*s, the message can be understood without further processing -- simply ignore unknown encoding tokens in `Content-Encoding:`.

A.3 Operating as a Proxy

A proxy, of course, operates in both roles: as a client to the next server, and a server to the original requestor. On forwarding either a request or a reply message:

1. **Parse PEP Headers**. Parse the bags in the `Protocol:` and `Request-Protocol:` headers. The only fields that need to be kept are the *protocol identifier*, *scope*, and *strength*, and whether the bag represents an *instance* or *request*. Return Error 420 to the client if there are any difficulties at this stage -- there is no way to return an error upstream to the server if a reply message could not be parsed.
2. **Return Error 521**. For each *request* or *instance*, if the *scope* is `conn` or `route` **and** the *strength* is `req`, return an Error 521 to the client. Explanatory text could be generated for each error raised, as well as indicating if the fault was generated by the proxy or a later server.
3. **Strip `conn`**. For each *request* or *instance*, if the *scope* is `conn`, strip that bag and any associated headers.
4. **Proceed.** If there are no required *instance*s in scope, the message can be relayed without further processing.

In addition, a caching proxy should be aware of the following policies:

- *Don't cache 220 responses*. Documents retreived that use PEP should be presumed noncacheable unless the `Cache-control:` information says otherwise. For example, a merchant per Section 5.2, ''Payments Negotiation'', may want the `/Catalog` page cached along with its default set of payment instruments, but not the later, specific-cost pages. This is a conservative policy recommendation; see the discussion in Section 4.3, ''Status Codes''.
- *Don't answer PEP-enhanced requests from the cache*. If a request uses a PEP enhancement clearly addressed to the origin or other servers (i.e. any *scope* other than `conn`), it is likely incorrect to return an answer from the cache.

Appendix B: Implementation Model

One nice property of PEP is that an implementor's effort in building PEP compatibility will be directly proportional to the complexity of the PEP application. For a bare HTTP/1.1 agent, only errors need be reported. An agent that only supports a fixed set can recognize those few, generate its own requests, and raise errors for the rest. Only the most dynamic, plug-and-play, recipe-checking agents need to build complex PEP implementations that build and check pipelines, maintain negotiation contexts, and eventual PEP HTML integration.

This appendix describes a formal model of how an HTTP agent works with PEP messages, in any of the roles just described.

> An agent `A` has a set `L` of protocol modules: `L.encoders` and `L.decoders`. Each module `l` registers the protocol extensions `e` it implements, yielding a set `E` of protocol extensions that can be generated (`E.encoders`) and accepted (`E.decoders`). For any extension, `e`, `E.compatible(e)` is a set of extensions registered as compatible with `e`.
>
> An agent `A` has a set `R` of resources it knows of. A resource has a set of URIs (`r.uri`), an entity body (`r.entity`), and a *policy* set (`r.policy`), in addition to the usual HTTP message attributes. A policy is a list of `Protocol-Request:` lines that `A` believes must be satisfied to access or transmit `r`.
>
> Any message `m` between two agents includes the resource (`m.r`), an on-the-wire entity body (`m.entity`), a list of extensions applied (`m.extensions`), an order (`m.encoding`), and a policy (`m.policy`), which can include statements about other resources. Each extension or policy entry has `strength`, `scope`, etc attributes.
>
> An agent stores policy statements into a database queryable by resource name matching (`policy.for(r.name)`)
>
> An agent `A` functions in a fixed scope, `A.scope`. For an origin client or server, it is `{conn, route, origin}`; for a proxy, it is only `{conn, route}`.

A PEP-enhanced message `m` can be received correctly with the following steps:

1. **Check the policy**. The message must be acceptable according to `A`'s policy. The magic predicate `isAcceptable(E, m.r.policy, m.extensions, m.encoding)` checks whether the order and configuration of the extensions applied to `m` satisfy `A`'s policy for the resource `m` is addressed to.
2. **Run the pipeline**. For each encoding `g` in `m.encoding`, execute some `e'.decoder` from `E.decoders.compatible(m.extensions.enc = g)` if `e'.scope = A.scope` and `e'.strength = req`.
3. **Run the rest**. For each extension `e'` where `e'.encoding` is undefined or not in `m.encoding`, execute some `e'.decoder` from `E.decoders.compatible(m.extensions.enc = g)` if `e'.scope = A.scope` and `e'.strength = req`.
4. **Store the policy**. Add each policy to the database, expanding out relative URI prefixes.

After this, `A` generates a reply. Either the user clicks on a link from that resource (client), a response is generated (server), or a reply is relayed back (proxy). The original HTTP response `m` for context `m.c` is modified into `m'` according to:

1. **Advertise the policy**. The message should express the resource's policy:

`m′.policy = m.r.policy`.

2. **Extend the message**. The message must be extended to satisfy the resource's *and* the context policy (i.e. the originator). There is a magic predicate `extendMessage(E, m.r.policy, policy.for(m.r.name))` that specifies `m′.extensions` and `m′.encoding`. `m′` must then actually be encoded according to that pipeline, etc.

In summary:

> For each `e`, the recipient may/must apply `e′.decode` for some `e′` in `e.compatible`, according to the order stated by `m.pipeline`, if applicable. Furthermore, any access to a resource must conform to `m.policy` and prefix-matching policy requests.

The software architecture for calculating `extendMessage()` and `isAcceptable()` based only on callbacks to each module is an open challenge. In particular, implementations will encounter nondeterministic interactions in negotiating solutions to `extendMessage()`.