Network Working Group
Internet-Draft
Updates: 1738 (if approved)
Obsoletes: 2732, 2396, 1808 (if approved)

Expires: December 5, 2003

T. Berners-Lee
MIT/LCS
R. Fielding
Day Software
L. Masinter
Adobe

June 6, 2003

Uniform Resource Identifier (URI): Generic Syntax
draft-fielding-uri-rfc2396bis-03

Status of this Memo

Copyright Notice

Abstract

A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource. This specification defines the generic URI syntax and a process for resolving URI references that might be in relative form, along with guidelines and security considerations for the use of URIs on the Internet.

The URI syntax defines a grammar that is a superset of all valid URIs, such that an implementation can parse the common components of a URI reference without knowing the scheme-specific requirements of every possible identifier. This specification does not define a generative grammar for URIs; that task is performed by the individual specifications of each URI scheme.

Editorial Note

Discussion of this draft and comments to the editors should be sent to the uri@w3.org mailing list. An issues list and version history is available at <http://www.apache.org/~fielding/uri/rev-2002/issues.html>.

Table of Contents

1. Introduction

A Uniform Resource Identifier (URI) provides a simple and extensible means for identifying a resource. This specification of URI syntax and semantics is derived from concepts introduced by the World Wide Web global information initiative, whose use of such identifiers dates from 1990 and is described in "Universal Resource Identifiers in WWW" [RFC1630], and is designed to meet the recommendations laid out in "Functional Recommendations for Internet Resource Locators" [RFC1736] and "Functional Requirements for Uniform Resource Names" [RFC1737].

This document obsoletes [RFC2396], which merged "Uniform Resource Locators" [RFC1738] and "Relative Uniform Resource Locators" [RFC1808] in order to define a single, generic syntax for all URIs. It excludes those portions of RFC 1738 that defined the specific syntax of individual URI schemes; those portions will be updated as separate documents. The process for registration of new URI schemes is defined separately by [RFC2717].

All significant changes from RFC 2396 are noted in Appendix D.

1.1 Overview of URIs

URIs are characterized as follows:

Uniform

Uniformity provides several benefits: it allows different types of resource identifiers to be used in the same context, even when the mechanisms used to access those resources may differ; it allows uniform semantic interpretation of common syntactic conventions across different types of resource identifiers; it allows introduction of new types of resource identifiers without interfering with the way that existing identifiers are used; and, it allows the identifiers to be reused in many different contexts, thus permitting new applications or protocols to leverage a pre-existing, large, and widely-used set of resource identifiers.

Resource

Anything that can be named or described can be a resource. Familiar examples include an electronic document, an image, a service (e.g., "today's weather report for Los Angeles"), and a collection of other resources. A resource is not necessarily accessible via the Internet; e.g., human beings, corporations, and bound books in a library can also be resources. Likewise, abstract concepts can be resources, such as the operators and operands of a

mathematical equation or the types of a relationship (e.g., "parent" or "employee").

Identifier

An identifier embodies the information required to distinguish what is being identified from all other things within its scope of identification.

A URI is an identifier that consists of a sequence of characters matching the syntax defined by the grammar rule named "URI" in Section 3. A URI can be used to refer to a resource. This specification does not place any limits on the nature of a resource or the reasons why an application might wish to refer to a resource. URIs have a global scope and should be interpreted consistently regardless of context, but that interpretation may be defined in relation to the user's context (e.g., "http://localhost/" refers to a resource that is relative to the user's network interface and yet not specific to any one user).

1.1.1 Generic Syntax

Each URI begins with a scheme name, as defined in Section 3.1, that refers to a specification for assigning identifiers within that scheme. As such, the URI syntax is a federated and extensible naming system wherein each scheme's specification may further restrict the syntax and semantics of identifiers using that scheme.

This specification defines those elements of the URI syntax that are required of all URI schemes or are common to many URI schemes. It thus defines the syntax and semantics that are needed to implement a scheme-independent parsing mechanism for URI references, such that the scheme-dependent handling of a URI can be postponed until the scheme-dependent semantics are needed. Likewise, protocols and data formats that make use of URI references can refer to this specification as defining the range of syntax allowed for all URIs, including those schemes that have yet to be defined.

A parser of the generic URI syntax is capable of parsing any URI reference into its major components; once the scheme is determined, further scheme-specific parsing can be performed on the components. In other words, the URI generic syntax is a superset of the syntax of all URI schemes.

1.1.2 Examples

The following examples illustrate URIs that are in common use.

ftp://ftp.is.co.za/rfc/rfc1808.txt
-- ftp scheme for File Transfer Protocol services

gopher://gopher.tc.umn.edu:70/11/Mailing%20Lists/
-- gopher scheme for Gopher and Gopher+ Protocol services

http://www.ietf.org/rfc/rfc2396.txt
-- http scheme for Hypertext Transfer Protocol services

```
mailto:John.Doe@example.com
   -- mailto scheme for electronic mail addresses

news:comp.infosystems.www.servers.unix
   -- news scheme for USENET news groups and articles

telnet://melvyl.ucop.edu/
   -- telnet scheme for interactive TELNET services
```

1.1.3 URI, URL, and URN

A URI can be further classified as a locator, a name, or both. The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location"). The term "Uniform Resource Name" (URN) refers to URIs under the "urn" scheme [RFC2141], which are required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable.

An individual scheme does not need to be classified as being just one of "name" or "locator". Instances of URIs from any given scheme may have the characteristics of names or locators or both, often depending on the persistence and care in the assignment of identifiers by the naming authority, rather than any quality of the scheme.

1.2 Design Considerations

1.2.1 Transcription

The URI syntax has been designed with global transcription as one of its main considerations. A URI is a sequence of characters from a very limited set: the letters of the basic Latin alphabet, digits,

and a few special characters. A URI may be represented in a variety of ways: e.g., ink on paper, pixels on a screen, or a sequence of octets in a coded character set. The interpretation of a URI depends only on the characters used and not how those characters are represented in a network protocol.

The goal of transcription can be described by a simple scenario. Imagine two colleagues, Sam and Kim, sitting in a pub at an international conference and exchanging research ideas. Sam asks Kim for a location to get more information, so Kim writes the URI for the research site on a napkin. Upon returning home, Sam takes out the napkin and types the URI into a computer, which then retrieves the information to which Kim referred.

There are several design considerations revealed by the scenario:

- A URI is a sequence of characters that is not always represented as a sequence of octets.

- A URI might be transcribed from a non-network source, and thus should consist of characters that are most likely to be able to be entered into a computer, within the constraints imposed by keyboards (and related input devices) across languages and locales.

- A URI often needs to be remembered by people, and it is easier for people to remember a URI when it consists of meaningful or familiar components.

These design considerations are not always in alignment. For example, it is often the case that the most meaningful name for a URI component would require characters that cannot be typed into some systems. The ability to transcribe a resource identifier from one medium to another has been considered more important than having a URI consist of the most meaningful of components. In local or regional contexts and with improving technology, users might benefit from being able to use a wider range of characters; such use is not defined in this specification.

1.2.2 Separating Identification from Interaction

A common misunderstanding of URIs is that they are only used to refer to accessible resources. In fact, the URI alone only provides identification; access to the resource is neither guaranteed nor implied by the presence of a URI. Instead, an operation (if any) associated with a URI reference is defined by the protocol element, data format attribute, or natural language text in which it appears.

Given a URI, a system may attempt to perform a variety of operations on the resource, as might be characterized by such words as "denote", "access", "update", "replace", or "find attributes". Such operations are defined by the protocols that make use of URIs, not by this specification. However, we do use a few general terms for describing common operations on URIs. URI "resolution" is the process of determining an access mechanism and the appropriate parameters necessary to dereference a URI; such resolution may require several iterations. Use of that access mechanism to perform an action on the URI's resource is termed a "dereference" of the URI.

When URIs are used within information systems to identify sources of information, the most common form of URI dereference is "retrieval": making use of a URI in order to retrieve a representation of its associated resource. A "representation" is a sequence of octets, along with metadata describing those octets, that constitutes a record of the state of the resource at the time that the representation is generated. Retrieval is achieved by a process that might include using the URI as a cache key to check for a locally cached representation, resolution of the URI to determine an appropriate access mechanism (if any), and dereference of the URI for the sake of applying a retrieval operation.

URI references in information systems are designed to be late-binding: the result of an access is generally determined at the time it is accessed and may vary over time or due to other aspects of the interaction. When an author creates a reference to such a resource, they do so with the intention that the reference be used in the future; what is being identified is not some specific result that was obtained in the past, but rather some characteristic that is expected to be true for future results. In such cases, the resource referred to by the URI is actually a sameness of characteristics as observed over time, perhaps elucidated by additional comments or assertions made by the resource provider.

Although many URI schemes are named after protocols, this does not imply that use of such a URI will result in access to the resource via the named protocol. URIs are often used simply for the sake of identification. Even when a URI is used to retrieve a representation of a resource, that access might be through gateways, proxies, caches, and name resolution services that are independent of the protocol associated with the scheme name, and the resolution of some URIs may require the use of more than one protocol (e.g., both DNS and HTTP are typically used to access an "http" URI's origin server when a representation isn't found in a local cache).

1.2.3 Hierarchical Identifiers

The URI syntax is organized hierarchically, with components listed in decreasing order from left to right. For some URI schemes, the visible hierarchy is limited to the scheme itself: everything after the scheme component delimiter is considered opaque to URI processing. Other URI schemes make the hierarchy explicit and visible to generic parsing algorithms.

The URI syntax reserves the slash ("/"), question-mark ("?"), and number-sign ("#") characters for the purpose of delimiting components that are significant to the generic parser's hierarchical interpretation of an identifier. In addition to aiding the readability of such identifiers through the consistent use of familiar syntax, this uniform representation of hierarchy across naming schemes allows scheme-independent references to be made relative to that hierarchy.

It is often the case that a group or "tree" of documents has been constructed to serve a common purpose; the vast majority of URIs in these documents point to resources within the tree rather than outside of it. Similarly, documents located at a particular site are much more likely to refer to other resources at that site than to resources at remote sites.

Relative referencing of URIs allows document trees to be partially independent of their location and access scheme. For instance, it is possible for a single set of hypertext documents to be simultaneously accessible and traversable via each of the "file", "http", and "ftp" schemes if the documents refer to each other using relative references. Furthermore, such document trees can be moved, as a whole, without changing any of the relative references.

A relative URI reference (Section 4.2) refers to a resource by describing the difference within a hierarchical name space between the current context and the target URI. The reference resolution algorithm, presented in Section 5, defines how such references are resolved.

1.3 Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC2234] to define the URI syntax. Although the ABNF defines syntax in terms of the US-ASCII character encoding [ASCII], the URI syntax should be interpreted in terms of the character that

the ASCII-encoded octet represents, rather than the octet encoding itself. How a URI is represented in terms of bits and bytes on the wire is dependent upon the character encoding of the protocol used to transport it, or the charset of the document that contains it.

The following core ABNF productions are used by this specification as defined by Section 6.1 of [RFC2234]: ALPHA, CR, CTL, DIGIT, DQUOTE, HEXDIG, LF, OCTET, and SP. The complete URI syntax is collected in Appendix A.

2. Characters

A URI consists of a restricted set of characters, primarily chosen to aid transcription and usability both in computer systems and in non-computer communications. Characters used conventionally as delimiters around a URI are excluded. The set of URI characters consists of digits, letters, and a few graphic symbols chosen from those common to most of the character encodings and input facilities available to Internet users.

```
   uric        = reserved / unreserved / escaped
```

Within a URI, reserved characters are used to delimit syntax components, unreserved characters are used to describe registered names, and unreserved, non-delimiting reserved, and escaped characters are used to represent strings of data (1*OCTET) within the components.

2.1 Encoding of Characters

As described above (Section 1.3), the URI syntax is defined in terms of characters by reference to the US-ASCII encoding of characters to octets. This specification does not mandate the use of any particular mapping between its character set and the octets used to store or transmit those characters.

URI characters representing strings of data within a component may, if allowed by the component production, represent an arbitrary sequence of octets. For example, portions of a given URI might

correspond to a filename on a non-ASCII file system, a query on non-ASCII data, numeric coordinates on a map, etc. Some URI schemes define a specific encoding of raw data to US-ASCII characters as part of their scheme-specific requirements. Most URI schemes represent data octets by the US-ASCII character corresponding to that octet, either directly in the form of the character's glyph or by use of an escape triplet (Section 2.4).

When a URI scheme defines a component that represents textual data consisting of characters from the Unicode (ISO 10646) character set, we recommend that the data be encoded first as octets according to the UTF-8 [UTF-8] character encoding, and then escaping only those octets that are not in the unreserved character set.

2.2 Reserved Characters

URIs include components and sub-components that are delimited by certain special characters. These characters are called "reserved", since their usage within a URI component is limited to their reserved

purpose within that component. If data for a URI component would conflict with the reserved purpose, then the conflicting data must be escaped (Section 2.4) before forming the URI.

```
   reserved    = "/" / "?" / "#" / "[" / "]" / ";" /
                 ":" / "@" / "&" / "=" / "+" / "$" / ","
```

Reserved characters are used as delimiters of the generic URI components described in Section 3, as well as within those components for delimiting sub-components. A component's ABNF syntax rule will not use the "reserved" production directly; instead, each rule lists those reserved characters that are allowed within that component. Allowed reserved characters that are not assigned a sub-component delimiter role by this specification should be considered reserved for special use by whatever software generates the URI (i.e., they may be used to delimit or indicate information that is significant to interpretation of the identifier, but that significance is outside the scope of this specification). Outside of the URI's origin, a reserved character cannot be escaped without fear of changing how it will be interpreted; likewise, an escaped octet that corresponds to a reserved character cannot be unescaped outside the software that is responsible for interpreting it during URI resolution.

The slash ("/"), question-mark ("?"), and number-sign ("#") characters are reserved in all URIs for the purpose of delimiting components that are significant to the generic parser's hierarchical interpretation of an identifier. The hierarchical prefix of a URI, wherein the slash ("/") character signifies a hierarchy delimiter, extends from the scheme (Section 3.1) through to the first question-mark ("?"), number-sign ("#"), or the end of the URI string. In other words, the slash ("/") character is not treated as a hierarchical separator within the query (Section 3.4) and fragment (Section 3.5) components of a URI, but is still considered reserved within those components for purposes outside the scope of this specification.

2.3 Unreserved Characters

Characters that are allowed in a URI but do not have a reserved purpose are called unreserved. These include uppercase and lowercase letters, decimal digits, and a limited set of punctuation marks and symbols.

```
unreserved  = ALPHA / DIGIT / mark

mark        = "-" / "_" / "." / "!" / "~" / "*" / "'" / "(" / ")"
```

Escaping unreserved characters in a URI does not change what resource

is identified by that URI. However, it may change the result of a URI comparison (Section 6), potentially leading to less efficient actions by an application. Therefore, unreserved characters should not be escaped unless the URI is being used in a context that does not allow the unescaped character to appear. URI normalization processes may unescape sequences in the ranges of ALPHA (%41-%5A and %61-%7A), DIGIT (%30-%39), hyphen (%2D), underscore (%5F), or tilde (%7E) without fear of creating a conflict, but unescaping the other mark characters is usually counterproductive.

2.4 Escaped Characters

Data must be escaped if it does not have a representation using an unreserved character; this includes data that does not correspond to a printable character of the US-ASCII coded character set or

corresponds to a US-ASCII character that delimits the component from others, is reserved in that component for delimiting sub-components, or is excluded from any use within a URI (Section 2.5).

2.4.1 Escaped Encoding

An escaped octet is encoded as a character triplet, consisting of the percent character "%" followed by the two hexadecimal digits representing that octet's numeric value. For example, "%20" is the escaped encoding for the binary octet "00100000" (ABNF: %x20), which corresponds to the US-ASCII space character (SP). This is sometimes referred to as "percent-encoding" the octet.

```
   escaped     = "%" HEXDIG HEXDIG
```

The uppercase hexadecimal digits 'A' through 'F' are equivalent to the lowercase digits 'a' through 'f', respectively. Two URIs that differ only in the case of hexadecimal digits used in escaped octets are equivalent. For consistency, we recommend that uppercase digits be used by URI generators and normalizers.

2.4.2 When to Escape and Unescape

Under normal circumstances, the only time that characters within a URI string are escaped is during the process of generating the URI from its component parts. Each component may have its own set of characters that are reserved, so only the mechanism responsible for generating or interpreting that component can determine whether or not escaping a character will change its semantics. The exception is when a URI is being used within a context where the unreserved "mark" characters might need to be escaped, such as when used for a command-line argument or within a single-quoted attribute.

Once generated, a URI is always in an escaped form. When a URI is resolved, the components significant to that scheme-specific resolution process (if any) must be parsed and separated before the escaped characters within those components can be safely unescaped.

In some cases, data that could be represented by an unreserved character may appear escaped; for example, some of the unreserved "mark" characters are automatically escaped by some systems. A URI

normalizer may unescape escaped octets that are represented by characters in the unreserved set. For example, "%7E" is sometimes used instead of tilde ("~") in an "http" URI path and can be converted to "~" without changing the interpretation of the URI.

In all cases, a URI character is equivalent to its corresponding ASCII-encoded octet, even when that octet is represented as a percent-escape. URI characters are provided as an external ASCII interface for identification between systems. A system that internally provides identifiers in the form of a different character encoding, such as EBCDIC, will generally perform character translation of textual identifiers to UTF-8 at some internal interface, thus providing meaningful identifiers in ASCII even though the back-end identifiers are in a different encoding. Escaped octets must be unescaped before such a transcoding is applied. Although this specification does not define the character encoding of escaped octets outside the ASCII range, the general principle of unescaping before transcoding should be applied for all character encodings.

Because the percent ("%") character serves as the escape indicator, it must be escaped as "%25" in order for that octet to be used as data within a URI. Implementers should be careful not to escape or unescape the same string more than once, since unescaping an already unescaped string might lead to misinterpreting a percent data character as another escaped character, or vice versa in the case of escaping an already escaped string.

2.5 Excluded Characters

Although they are disallowed within the URI syntax, we include here a description of those characters that have been excluded and the reasons for their exclusion.

```
   excluded    = invisible / delims / unwise
```

The control characters (CTL) in the US-ASCII coded character set are not used within a URI, both because they are non-printable and because they are likely to be misinterpreted by some control mechanisms. The space character (SP) is excluded because significant spaces may disappear and insignificant spaces may be introduced when

a URI is transcribed, typeset, or subjected to the treatment of

word-processing programs. Whitespace is also used to delimit a URI in many contexts. Characters outside the US-ASCII set are excluded as well.

```
   invisible   = CTL / SP / %x80-FF
```

The angle-bracket ("<" and ">") and double-quote (") characters are excluded because they are often used as the delimiters around a URI in text documents and protocol fields. The percent character ("%") is excluded because it is used for the encoding of escaped (Section 2.4) characters.

```
   delims      = "<" / ">" / "%" / DQUOTE
```

Other characters are excluded because gateways and other transport agents are known to sometimes modify such characters.

```
   unwise      = "{" / "}" / "|" / "\" / "^" / "`"
```

Data octets corresponding to excluded characters must be escaped in order to be represented within a URI.

3. Syntax Components

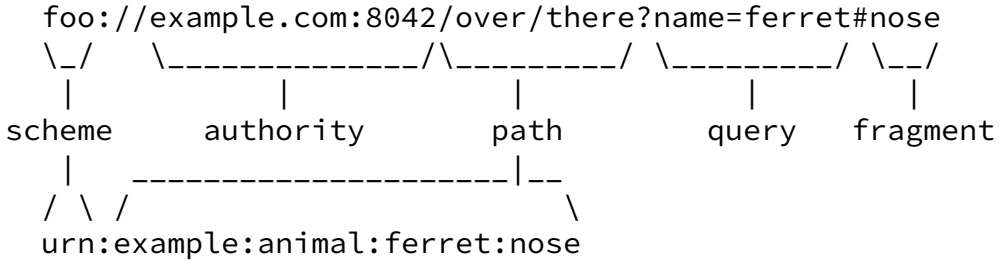
The generic URI syntax consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment.

```
URI         = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

hier-part   = net-path / abs-path / rel-path

net-path    = "//" authority [ abs-path ]
abs-path    = "/"  path-segments
rel-path    = path-segments
```

The scheme and path components are required, though path may be empty (no characters). An ABNF-driven parser of hier-part will find that the three productions in the rule are ambiguous: they are disambiguated by the "first-match-wins" (a.k.a. "greedy") algorithm. In other words, if the string begins with two slash characters ("//"), then it is a net-path; if it begins with only one slash character, then it is an abs-path; otherwise, it is a rel-path. Note that rel-path does not necessarily contain any slash ("/") characters; a non-hierarchical path will be treated as opaque data by a generic URI parser.

The authority component is only present when a string matches the net-path production. Since the presence of an authority component restricts the remaining syntax for path, we have not included a specific "path" rule in the syntax. Instead, what we refer to as the URI path is that part of the parsed URI string matching the abs-path or rel-path production in the syntax above, since they are mutually exclusive for any given URI and can be parsed as a single component.

The following are two example URIs and their component parts:



3.1 Scheme

Each URI begins with a scheme name that refers to a specification for assigning identifiers within that scheme. As such, the URI syntax is

a federated and extensible naming system wherein each scheme's specification may further restrict the syntax and semantics of identifiers using that scheme.

Scheme names consist of a sequence of characters beginning with a letter and followed by any combination of letters, digits, plus ("+"), period ("."), or hyphen ("-"). Although scheme is case-insensitive, the canonical form is lowercase and documents that specify schemes must do so using lowercase letters. An implementation should accept uppercase letters as equivalent to lowercase in scheme names (e.g., allow "HTTP" as well as "http"), for the sake of robustness, but should only generate lowercase scheme names, for consistency.

```
   scheme      = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )
```

Individual schemes are not specified by this document. The process for registration of new URI schemes is defined separately by [RFC2717]. The scheme registry maintains the mapping between scheme names and their specifications.

3.2 Authority

Many URI schemes include a hierarchical element for a naming authority, such that governance of the name space defined by the remainder of the URI is delegated to that authority (which may, in turn, delegate it further). The generic syntax provides a common means for distinguishing an authority based on a registered domain name or server address, along with optional port and user information.

The authority component is preceded by a double slash ("//") and is terminated by the next slash ("/"), question-mark ("?"), or number-sign ("#") character, or by the end of the URI.

```
   authority   = [ userinfo "@" ] host [ ":" port ]
```

The parts "<userinfo>@" and ":<port>" may be omitted.

Some schemes do not allow the userinfo and/or port sub-components. When presented with a URI that violates one or more scheme-specific restrictions, the scheme-specific URI resolution process should flag the reference as an error rather than ignore the unused parts; doing so reduces the number of equivalent URIs and helps detect abuses of the generic syntax that might indicate the URI has been constructed to mislead the user (Section 7.5).

3.2.1 User Information

The userinfo sub-component may consist of a user name and, optionally, scheme-specific information about how to gain authorization to access the server. The user information, if present, is followed by a commercial at-sign ("@") that delimits it from the host.

```
userinfo    = *( unreserved / escaped / ";" /
                 ":" / "&" / "=" / "+" / "$" / "," )
```

Some URI schemes use the format "user:password" in the userinfo field. This practice is NOT RECOMMENDED, because the passing of authentication information in clear text has proven to be a security risk in almost every case where it has been used. Note also that userinfo might be crafted to look like a trusted domain name in order to mislead users, as described in Section 7.5.

3.2.2 Host

The host sub-component of authority is identified by an IPv6 literal encapsulated within square brackets, an IPv4 address in dotted-decimal form, or a domain name.

```
host        = [ IPv6reference / IPv4address / hostname ]
```

If host is omitted, a default may be defined by the scheme-specific semantics of the URI. For example, the "file" URI scheme defaults to "localhost", whereas the "http" URI scheme does not allow host to be omitted.

The production for host is ambiguous because it does not completely

distinguish between an IPv4address and a hostname. Again, the "first-match-wins" algorithm applies: If host matches the production for IPv4address, then it should be considered an IPv4 address literal and not a hostname.

A hostname takes the form described in Section 3 of [RFC1034] and Section 2.1 of [RFC1123]: a sequence of domain labels separated by ".", each domain label starting and ending with an alphanumeric character and possibly also containing "-" characters. The rightmost domain label of a fully qualified domain name may be followed by a single "." if it is necessary to distinguish between the complete domain name and some local domain.

```
   hostname    = domainlabel qualified
   qualified   = *( "." domainlabel ) [ "." ]
   domainlabel = alphanum [ 0*61( alphanum / "-" ) alphanum ]
```



```
   alphanum    = ALPHA / DIGIT
```

A host identified by an IPv4 literal address is represented in dotted-decimal notation (a sequence of four decimal numbers in the range 0 to 255, separated by "."), as described in [RFC1123] by reference to [RFC0952]. Note that other forms of dotted notation may be interpreted on some platforms, as described in Section 7.3, but only the dotted-decimal form of four octets is allowed by this grammar.

```
   IPv4address = dec-octet "." dec-octet "." dec-octet "." dec-octet

   dec-octet   = DIGIT                 ; 0-9
               / %x31-39 DIGIT         ; 10-99
               / "1" 2DIGIT            ; 100-199
               / "2" %x30-34 DIGIT     ; 200-249
               / "25" %x30-35          ; 250-255
```

A host identified by an IPv6 literal address [RFC3513] is distinguished by enclosing the IPv6 literal within square-brackets ("[" and "]"). This is the only place where square-bracket characters are allowed in the URI syntax.

```
   IPv6reference = "[" IPv6address "]"
```

```
   IPv6address =                            6( h4 ":" ) ls32
               /                       "::" 5( h4 ":" ) ls32
               / [               h4 ] "::" 4( h4 ":" ) ls32
               / [ *1( h4 ":" ) h4 ] "::" 3( h4 ":" ) ls32
               / [ *2( h4 ":" ) h4 ] "::" 2( h4 ":" ) ls32
               / [ *3( h4 ":" ) h4 ] "::"    h4 ":"   ls32
               / [ *4( h4 ":" ) h4 ] "::"             ls32
               / [ *5( h4 ":" ) h4 ] "::"             h4
               / [ *6( h4 ":" ) h4 ] "::"

   ls32        = ( h4 ":" h4 ) / IPv4address
               ; least-significant 32 bits of address

   h4          = 1*4HEXDIG
```

The presence of host within a URI does not imply that the scheme requires access to the given host on the Internet. In many cases, the host syntax is used only for the sake of reusing the existing registration process created and deployed for DNS, thus obtaining a globally unique name without the cost of deploying another registry. However, such use comes with its own costs: domain name ownership may change over time for reasons not anticipated by the URI creator.

3.2.3 Port

The port sub-component of authority is designated by an optional port number in decimal following the host and delimited from it by a single colon (":") character.

```
   port        = *DIGIT
```

If port is omitted, a default may be defined by the scheme-specific semantics of the URI. Likewise, the type of network port designated by the port number (e.g., TCP, UDP, SCTP, etc.) is defined by the URI scheme. For example, the "http" URI scheme defines a default of TCP port 80.

3.3 Path

The path component contains hierarchical data that, along with data in the optional query (Section 3.4) component, serves to identify a

resource within the scope of that URI's scheme and naming authority (if any). There is no specific "path" syntax production in the generic URI syntax. Instead, what we refer to as the URI path is that part of the parsed URI string matching either the abs-path or the rel-path production, since they are mutually exclusive for any given URI and can be parsed as a single component. The path is terminated by the first question-mark ("?") or number-sign ("#") character, or by the end of the URI.

```
path-segments = segment *( "/" segment )
segment       = *pchar

pchar         = unreserved / escaped / ";" /
                ":" / "@" / "&" / "=" / "+" / "$" / ","
```

The path consists of a sequence of path segments separated by a slash ("/") character. A path is always defined for a URI, though the defined path may be empty (zero length) or opaque (not containing any "/" delimiters). For example, the URI <mailto:fred@example.com> has a path of "fred@example.com".

The path segments "." and ".." are defined for relative reference within the path name hierarchy. They are intended for use at the beginning of a relative path reference (Section 4.2) for indicating relative position within the hierarchical tree of names, with a similar effect to how they are used within some operating systems' file directory structure to indicate the current directory and parent directory, respectively. Unlike a file system, however, these dot-segments are only interpreted within the URI path hierarchy and are removed as part of the URI normalization or resolution process,

as described in Section 5.2.

Aside from dot-segments in hierarchical paths, a path segment is considered opaque by the generic syntax. URI generating applications often use the reserved characters allowed in segment for the purpose of delimiting scheme-specific or generator-specific sub-components. For example, the semicolon (";") and equals ("=") reserved characters are often used for delimiting parameters and parameter values applicable to that segment. The comma (",") reserved character is often used for similar purposes. For example, one URI generator might use a segment like "name;v=1.1" to indicate a reference to

version 1.1 of "name", whereas another might use a segment like "name,1.1" to indicate the same. Parameter types may be defined by scheme-specific semantics, but in most cases the meaning of a parameter is specific to the URI originator.

3.4 Query

The query component contains non-hierarchical data that, along with data in the path (Section 3.3) component, serves to identify a resource within the scope of that URI's scheme and naming authority (if any). The query component is indicated by the first question-mark ("?") character and terminated by a number-sign ("#") character or by the end of the URI.

```
query       = *( pchar / "/" / "?" )
```

The characters slash ("/") and question-mark ("?") are allowed to represent data within the query component, but such use is discouraged; incorrect implementations of reference resolution often fail to distinguish them from hierarchical separators, thus resulting in non-interoperable results while parsing relative references. However, since query components are often used to carry identifying information in the form of "key=value" pairs, and one frequently used value is a reference to another URI, it is sometimes better for usability to include those characters unescaped.

> Note: Some client applications will fail to separate a reference's query component from its path component before merging the base and reference paths (Section 5.2). This may result in loss of information if the query component contains the strings "/../" or "/./".

3.5 Fragment

The fragment identifier component allows indirect identification of a secondary resource by reference to a primary resource and additional

identifying information that is selective within that resource. The identified secondary resource may be some portion or subset of the primary resource, some view on representations of the primary resource, or some other resource that is merely named within the primary resource. A fragment identifier component is indicated by the presence of a number-sign ("#") character and terminated by the end of the URI string.

```
   fragment    = *( pchar / "/" / "?" )
```

The semantics of a fragment identifier are defined by the set of representations that might result from a retrieval action on the primary resource. The fragment's format and resolution is therefore dependent on the media type [RFC2046] of the retrieved representation, even though such a retrieval is only performed if the URI is dereferenced. Individual media types may define their own restrictions on, or structure within, the fragment identifier syntax for specifying different types of subsets, views, or external references that are identifiable as secondary resources by that media type. If the primary resource is represented by multiple media types, as is often the case for resources whose representation is selected based on attributes of the retrieval request, then interpretation of the fragment identifier must be consistent across all of those media types in order for it to be viable as an identifier.

As with any URI, use of a fragment identifier component does not imply that a retrieval action will take place. A URI with a fragment identifier may be used to refer to the secondary resource without any implication that the primary resource is accessible. However, if that URI is used in a context that does call for retrieval and is not a same-document reference (Section 4.4), the fragment identifier is only valid as a reference if a retrieval action on the primary resource succeeds and results in a representation for which the fragment identifier is meaningful.

Fragment identifiers have a special role in information systems as the primary form of client-side indirect referencing, allowing an author to specifically identify those aspects of an existing resource that are only indirectly provided by the resource owner. As such, interpretation of the fragment identifier during a retrieval action is performed solely by the user agent; the fragment identifier is not passed to other systems during the process of retrieval. Although this is often perceived to be a loss of information, particularly in

regards to accurate redirection of references as content moves over time, it also serves to prevent information providers from denying reference authors the right to selectively refer to information within a resource.

The characters slash ("/") and question-mark ("?") are allowed to represent data within the fragment identifier, but such use is discouraged for the same reasons as described above for query.

4. Usage

When applications make reference to a URI, they do not always use the full form of reference defined by the "URI" syntax production. In order to save space and take advantage of hierarchical locality, many Internet protocol elements and media type formats allow an abbreviation of a URI, while others restrict the syntax to a particular form of URI. We define the most common forms of reference syntax in this specification because they impact and depend upon the design of the generic syntax, requiring a uniform parsing algorithm in order to be interpreted consistently.

4.1 URI Reference

The ABNF rule URI-reference is used to denote the most common usage of a resource identifier.

```
URI-reference = URI / relative-URI
```

A URI-reference may be relative: if the reference string's prefix matches the syntax of a scheme followed by its colon separator, then the reference is a URI rather than a relative-URI.

A URI-reference is typically parsed first into the five URI components, in order to determine what components are present and whether or not the reference is relative, and then each component is parsed for its subparts and their validation. The ABNF of URI-reference, along with the "first-match-wins" disambiguation rule, is sufficient to define a validating parser for the generic syntax. Readers familiar with regular expressions should see Appendix B for an example of a non-validating URI-reference parser that will take any given string and extract the URI components.

4.2 Relative URI

A relative URI reference takes advantage of the hier-part syntax (Section 3) in order to express a reference that is relative to the name space of another hierarchical URI.

```
relative-URI  = hier-part [ "?" query ] [ "#" fragment ]
```

The URI referred to by a relative reference is obtained by applying the reference resolution algorithm of Section 5.

A relative reference that begins with two slash characters is termed a network-path reference; such references are rarely used. A relative reference that begins with a single slash character is termed an absolute-path reference. A relative reference that does not begin

with a slash character is termed a relative-path reference.

A path segment that contains a colon character (e.g., "this:that") cannot be used as the first segment of a relative-path reference because it would be mistaken for a scheme name. Such a segment must be preceded by a dot-segment (e.g., "./this:that") to make a relative-path reference.

4.3 Absolute URI

Some protocol elements allow only the absolute form of a URI without a fragment identifier. For example, defining the base URI for later use by relative references calls for an absolute-URI production that does not allow a fragment.

```
absolute-URI  = scheme ":" hier-part [ "?" query ]
```

4.4 Same-document Reference

When a URI reference occurring within a document or message refers to a URI that is, aside from its fragment component (if any), identical to the base URI (Section 5.1), that reference is called a "same-document" reference. The most frequent examples of same-document references are relative references that are empty or include only the number-sign ("#") separator followed by a fragment identifier.

When a same-document reference is dereferenced for the purpose of a retrieval action, the target of that reference is defined to be within that current document or message; the dereference should not result in a new retrieval.

4.5 Suffix Reference

The URI syntax is designed for unambiguous reference to resources and extensibility via the URI scheme. However, as URI identification and usage have become commonplace, traditional media (television, radio, newspapers, billboards, etc.) have increasingly used a suffix of the URI as a reference, consisting of only the authority and path portions of the URI, such as

```
www.w3.org/Addressing/
```

or simply the DNS hostname on its own. Such references are primarily intended for human interpretation rather than machine, with the assumption that context-based heuristics are sufficient to complete the URI (e.g., most hostnames beginning with "www" are likely to have

a URI prefix of "http://"). Although there is no standard set of heuristics for disambiguating a URI suffix, many client implementations allow them to be entered by the user and heuristically resolved. It should be noted that such heuristics may change over time, particularly when new URI schemes are introduced.

Since a URI suffix has the same syntax as a relative path reference, a suffix reference cannot be used in contexts where a relative reference is expected. As a result, suffix references are limited to those places where there is no defined base URI, such as dialog boxes and off-line advertisements.

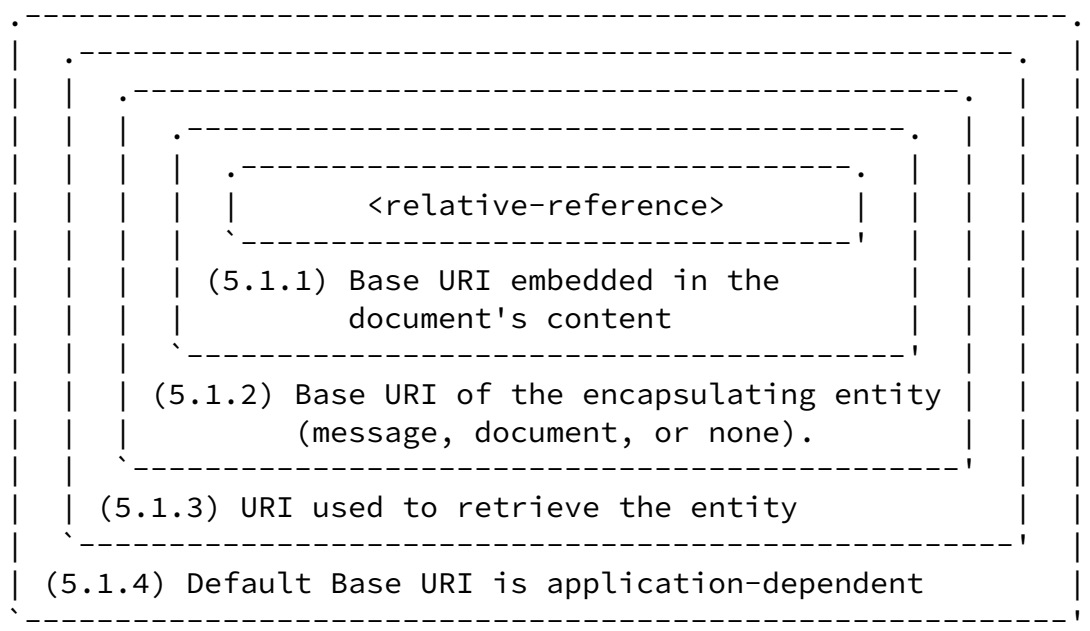
5. Reference Resolution

This section defines the process of resolving a URI reference within a context that allows relative references, such that the result is a string matching the "URI" syntax production of Section 3.

5.1 Establishing a Base URI

The term "relative" implies that there exists some "base URI" against which the relative reference is applied. Aside from same-document references (Section 4.4, relative references are only usable if the base URI is known. The base URI must be established by the parser prior to parsing URI references that might be relative.

The base URI of a document can be established in one of four ways, listed below in order of precedence. The order of precedence can be thought of in terms of layers, where the innermost defined base URI has the highest precedence. This can be visualized graphically as:



5.1.1 Base URI within Document Content

Within certain document media types, the base URI of the document can be embedded within the content itself such that it can be readily obtained by a parser. This can be useful for descriptive documents, such as tables of content, which may be transmitted to others through protocols other than their usual retrieval context (e.g., E-Mail or USENET news).

It is beyond the scope of this document to specify how, for each

media type, the base URI can be embedded. It is assumed that user agents manipulating such media types will be able to obtain the appropriate syntax from that media type's specification.

A mechanism for embedding the base URI within MIME container types (e.g., the message and multipart types) is defined by MHTML [RFC2110]. Protocols that do not use the MIME message header syntax, but do allow some form of tagged metadata to be included within messages, may define their own syntax for defining the base URI as part of a message.

5.1.2 Base URI from the Encapsulating Entity

If no base URI is embedded, the base URI of a document is defined by the document's retrieval context. For a document that is enclosed within another entity (such as a message or another document), the retrieval context is that entity; thus, the default base URI of the document is the base URI of the entity in which the document is encapsulated.

5.1.3 Base URI from the Retrieval URI

If no base URI is embedded and the document is not encapsulated within some other entity (e.g., the top level of a composite entity), then, if a URI was used to retrieve the base document, that URI shall be considered the base URI. Note that if the retrieval was the result of a redirected request, the last URI used (i.e., that which resulted in the actual retrieval of the document) is the base URI.

5.1.4 Default Base URI

If none of the conditions described in above apply, then the base URI is defined by the context of the application. Since this definition is necessarily application-dependent, failing to define the base URI using one of the other methods may result in the same content being interpreted differently by different types of application.

It is the responsibility of the distributor(s) of a document containing a relative reference to ensure that the base URI for that document can be established. It must be emphasized that a relative reference, aside from a same-document reference, cannot be used reliably in situations where the document's base URI is not well-defined.

5.2 Obtaining the Referenced URI

This section describes an example algorithm for resolving URI references that might be relative to a given base URI. The algorithm

is intended to provide a definitive result that can be used to test the output of other implementations. Implementation of the algorithm itself is not required, but the result given by an implementation must match the result that would be given by this algorithm.

The base URI (Base) is established according to the rules of Section 5.1 and parsed into the five main components described in Section 3. Note that only the scheme component is required to be present in the base URI; the other components may be empty or undefined. A component is undefined if its preceding separator does not appear in the URI reference; the path component is never undefined, though it may be empty. The algorithm assumes that the base URI is well-formed and does not contain dot-segments in its path.

For each URI reference (R), the following pseudocode describes an algorithm for transforming R into its target URI (T):

```
   -- The URI reference is parsed into the five URI components
   --
   (R.scheme, R.authority, R.path, R.query, R.fragment) = parse(R);

   -- A non-strict parser may ignore a scheme in the reference
   -- if it is identical to the base URI's scheme.
   --
   if ((not strict) and (R.scheme == Base.scheme)) then
      undefine(R.scheme);
   endif;

   if defined(R.scheme) then
      T.scheme    = R.scheme;
      T.authority = R.authority;
      T.path      = remove_dot_segments(R.path);
      T.query     = R.query;
   else
      if defined(R.authority) then
         T.authority = R.authority;
         T.path      = remove_dot_segments(R.path);
         T.query     = R.query;
      else
         if (R.path == "") then
            T.path = Base.path;
            if defined(R.query) then
               T.query = R.query;
            else
               T.query = Base.query;
            endif;
         else
            if (R.path starts-with "/") then
```

```
            T.path = remove_dot_segments(R.path);
         else
            T.path = merge(Base.path, R.path);
            T.path = remove_dot_segments(T.path);
         endif;
         T.query = R.query;
      endif;
      T.authority = Base.authority;
   endif;
   T.scheme = Base.scheme;
endif;

T.fragment = R.fragment;
```

The pseudocode above refers to a merge routine for merging a relative-path reference with the path of the base URI. This is accomplished as follows:

- If the base URI's path is empty, then return a string consisting of "/" concatenated with the reference's path component; otherwise,

- If the base URI's path is non-hierarchical, as indicated by not beginning with a slash, then return a string consisting of the reference's path component; otherwise,

- Return a string consisting of the reference's path component appended to all but the last segment of the base URI's path (i.e., any characters after the right-most "/" in the base URI path are excluded).

The pseudocode also refers to a remove_dot_segments routine for interpreting and removing the special "." and ".." complete path segments from a referenced path. This is done after the path is extracted from a reference, whether or not the path was relative, in order to remove any invalid or extraneous dot-segments prior to forming the target URI. Although there are many ways to accomplish this removal process, we describe a simple method using a separate string buffer:

1. The buffer is initialized with the unprocessed path component.

2. If the buffer begins with "./" or "../", the "." or ".." segment is removed.

3. All occurrences of "/./" in the buffer are replaced with "/".

4. If the buffer ends with "/.", the "." is removed.

5. All occurrences of "//../" in the buffer, where ".." and are complete path segments, are iteratively replaced with "/" in order from left to right until no matching pattern remains. If the buffer ends with "//..", that is also replaced with "/". Note that may be empty.

6. All prefixes of "/../" in the buffer, where ".." and are complete path segments, are iteratively replaced with "/" in order from left to right until no matching pattern remains. If the buffer ends with "/..", that is also replaced with "/". Note that may be empty.

7. The remaining buffer is returned as the result of remove_dot_segments.

Some systems may find it more efficient to implement the remove_dot_segments algorithm as a stack of path segments being compressed, rather than as a series of string pattern replacements.

5.3 Recomposition of a Parsed URI

Parsed URI components can be recomposed to obtain the corresponding URI reference string. Using pseudocode, this would be:

```
result = ""

if defined(scheme) then
   append scheme to result;
   append ":" to result;
endif;

if defined(authority) then
   append "//" to result;
   append authority to result;
endif;

append path to result;

if defined(query) then
   append "?" to result;
   append query to result;
endif;
```

```
      if defined(fragment) then
         append "#" to result;
         append fragment to result;
      endif;
```



```
      return result;
```

Note that we are careful to preserve the distinction between a component that is undefined, meaning that its separator was not present in the reference, and a component that is empty, meaning that the separator was present and was immediately followed by the next component separator or the end of the reference.

5.4 Reference Resolution Examples

Within an object with a well-defined base URI of

```
      http://a/b/c/d;p?q
```

a relative URI reference would be resolved as follows:

5.4.1 Normal Examples

```
      "g:h"           =  "g:h"
      "g"             =  "http://a/b/c/g"
      "./g"           =  "http://a/b/c/g"
      "g/"            =  "http://a/b/c/g/"
      "/g"            =  "http://a/g"
      "//g"           =  "http://g"
      "?y"            =  "http://a/b/c/d;p?y"
      "g?y"           =  "http://a/b/c/g?y"
      "#s"            =  "http://a/b/c/d;p?q#s"
      "g#s"           =  "http://a/b/c/g#s"
      "g?y#s"         =  "http://a/b/c/g?y#s"
      ";x"            =  "http://a/b/c/;x"
      "g;x"           =  "http://a/b/c/g;x"
      "g;x?y#s"       =  "http://a/b/c/g;x?y#s"
      "."             =  "http://a/b/c/"
      "./"            =  "http://a/b/c/"
      ".."            =  "http://a/b/"
```

```
      "../"           =  "http://a/b/"
      "../g"          =  "http://a/b/g"
      "../.."         =  "http://a/"
      "../../"        =  "http://a/"
      "../../g"       =  "http://a/g"
```

5.4.2 Abnormal Examples

Although the following abnormal examples are unlikely to occur in normal practice, all URI parsers should be capable of resolving them consistently. Each example uses the same base as above.

An empty reference refers to the current base URI.

```
      ""              =  "http://a/b/c/d;p?q"
```

Parsers must be careful in handling the case where there are more relative path ".." segments than there are hierarchical levels in the base URI's path. Note that the ".." syntax cannot be used to change the authority component of a URI.

```
      "../../../g"    =  "http://a/g"
      "../../../../g" =  "http://a/g"
```

Similarly, parsers must remove the dot-segments "." and ".." when they are complete components of a path, but not when they are only part of a segment.

```
      "/./g"          =  "http://a/g"
      "/../g"         =  "http://a/g"
      "g."            =  "http://a/b/c/g."
      ".g"            =  "http://a/b/c/.g"
      "g.."           =  "http://a/b/c/g.."
      "..g"           =  "http://a/b/c/..g"
```

Less likely are cases where the relative URI uses unnecessary or nonsensical forms of the "." and ".." complete path segments.

```
      "./../g"        =  "http://a/b/g"
      "./g/."         =  "http://a/b/c/g/"
```

```
   "g/./h"         =  "http://a/b/c/g/h"
   "g/../h"        =  "http://a/b/c/h"
   "g;x=1/./y"     =  "http://a/b/c/g;x=1/y"
   "g;x=1/../y"    =  "http://a/b/c/y"
```

Some applications fail to separate the reference's query and/or fragment components from a relative path before merging it with the base path and removing dot-segments. This error is rarely noticed, since typical usage of a fragment never includes the hierarchy ("/") character, and the query component is not normally used within relative references.

```
   "g?y/./x"       =  "http://a/b/c/g?y/./x"
   "g?y/../x"      =  "http://a/b/c/g?y/../x"
   "g#s/./x"       =  "http://a/b/c/g#s/./x"
   "g#s/../x"      =  "http://a/b/c/g#s/../x"
```

Some parsers allow the scheme name to be present in a relative URI if it is the same as the base URI scheme. This is considered to be a loophole in prior specifications of partial URI [RFC1630]. Its use

should be avoided, but is allowed for backward compatibility.

```
   "http:g"        =  "http:g"         ; for strict parsers
                   /  "http://a/b/c/g" ; for backward compatibility
```

6. Normalization and Comparison

One of the most common operations on URIs is simple comparison: determining if two URIs are equivalent without using the URIs to access their respective resource(s). A comparison is performed every time a response cache is accessed, a browser checks its history to color a link, or an XML parser processes tags within a namespace. Extensive normalization prior to comparison of URIs is often used by spiders and indexing engines to prune a search space or reduce duplication of request actions and response storage.

URI comparison is performed in respect to some particular purpose, and software with differing purposes will often be subject to differing design trade-offs in regards to how much effort should be

spent in reducing duplicate identifiers. This section describes a variety of methods that may be used to compare URIs, the trade-offs between them, and the types of applications that might use them.

6.1 Equivalence

Since URIs exist to identify resources, presumably they should be considered equivalent when they identify the same resource. However, such a definition of equivalence is not of much practical use, since there is no way for software to compare two resources without knowledge of their origin. For this reason, determination of equivalence or difference of URIs is based on string comparison, perhaps augmented by reference to additional rules provided by URI scheme definitions. We use the terms "different" and "equivalent" to describe the possible outcomes of such comparisons, but there are many application-dependent versions of equivalence.

Even though it is possible to determine that two URIs are equivalent, it is never possible to be sure that two URIs identify different resources. Therefore, comparison methods are designed to minimize false negatives while strictly avoiding false positives.

In testing for equivalence, it is generally unwise to directly compare relative URI references; they should be converted to their absolute forms before comparison. Furthermore, when URI references are being compared for the purpose of selecting (or avoiding) a network action, such as retrieval of a representation, it is often necessary to remove fragment identifiers from the URIs prior to comparison.

6.2 Comparison Ladder

A variety of methods are used in practice to test URI equivalence. These methods fall into a range, distinguished by the amount of

processing required and the degree to which the probability of false negatives is reduced. As noted above, false negatives cannot in principle be eliminated. In practice, their probability can be reduced, but this reduction requires more processing and is not cost-effective for all applications.

If this range of comparison practices is considered as a ladder, the

following discussion will climb the ladder, starting with those that are cheap but have a relatively higher chance of producing false negatives, and proceeding to those that have higher computational cost and lower risk of false negatives.

6.2.1 Simple String Comparison

If two URIs, considered as character strings, are identical, then it is safe to conclude that they are equivalent. This type of equivalence test has very low computational cost and is in wide use in a variety of applications, particularly in the domain of parsing.

Testing strings for equivalence requires some basic precautions. This procedure is often referred to as "bit-for-bit" or "byte-for-byte" comparison, which is potentially misleading. Testing of strings for equality is normally based on pairwise comparison of the characters that make up the strings, starting from the first and proceeding until both strings are exhausted and all characters found to be equal, a pair of characters compares unequal, or one of the strings is exhausted before the other.

Such character comparisons require that each pair of characters be put in comparable form. For example, should one URI be stored in a byte array in EBCDIC encoding, and the second be in a Java String object, bit-for-bit comparisons applied naively will produce both false-positive and false-negative errors. Thus, in principle, it is better to speak of equality on a character-for-character rather than byte-for-byte or bit-for-bit basis.

Unicode defines a character as being identified by number ("codepoint") with an associated bundle of visual and other semantics. At the software level, it is not practical to compare semantic bundles, so in practical terms, character-by-character comparisons are done codepoint-by-codepoint.

6.2.2 Syntax-based Normalization

Software may use logic based on the definitions provided by this specification to reduce the probability of false negatives. Such processing is moderately higher in cost than character-for-character string comparison. For example, an application using this approach could reasonably consider the following two URIs equivalent:

```
example://a/b/c/%7A
eXAMPLE://a/./b/../b/c/%7a
```

Web user agents, such as browsers, typically apply this type of URI normalization when determining whether a cached response is available. Syntax-based normalization includes such techniques as case normalization, escape normalization, and removal of dot-segments.

6.2.2.1 Case Normalization

When a URI scheme uses components of the generic syntax, it will also use the common syntax equivalence rules, namely that the scheme and hostname are case insensitive and therefore can be normalized to lowercase. For example, the URI <HTTP://www.EXAMPLE.com/> is equivalent to <http://www.example.com/>.

6.2.2.2 Escape Normalization

The percent-escape mechanism described in Section 2.4 is a frequent source of variance among otherwise identical URIs. One cause is the choice of uppercase or lowercase letters for the hexadecimal digits within the escape sequence (e.g., "%3a" versus "%3A"). Such sequences are always equivalent; for the sake of uniformity, URI generators and normalizers are strongly encouraged to use uppercase letters for the hex digits A-F.

Only characters that are excluded from or reserved within the URI syntax must be escaped when used as data. However, some URI generators go beyond that and escape characters that do not require escaping, resulting in URIs that are equivalent to their unescaped counterparts. Such URIs can be normalized by unescaping sequences that represent the unreserved characters, as described in Section 2.3.

6.2.2.3 Path Segment Normalization

The complete path segments "." and ".." have a special meaning within hierarchical URI schemes. As such, they should not appear in absolute paths; if they are found, they can be removed by applying

the remove_dot_segments algorithm to the path, as described in Section 5.2.

6.2.3 Scheme-based Normalization

The syntax and semantics of URIs vary from scheme to scheme, as described by the defining specification for each scheme. Software may use scheme-specific rules, at further processing cost, to reduce the probability of false negatives. For example, Web spiders that populate most large search engines would consider the following two URIs to be equivalent:

```
http://example.com/
http://example.com:80/
```

This behavior is based on the rules provided by the syntax and semantics of the "http" URI scheme, which defines an empty port component as being equivalent to the default TCP port for HTTP (port 80). In general, a URI scheme that uses the generic syntax for authority is defined such that a URI with an explicit ":port", where the port is the default for the scheme, is equivalent to one where the port is elided.

6.2.4 Protocol-based Normalization

Web spiders, for which substantial effort to reduce the incidence of false negatives is often cost-effective, are observed to implement even more aggressive techniques in URI comparison. For example, if they observe that a URI such as

```
http://example.com/data
```

redirects to a URI differing only in the trailing slash

```
http://example.com/data/
```

they will likely regard the two as equivalent in the future. Obviously, this kind of technique is only appropriate in special situations.

6.3 Canonical Form

It is in the best interests of everyone to avoid false-negatives in

comparing URIs and to minimize the amount of software processing for such comparisons. Those who generate and make reference to URIs can reduce the cost of processing and the risk of false negatives by consistently providing them in a form that is reasonably canonical with respect to their scheme. Specifically:

- Always provide the URI scheme in lowercase characters.
- Always provide the hostname, if any, in lowercase characters.
- Only perform percent-escaping where it is essential.
- Always use uppercase A-through-F characters when percent-escaping.
- Prevent /./ and /../ from appearing in non-relative URI paths.

The good practices listed above are motivated by deployed software that frequently use these techniques for the purposes of normalization.

7. Security Considerations

A URI does not in itself pose a security threat. However, since URIs are often used to provide a compact set of instructions for access to network resources, care must be taken to properly interpret the data within a URI, to prevent that data from causing unintended access, and to avoid including data that should not be revealed in plain text.

7.1 Reliability and Consistency

There is no guarantee that, having once used a given URI to retrieve some information, the same information will be retrievable by that URI in the future. Nor is there any guarantee that the information retrievable via that URI in the future will be observably similar to that retrieved in the past. The URI syntax does not constrain how a given scheme or authority apportions its name space or maintains it over time. Such a guarantee can only be obtained from the person(s) controlling that name space and the resource in question. A specific URI scheme may define additional semantics, such as name persistence, if those semantics are required of all naming authorities for that scheme.

7.2 Malicious Construction

It is sometimes possible to construct a URI such that an attempt to perform a seemingly harmless, idempotent operation, such as the retrieval of a representation, will in fact cause a possibly damaging remote operation to occur. The unsafe URI is typically constructed

by specifying a port number other than that reserved for the network protocol in question. The client unwittingly contacts a site that is running a different protocol service. The content of the URI contains instructions that, when interpreted according to this other protocol, cause an unexpected operation. An example has been the use of a gopher URI to cause an unintended or impersonating message to be sent via a SMTP server.

Caution should be used when dereferencing a URI that specifies a TCP port number other than the default for the scheme, especially when it is a number within the reserved space.

Care should be taken when a URI contains escaped delimiters for a given protocol (for example, CR and LF characters for telnet protocols) that these octets are not unescaped before transmission. This might violate the protocol, but avoids the potential for such characters to be used to simulate an extra operation or parameter in that protocol which might lead to an unexpected and possibly harmful remote operation being performed.

7.3 Rare IP Address Formats

Although the URI syntax for IPv4address only allows the common, dotted-decimal form of IPv4 address literal, many implementations that process URIs make use of platform-dependent system routines, such as gethostbyname() and inet_aton(), to translate the string literal to an actual IP address. Unfortunately, such system routines often allow and process a much larger set of formats than those described in Section 3.2.2.

For example, many implementations allow dotted forms of three numbers, wherein the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address (e.g., a Class B network). Likewise, a dotted form of two numbers means the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address (Class A), and a single number (without dots) is interpreted as a 32-bit quantity and stored directly in the network address. Adding further to the confusion, some implementations allow each dotted part to be interpreted as decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

These additional IP address formats are not allowed in the URI syntax due to differences between platform implementations. However, they can become a security concern if an application attempts to filter access to resources based on the IP address in string literal format. If such filtering is performed, it is recommended that literals be converted to numeric form and filtered based on the numeric value, rather than a prefix or suffix of the string form.

7.4 Sensitive Information

It is clearly unwise to use a URI that contains a password which is intended to be secret. In particular, the use of a password within the userinfo component of a URI is strongly discouraged except in those rare cases where the 'password' parameter is intended to be public.

7.5 Semantic Attacks

Because the userinfo component is rarely used and appears before the hostname in the authority component, it can be used to construct a URI that is intended to mislead a human user by appearing to identify one (trusted) naming authority while actually identifying a different authority hidden behind the noise. For example

```
http://www.example.com&story=breaking_news@10.0.0.1/top_story.htm
```


might lead a human user to assume that the host is 'www.example.com', whereas it is actually '10.0.0.1'. Note that the misleading userinfo could be much longer than the example above.

A misleading URI, such as the one above, is an attack on the user's preconceived notions about the meaning of a URI, rather than an attack on the software itself. User agents may be able to reduce the impact of such attacks by visually distinguishing the various components of the URI when rendered, such as by using a different color or tone to render userinfo if any is present, though there is no general panacea. More information on URI-based semantic attacks can be found in [Siedzik].

8. Acknowledgments

This specification is derived from RFC 2396 [RFC2396], RFC 1808 [RFC1808], and RFC 1738 [RFC1738]; the acknowledgments in those documents still apply. It also incorporates the update (with corrections) for IPv6 literals in the host syntax, as defined by Robert M. Hinden, Brian E. Carpenter, and Larry Masinter in [RFC2732]. In addition, contributions by Reese Anschultz, Tim Bray,

Rob Cameron, Dan Connolly, Adam M. Costello, John Cowan, Jason Diamond, Martin Duerst, Stefan Eissing, Clive D.W. Feather, Pat Hayes, Henry Holtzman, Graham Klyne, Dan Kohn, Bruce Lilly, Andrew Main, Michael Mealling, Julian Reschke, Tomas Rokicki, Miles Sabin, Ronald Tschalaer, Marc Warne, Stuart Williams, and Henry Zongaro are gratefully acknowledged.

Normative References

[ASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[RFC2234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

Informative References

[RFC2277] Alvestrand, H., "IETF Policy on Character Sets and Languages", BCP 18, RFC 2277, January 1998.

[RFC1630] 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, June 1994.

[RFC1738] Berners-Lee, T., Masinter, L. and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, December 1994.

[RFC2396] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.

[RFC1123] Braden, R., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, October 1989.

[RFC1808] Fielding, R., "Relative Uniform Resource Locators", RFC 1808, June 1995.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.

[RFC2518] Goland, Y., Whitehead, E., Faizi, A., Carter, S. and D. Jensen, "HTTP Extensions for Distributed Authoring -- WEBDAV", RFC 2518, February 1999.

[RFC0952] Harrenstien, K., Stahl, M. and E. Feinler, "DoD Internet host table specification", RFC 952, October 1985.

[RFC3513] Hinden, R. and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture", RFC 3513, April 2003.

[RFC2732] Hinden, R., Carpenter, B. and L. Masinter, "Format for Literal IPv6 Addresses in URL's", RFC 2732, December 1999.

[RFC1736] Kunze, J., "Functional Recommendations for Internet Resource Locators", RFC 1736, February 1995.

[RFC1737] Masinter, L. and K. Sollins, "Functional Requirements for Uniform Resource Names", RFC 1737, December 1994.

[RFC2141] Moats, R., "URN Syntax", RFC 2141, May 1997.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

[RFC2110] Palme, J. and A. Hopmann, "MIME E-mail Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2110, March 1997.

[RFC2717] Petke, R. and I. King, "Registration Procedures for URL Scheme Names", BCP 35, RFC 2717, November 1999.

[Siedzik] Siedzik, R., "Semantic Attacks: What's in a URL?", April 2001.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

Authors' Addresses

Tim Berners-Lee
World Wide Web Consortium
MIT/LCS, Room NE43-356
200 Technology Square
Cambridge, MA 02139
USA

Phone: +1-617-253-5702
Fax: +1-617-258-5999
EMail: timbl@w3.org
URI: http://www.w3.org/People/Berners-Lee/

Roy T. Fielding
Day Software
2 Corporate Plaza, Suite 150
Newport Beach, CA 92660
USA

Phone: +1-949-999-2523
Fax: +1-949-644-5064
EMail: roy.fielding@day.com
URI: http://www.apache.org/~fielding/

Larry Masinter
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA

Phone: +1-408-536-3024
EMail: LMM@acm.org
URI: http://larry.masinter.net/

Appendix A. Collected ABNF for URI

```
abs-path      = "/" path-segments

absolute-URI  = scheme ":" hier-part [ "?" query ]

alphanum      = ALPHA / DIGIT

authority     = [ userinfo "@" ] host [ ":" port ]

dec-octet     = DIGIT                 ; 0-9
              / %x31-39 DIGIT         ; 10-99
              / "1" 2DIGIT            ; 100-199
              / "2" %x30-34 DIGIT     ; 200-249
              / "25" %x30-35          ; 250-255

domainlabel   = alphanum [ 0*61( alphanum / "-" ) alphanum ]

escaped       = "%" HEXDIG HEXDIG

fragment      = *( pchar / "/" / "?" )

h4            = 1*4HEXDIG

hier-part     = net-path / abs-path / rel-path
```

```
    host          = [ IPv6reference / IPv4address / hostname ]

    hostname      = domainlabel qualified

    IPv4address   = dec-octet "." dec-octet "." dec-octet "." dec-octet

    IPv6address   =                            6( h4 ":" ) ls32
                  /                       "::" 5( h4 ":" ) ls32
                  / [               h4 ] "::" 4( h4 ":" ) ls32
                  / [ *1( h4 ":" ) h4 ] "::" 3( h4 ":" ) ls32
                  / [ *2( h4 ":" ) h4 ] "::" 2( h4 ":" ) ls32
                  / [ *3( h4 ":" ) h4 ] "::"    h4 ":"   ls32
                  / [ *4( h4 ":" ) h4 ] "::"             ls32
                  / [ *5( h4 ":" ) h4 ] "::"             h4
                  / [ *6( h4 ":" ) h4 ] "::"

    IPv6reference = "[" IPv6address "]"

    ls32          = ( h4 ":" h4 ) / IPv4address

    mark          = "-" / "_" / "." / "!" / "~" / "*" / "'" / "(" / ")"
```



```
    net-path      = "//" authority [ abs-path ]

    path-segments = segment *( "/" segment )

    pchar         = unreserved / escaped / ";" /
                    ":" / "@" / "&" / "=" / "+" / "$" / ","

    port          = *DIGIT

    qualified     = *( "." domainlabel ) [ "." ]

    query         = *( pchar / "/" / "?" )

    rel-path      = path-segments

    relative-URI  = hier-part [ "?" query ] [ "#" fragment ]

    reserved      = "/" / "?" / "#" / "[" / "]" / ";" /
```

```
                  ":" / "@" / "&" / "=" / "+" / "$" / ","

    scheme        = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )

    segment       = *pchar

    unreserved    = ALPHA / DIGIT / mark

    URI           = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

    URI-reference = URI / relative-URI

    uric          = reserved / unreserved / escaped

    userinfo      = *( unreserved / escaped / ";" /
                       ":" / "&" / "=" / "+" / "$" / "," )
```


Appendix B. Parsing a URI Reference with a Regular Expression

Since the "first-match-wins" algorithm is identical to the "greedy" disambiguation method used by POSIX regular expressions, it is natural and commonplace to use a regular expression for parsing the potential five components of a URI reference.

The following line is the regular expression for breaking-down a well-formed URI reference into its components.

```
^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?
```

```
    12            3  4          5       6  7        8 9
```

The numbers in the second line above are only to assist readability; they indicate the reference points for each subexpression (i.e., each paired parenthesis). We refer to the value matched for subexpression <n> as $<n>. For example, matching the above expression to

http://www.ics.uci.edu/pub/ietf/uri/#Related

results in the following subexpression matches:

```
$1 = http:
$2 = http
$3 = //www.ics.uci.edu
$4 = www.ics.uci.edu
$5 = /pub/ietf/uri/
$6 = <undefined>
$7 = <undefined>
$8 = #Related
$9 = Related
```

where <undefined> indicates that the component is not present, as is the case for the query component in the above example. Therefore, we can determine the value of the four components and fragment as

```
scheme    = $2
authority = $4
path      = $5
query     = $7
fragment  = $9
```

and, going in the opposite direction, we can recreate a URI reference from its components using the algorithm of Section 5.3.

Appendix C. Delimiting a URI in Context

URIs are often transmitted through formats that do not provide a clear context for their interpretation. For example, there are many

occasions when a URI is included in plain text; examples include text sent in electronic mail, USENET news messages, and, most importantly, printed on paper. In such cases, it is important to be able to delimit the URI from the rest of the text, and in particular from punctuation marks that might be mistaken for part of the URI.

In practice, URI are delimited in a variety of ways, but usually within double-quotes "http://example.com/", angle brackets <http://example.com/>, or just using whitespace

```
   http://example.com/
```

These wrappers do not form part of the URI.

In the case where a fragment identifier is associated with a URI reference, the fragment would be placed within the brackets as well (separated from the URI with a "#" character).

In some cases, extra whitespace (spaces, line-breaks, tabs, etc.) may need to be added to break a long URI across lines. The whitespace should be ignored when extracting the URI.

No whitespace should be introduced after a hyphen ("-") character. Because some typesetters and printers may (erroneously) introduce a hyphen at the end of line when breaking a line, the interpreter of a URI containing a line break immediately after a hyphen should ignore all unescaped whitespace around the line break, and should be aware that the hyphen may or may not actually be part of the URI.

Using <> angle brackets around each URI is especially recommended as a delimiting style for a URI that contains whitespace.

The prefix "URL:" (with or without a trailing space) was formerly recommended as a way to help distinguish a URI from other bracketed designators, though it is not commonly used in practice and is no longer recommended.

For robustness, software that accepts user-typed URI should attempt to recognize and strip both delimiters and embedded whitespace.

For example, the text:

> Yes, Jim, I found it under "http://www.w3.org/Addressing/", but you can probably pick it up from <ftp://ds.internic.net/rfc/>. Note the warning in <http://www.ics.uci.edu/pub/ietf/uri/historical.html#WARNING>.

contains the URI references

> http://www.w3.org/Addressing/
> ftp://ds.internic.net/rfc/
> http://www.ics.uci.edu/pub/ietf/uri/historical.html#WARNING

Appendix D. Summary of Non-editorial Changes

D.1 Additions

IPv6 literals have been added to the list of possible identifiers for the host portion of a authority component, as described by [RFC2732], with the addition of "[" and "]" to the reserved and uric sets. Square brackets are now specified as reserved within the authority component and not allowed outside their use as delimiters for an IPv6reference within host. In order to make this change without changing the technical definition of the path, query, and fragment components, those rules were redefined to directly specify the characters allowed rather than be defined in terms of uric.

Since [RFC2732] defers to [RFC3513] for definition of an IPv6 literal address, which unfortunately lacks an ABNF description of IPv6address, we created a new ABNF rule for IPv6address that matches the text representations defined by Section 2.2 of [RFC3513]. Likewise, the definition of IPv4address has been improved in order to limit each decimal octet to the range 0-255, and the definition of hostname has been improved to better specify length limitations and partially-qualified domain names.

Section 6 (Section 6) on URI normalization and comparison has been completely rewritten and extended using input from Tim Bray and discussion within the W3C Technical Architecture Group. Likewise, Section 2.1 on the encoding of characters has been replaced.

An ABNF production for URI has been introduced to correspond to the common usage of the term: an absolute URI with optional fragment.

D.2 Modifications from RFC 2396

The ad-hoc BNF syntax has been replaced with the ABNF of [RFC2234]. This change required all rule names that formerly included underscore characters to be renamed with a dash instead.

Section 2.2 on reserved characters has been rewritten to clearly explain what characters are reserved, when they are reserved, and why they are reserved even when not used as delimiters by the generic

syntax. Likewise, the section on escaped characters has been rewritten, and URI normalizers are now given license to unescape any octets corresponding to unreserved characters. The number-sign ("#") character has been moved back from the excluded delims to the reserved set.

The ABNF for URI and URI-reference has been redesigned to make them more friendly to LALR parsers and significantly reduce complexity. As

a result, the layout form of syntax description has been removed, along with the uric-no-slash, opaque-part, and rel-segment productions. All references to "opaque" URIs have been replaced with a better description of how the path component may be opaque to hierarchy. The fragment identifier has been moved back into the section on generic syntax components and within the URI and relative-URI productions, though it remains excluded from absolute-URI. The ambiguity regarding the parsing of URI-reference as a URI or a relative-URI with a colon in the first segment is now explained and disambiguated in the section defining relative-URI.

The ABNF of hier-part and relative-URI has been corrected to allow a relative URI path to be empty. This also allows an absolute-URI to consist of nothing after the "scheme:", as is present in practice with the "DAV:" namespace [RFC2518] and the "about:" URI used by many browser implementations. The ambiguity regarding the parsing of net-path, abs-path, and rel-path is now explained and disambiguated in the same section.

Registry-based naming authorities that use the generic syntax authority component are now limited to DNS hostnames, since those have been the only such URIs in deployment. This change was necessary to enable internationalized domain names to be processed in their native character encodings at the application layers above URI processing. The reg_name, server, and hostport productions have been removed to simplify parsing of the URI syntax.

The ABNF of qualified has been simplified to remove a parsing ambiguity without changing the allowed syntax. The toplabel production has been removed because it served no useful purpose. The ambiguity regarding the parsing of host as IPv4address or hostname is now explained and disambiguated in the same section.

The resolving relative references algorithm of [RFC2396] has been rewritten using pseudocode for this revision to improve clarity and fix the following issues:

- [RFC2396] section 5.2, step 6a, failed to account for a base URI with no path.

- Restored the behavior of [RFC1808] where, if the reference contains an empty path and a defined query component, then the target URI inherits the base URI's path component.

- Removed the special-case treatment of same-document references in favor of a section that explains that a new retrieval action should not be made if the target URI and base URI, excluding fragments, match. This change has no impact on user agent

behavior aside from how the resolved reference might be described to the user.

- Separated the path merge routine into two routines: merge, for describing combination of the base URI path with a relative-path reference, and remove_dot_segments, for describing how to remove the special "." and ".." segments from a composed path. The remove_dot_segments algorithm is now applied to all URI reference paths in order to match common implementations and improve the normalization of URIs in practice. This change only impacts the parsing of abnormal references and same-scheme references wherein the base URI has a non-hierarchical path.

Index

URN 6
userinfo 18

Intellectual Property Statement

Full Copyright Statement

revoked by the Internet Society or its successors or assignees.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION

HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.