Network Working Group
Internet-Draft
Updates: 1738 (if approved)
Obsoletes: 2732, 2396, 1808 (if approved)
Expires: September 1, 2003

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

March 3, 2003

Uniform Resource Identifier (URI): Generic Syntax
draft-fielding-uri-rfc2396bis-01

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 document defines the generic syntax of a URI, including both absolute and relative forms, and guidelines for their use.

This document 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 type. This document does not define a generative

grammar for all URIs; that task will be 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/>.

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 objects 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 G.

1.1 Overview of URIs

URIs are characterized by the following definitions:

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

A resource can be anything that has identity. 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. Not all resources are network "retrievable"; e.g., human beings, corporations, and bound books in a library can also be considered resources.

The resource is the conceptual mapping to an entity or set of

entities, not necessarily the entity which corresponds to that mapping at any particular instance in time. Thus, a resource can remain constant even when its content---the entities to which it currently corresponds---changes over time, provided that the conceptual mapping is not changed in the process.

Identifier

An identifier is an object that can act as a reference to something that has identity. In the case of a URI, the object is a sequence of characters with a restricted syntax.

Having identified a resource, a system may perform a variety of operations on the resource, as might be characterized by such words as `access', `update', `replace', or `find attributes'.

1.2 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 the 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 the subset of URIs that 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 cast into one of a discrete set of URI types such as "URL", "URN", "URC", etc. Any given URI scheme may define subspaces that have the characteristics of a name, a locator, or both, often depending on the persistence and care in the assignment of identifiers by the naming authority, rather than on any quality of the URI scheme. For that reason, this specification deprecates use of the terms URL or URN to distinguish between schemes, instead using the term URI throughout.

Each URI scheme (Section 3.1) defines the namespace of the URI, and thus may further restrict the syntax and semantics of identifiers using that scheme. This specification defines those elements of the URI syntax that are either 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.

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 in contexts that are

purely for identification, just like any other identifier. Even when a URI is used to obtain a representation of a resource, that access might be through gateways, proxies, caches, and name resolution services that are independent of the protocol of the resource origin, 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 resource when it can't be found in a local cache).

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.

1.3 Example URIs

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

1.4 Hierarchical URIs and Relative Forms

An absolute identifier refers to a resource independent of the context in which the identifier is used. In contrast, a relative identifier refers to a resource by describing the difference within a hierarchical namespace between the current context and an absolute identifier of the resource.

Some URI schemes support a hierarchical naming system, where the hierarchy of the name is denoted by a "/" delimiter separating the components in the scheme. This document defines a scheme-independent

`relative' form of URI reference that can be used in conjunction with a `base' URI of a hierarchical scheme to produce the `absolute' URI form of the reference. The syntax of a hierarchical URI is described in Section 3; the relative URI calculation is described in Section 5.

1.5 URI Transcribability

The URI syntax was designed with global transcribability as one of its main concerns. A URI is a sequence of characters from a very limited set, i.e. 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 transcribability 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 concerns revealed by the scenario:

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

- A URI may be transcribed from a non-network source, and thus should consist of characters that are most likely to be able to be typed 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 components.

These design concerns 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 the resource identifier from one medium to another was considered more important than having its URI consist of the most meaningful of components. In local and 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 document.

1.6 Syntax Notation and Common Elements

This document uses two conventions to describe and define the syntax for URI. The first, called the layout form, is a general description of the order of components and component separators, as in

    <first>/<second>;<third>?<fourth>

The component names are enclosed in angle-brackets and any characters outside angle-brackets are literal separators. Whitespace should be ignored. These descriptions are used informally and do not define the syntax requirements.

The second convention is a formal grammar defined using the Augmented Backus-Naur Form (ABNF) notation of [RFC2234]. Although the ABNF defines syntax in terms of the 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 complete URI syntax is collected in Appendix A.

2. URI Characters and Escape Sequences

A URI consists of a restricted set of characters, primarily chosen to aid transcribability and usability both in computer systems and in non-computer communications. Characters used conventionally as delimiters around a URI are excluded. The restricted set of 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, characters are either used as delimiters or to represent strings of data (octets) within the delimited portions. Octets are either represented directly by a character (using the US-ASCII character for that octet [ASCII]) or by an escape encoding. This representation is elaborated below.

2.1 URIs and non-ASCII characters

The relationship between URIs and characters has been a source of confusion for characters that are not part of US-ASCII. To describe the relationship, it is useful to distinguish between a "character" (as a distinguishable semantic entity) and an "octet" (an 8-bit byte). There are two mappings, one from URI characters to octets, and a second from octets to original characters:

URI character sequence->octet sequence->original character sequence

A URI is represented as a sequence of characters, not as a sequence of octets. That is because a URI might be "transported" by means that are not through a computer network, e.g., printed on paper, read over the radio, etc.

Within a delimited component of a URI, a sequence of characters is used to represent a sequence of octets. For example, the character "a" represents the octet 97 (decimal), while the character sequence "%", "0", "a" represents the octet 10 (decimal).

There is a second translation for some resources: the sequence of octets defined by a component of the URI is subsequently used to represent a sequence of characters. A 'charset' defines this mapping. There are many charsets in use in Internet protocols. For example,

UTF-8 [UTF-8] defines a mapping from sequences of octets to sequences of characters in the repertoire of ISO 10646.

In the simplest case, the original character sequence contains only characters that are defined in US-ASCII, and the two levels of

mapping are simple and easily invertible: each 'original character' is represented as the octet for the US-ASCII code for it, which is, in turn, represented as either the US-ASCII character, or else the "%" escape sequence for that octet.

For original character sequences that contain non-ASCII characters, however, the situation is more difficult. Internet protocols that transmit octet sequences intended to represent character sequences are expected to provide some way of identifying the charset used, if there might be more than one [RFC2277]. However, there is currently no provision within the generic URI syntax to accomplish this identification. An individual URI scheme may require a single charset, define a default charset, or provide a way to indicate the charset used. For example, a new scheme "foo" might be defined such that any escaped octet is keyed to the UTF-8 encoding in order to determine the corresponding Unicode character.

It is expected that a systematic treatment of character encoding within URIs will be developed as a future modification of this specification.

2.2 Reserved Characters

Many URI include components consisting of or delimited by, certain special characters. These characters are called "reserved", since their usage within the URI component is limited to their reserved purpose. If the data for a URI component would conflict with the reserved purpose, then the conflicting data must be escaped before forming the URI.

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

The "reserved" syntax class above refers to those characters that are allowed within a URI, but which may not be allowed within a particular component of the generic URI syntax; they are used as

delimiters of the components described in Section 3.

Characters in the "reserved" set are not reserved in all contexts. The set of characters actually reserved within any given URI component is defined by that component. In general, a character is reserved if the semantics of the URI changes if the character is replaced with its escaped US-ASCII encoding.

2.3 Unreserved Characters

Data characters that are allowed in a URI but do not have a reserved purpose are called unreserved. These include upper and lower case letters, decimal digits, and a limited set of punctuation marks and symbols.

```
unreserved  = ALPHA / DIGIT / mark

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

Unreserved characters can be escaped without changing the semantics of the URI, but this should not be done 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), underscore (%5F), or tilde (%7E) without fear of creating a conflict, but unescaping the other mark characters is usually counterproductive.

2.4 Escape Sequences

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 that corresponds to any US-ASCII character that is disallowed, as explained below.

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 the octet code in . For example, "%20" is the escaped encoding for the US-ASCII space character.

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

2.4.2 When to Escape and Unescape

A URI is always in an "escaped" form, since escaping or unescaping a completed URI might change its semantics. Normally, the only time escape encodings can safely be made is when the URI is being created 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. Likewise, a URI must be separated into its components before the escaped characters within those components can be safely decoded.

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. If the given URI scheme defines a canonicalization algorithm, then unreserved characters may be unescaped according to that algorithm. For example, "%7e" is sometimes used instead of "~" in an http URI path, but the two are equivalent for an http URI.

Because the percent "%" character always has the reserved purpose of being the escape indicator, it must be escaped as "%25" in order 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.

2.4.3 Excluded US-ASCII Characters

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

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 or typeset or subjected to the treatment of word-processing programs. Whitespace is also used to delimit a URI in many contexts.

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 character "#" is excluded because it is used to delimit a URI from a fragment identifier in a URI reference (Section 4). The percent character "%" is excluded because it is used for the encoding of escaped characters.

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

Other characters are excluded because gateways and other transport agents are known to sometimes modify such characters, or they are used as delimiters.

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


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

3. URI Syntactic Components

The URI syntax is dependent upon the scheme. In general, absolute URIs are written as follows:

<scheme>:<scheme-specific-part>

An absolute URI contains the name of the scheme being used (<scheme>)

followed by a colon (":") and then a string (the <scheme-specific-part>) whose interpretation depends on the scheme.

The URI syntax does not require that the scheme-specific-part have any general structure or set of semantics which is common among all URIs. However, a subset of URI do share a common syntax for representing hierarchical relationships within the namespace. This "generic URI" syntax consists of a sequence of four main components:

```
<scheme>://<authority><path>?<query>
```

each of which, except <scheme>, may be absent from a particular URI. For example, some URI schemes do not allow an <authority> component, and others do not use a <query> component.

```
absolute-URI  = scheme ":" ( hier-part / opaque-part )
```

URIs that are hierarchical in nature use the slash "/" character for separating hierarchical components. For some file systems, a "/" character (used to denote the hierarchical structure of a URI) is the delimiter used to construct a file name hierarchy, and thus the URI path will look similar to a file pathname. This does NOT imply that the resource is a file or that the URI maps to an actual filesystem pathname.

```
hier-part     = [ net-path / abs-path ] [ "?" query ]

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

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

URIs that do not make use of the slash "/" character for separating hierarchical components are considered opaque by the generic URI parser.

```
opaque-part   = uric-no-slash *uric

uric-no-slash = unreserved / escaped / "[" / "]" / ";" / "?" /
                ":" / "@" / "&" / "=" / "+" / "$" / ","
```


We use the term <path> to refer to both the <abs-path> and

<opaque-part> constructs, since they are mutually exclusive for any given URI and can be parsed as a single component.

3.1 Scheme Component

Just as there are many different methods of access to resources, there are a variety of schemes for identifying such resources. The URI syntax consists of a sequence of components separated by reserved characters, with the first component defining the semantics for the remainder of the URI string.

Scheme names consist of a sequence of characters beginning with a lower case letter and followed by any combination of lower case letters, digits, plus ("+"), period ("."), or hyphen ("-"). For resiliency, programs interpreting a URI should treat upper case letters as equivalent to lower case in scheme names (e.g., allow "HTTP" as well as "http").

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

Relative URI references are distinguished from absolute URI in that they do not begin with a scheme name. Instead, the scheme is inherited from the base URI, as described in Section 5.2.

3.2 Authority Component

Many URI schemes include a top hierarchical element for a naming authority, such that the namespace defined by the remainder of the URI is governed by that authority. This authority component is typically defined by an Internet-based server or a scheme-specific registry of naming authorities.

```
   authority     = server / reg-name
```

The authority component is preceded by a double slash "//" and is terminated by the next slash "/", question-mark "?", or by the end of the URI. Within the authority component, the characters ";", ":", "@", "?", "/", "[", and "]" are reserved.

An authority component is not required for a URI scheme to make use of relative references. A base URI without an authority component implies that any relative reference will also be without an authority component.

3.2.1 Registry-based Naming Authority

The structure of a registry-based naming authority is specific to the URI scheme, but constrained to the allowed characters for an authority component.

```
reg-name      = 1*( unreserved / escaped / ";" /
                    ":" / "@" / "&" / "=" / "+" / "$" / "," )
```

3.2.2 Server-based Naming Authority

URI schemes that involve the direct use of an IP-based protocol to a specified server on the Internet use a common syntax for the server component of the URI's scheme-specific data:

```
<userinfo>@<host>:<port>
```

where <userinfo> may consist of a user name and, optionally, scheme-specific information about how to gain authorization to access the server. The parts "<userinfo>@" and ":<port>" may be omitted. If <host> is omitted, the default host is defined by the scheme-specific semantics of the URI (e.g., the "file" URI scheme defaults to "localhost", whereas the "http" URI scheme does not allow host to be omitted).

```
server        = [ [ userinfo "@" ] hostport ]
```

The user information, if present, is followed by a commercial at-sign "@".

```
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 which is crafted to look like a trusted domain name might be used to mislead users, as described in Section 7.5.

The server is identified by a network host --- as described by an IPv6 literal encapsulated within square brackets, an IPv4 address in dotted-decimal form, or a domain name --- and an optional port number. The server's port, if any is required by the URI scheme, can be specified by a port number in decimal following the host and

delimited from it by a colon (":") character. If no explicit port number is given, the default port number, as defined by the URI

scheme, is assumed. The type of network port identified by the URI (e.g., TCP, UDP, SCTP, etc.) is defined by the scheme-specific semantics of the URI scheme.

```
   hostport      = host [ ":" port ]
   host          = IPv6reference / IPv4address / hostname
   port          = *DIGIT
```

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 will never start with a digit, thus syntactically distinguishing domain names from IPv4 addresses, and may be followed by a single "." if it is necessary to distinguish between the complete domain name and any local domain.

```
   hostname      = domainlabel qualified
   qualified     = *( "." domainlabel ) [ "." toplabel "." ]
   domainlabel   = alphanum [ 0*61( alphanum | "-" ) alphanum ]
   toplabel      = alpha    [ 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 [RFC2373] is distinguished by enclosing the IPv6 literal within square-brakets ("[" and "]"). This is the only place where square-bracket characters are allowed in the hierarchical 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
```

3.3 Path Component

The path component contains data, specific to the authority (or the scheme if there is no authority component), identifying the resource within the scope of that scheme and authority.

```
path          = [ abs-path / opaque-part ]

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

```
segment       = *pchar

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

The path may consist of a sequence of path segments separated by a single slash "/" character. Within a path segment, the characters "/", ";", "=", and "?" are reserved. The semicolon (";") and equals ("=") characters have the reserved purpose of delimiting parameters and parameter values within a path segment. However, parameters are not significant to the parsing of relative references.

3.4 Query Component

The query component is a string of information to be interpreted by the resource.

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

Within a query component, the characters ";", "/", "?", ":", "@", "&", "=", "+", ",", and "$" are reserved.

4. URI References

The term "URI-reference" is used here to denote the common usage of a resource identifier. A URI reference may be absolute or relative, and may have additional information attached in the form of a fragment identifier. However, "the URI" that results from such a reference includes only the absolute URI after the fragment identifier (if any) is removed and after any relative URI is resolved to its absolute form. Although it is possible to limit the discussion of URI syntax and semantics to that of the absolute result, most usage of URI is within general URI references, and it is impossible to obtain the URI from such a reference without also parsing the fragment and resolving the relative form.

```
URI-reference = [ absolute-URI / relative-URI ] [ "#" fragment ]
```

Many protocol elements allow only the absolute form of a URI with an optional fragment identifier.

```
absolute-URI-reference = absolute-URI [ "#" fragment ]
```

The syntax for a relative URI is a shortened form of that for an absolute URI, where some prefix of the URI is missing and certain path components ("." and "..") have a special meaning when, and only when, interpreting a relative path. The relative URI syntax is defined in Section 5.

4.1 Fragment Identifier

When a URI reference is used to perform a retrieval action on the identified resource, the optional fragment identifier, separated from the URI by a crosshatch ("#") character, consists of additional reference information to be interpreted by the user agent after the retrieval action has been successfully completed. As such, it is not part of a URI, but is often used in conjunction with a URI.

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

The semantics of a fragment identifier is a property of the data resulting from a retrieval action, regardless of the type of URI used in the reference. Therefore, the format and interpretation of fragment identifiers is dependent on the media type [RFC2046] of the retrieval result. The character restrictions described in Section 2 for a URI also apply to the fragment in a URI-reference. Individual media types may define additional restrictions or structure within the fragment for specifying different types of "partial views" that can be identified within that media type.

A fragment identifier is only meaningful when a URI reference is intended for retrieval and the result of that retrieval is a document for which the identified fragment is consistently defined.

4.2 Same-document References

A URI reference that does not contain a URI is a reference to the current document. In other words, an empty URI reference within a document is interpreted as a reference to the start of that document, and a reference containing only a fragment identifier is a reference to the identified fragment of that document. Traversal of such a

reference should not result in an additional retrieval action. However, if the URI reference occurs in a context that is always intended to result in a new request, as in the case of HTML's FORM element [HTML], then an empty URI reference represents the base URI of the current document and should be replaced by that URI when transformed into a request.

4.3 Parsing a URI Reference

A URI reference is typically parsed according to the four main components and fragment identifier in order to determine what components are present and whether the reference is relative or absolute. The individual components are then parsed for their subparts and, if not opaque, to verify their validity.

Although the BNF defines what is allowed in each component, it is ambiguous in terms of differentiating between an authority component and a path component that begins with two slash characters. The greedy algorithm is used for disambiguation: the left-most matching rule soaks up as much of the URI reference string as it is capable of matching. In other words, the authority component wins.

Readers familiar with regular expressions should see Appendix B for a concrete parsing example and test oracle.

5. Relative URI References

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 addressing 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 URIs. Furthermore, such document trees can be moved, as a whole, without changing any of the relative references. Experience within the WWW has demonstrated that the ability to perform relative referencing is necessary for the long-term usability of embedded URIs.

The relative URI syntax takes advantage of the <hier-part> syntax of <absolute-URI> (Section 3) in order to express a reference that is relative to the namespace of another hierarchical URI.

```
relative-URI  = [ net-path / abs-path / rel-path ] [ "?" query ]
```

A relative reference beginning with two slash characters is termed a network-path reference, as defined by <net-path> in Section 3. Such references are rarely used.

A relative reference beginning with a single slash character is termed an absolute-path reference, as defined by <abs-path> in Section 3.

A relative reference that does not begin with a scheme name or a slash character is termed a relative-path reference.

```
rel-path      = rel-segment [ abs-path ]

rel-segment   = 1*( unreserved / escaped / ";" /
                    "@" / "&" / "=" / "+" / "$" / "," )
```

Within a relative-path reference, the complete path segments "." and ".." have special meanings: "the current hierarchy level" and "the level above this hierarchy level", respectively. Although this is very similar to their use within Unix-based filesystems to indicate directory levels, these path components are only considered special when resolving a relative-path reference to its absolute form (Section 5.2).

Authors should be aware that a path segment which contains a colon character cannot be used as the first segment of a relative URI path (e.g., "this:that"), because it would be mistaken for a scheme name. It is therefore necessary to precede such segments with other segments (e.g., "./this:that") in order for them to be referenced as a relative path.

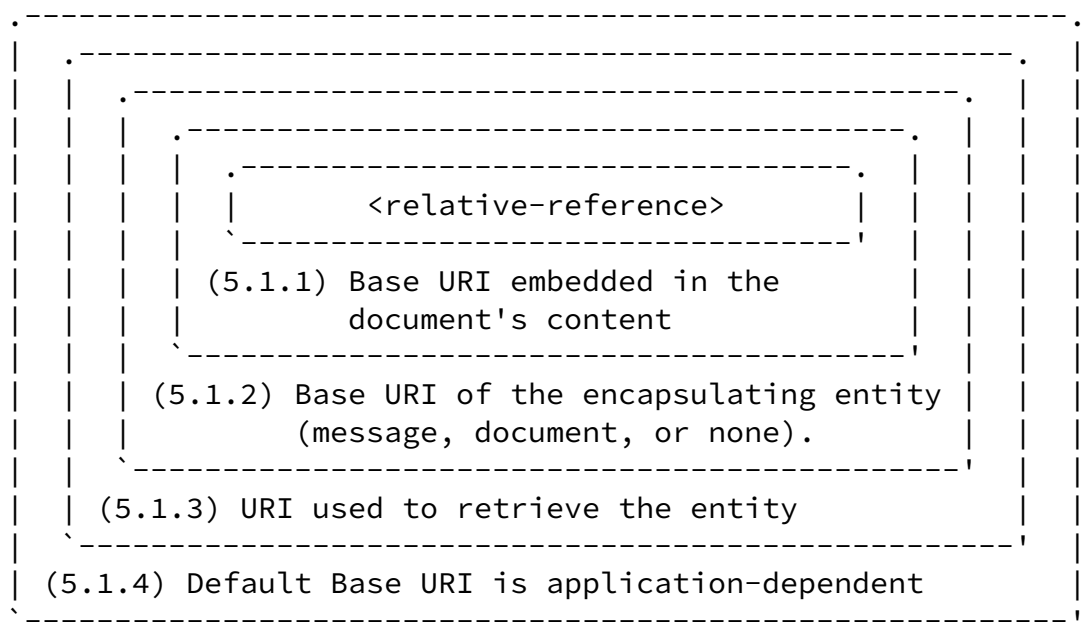
It is not necessary for all URI within a given scheme to be restricted to the <hier-part> syntax, since the hierarchical properties of that syntax are only necessary when a relative URI is used within a particular document. Documents can only make use of a relative URI when their base URI fits within the <hier-part> syntax. It is assumed that any document which contains a relative reference will also have a base URI that obeys the syntax. In other words, a relative URI cannot be used within a document that has an unsuitable base URI.

Some URI schemes do not allow a hierarchical syntax matching the <hier-part> syntax, and thus cannot use relative references.

5.1 Establishing a Base URI

The term "relative URI" implies that there exists some absolute "base URI" against which the relative reference is applied. Indeed, the base URI is necessary to define the semantics of any relative URI reference; without it, a relative reference is meaningless. In order for relative URI to be usable within a document, the base URI of that document must be known to the parser.

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. An example of how the base URI can be embedded in the Hypertext Markup Language (HTML) [HTML] is provided in Appendix D.

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 which do allow some form of tagged metainformation 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 Sections 5.1.1--5.1.3 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 URI to ensure that the base URI for that document can be established. It must be emphasized that a relative URI cannot be used reliably in situations where the document's base URI is not well-defined.

5.2 Resolving Relative References to Absolute Form

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 is established according to the rules of Section 5.1 and parsed into the four main components as 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 base URI's query component is not used by the resolution algorithm and may be discarded.

For each URI reference (R), the following pseudocode describes an algorithm for transforming R into its target (T), which is either an absolute URI or the current document, and R's optional fragment:


```
(R.scheme, R.authority, R.path, R.query, fragment) = parse(R);
   -- The URI reference is parsed into the four components and
   -- fragment identifier, as described in Section 4.3.

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

if defined(R.scheme) then
   T.scheme    = R.scheme;
   T.authority = R.authority;
   T.path      = R.path;
   T.query     = R.query;
else
   if defined(R.authority) then
      T.authority = R.authority;
      T.path      = R.path;
      T.query     = R.query;
   else
      if (R.path == "") then
         if defined(R.query) then
            T.path  = Base.path;
            T.query = R.query;
         else
```

```
                -- An empty reference refers to the current document
                return (current-document, fragment);
             endif;
          else
             if (R.path starts-with "/") then
                T.path = R.path;
             else
                T.path = merge(Base.path, R.path);
             endif;
             T.query = R.query;
          endif;
          T.authority = Base.authority;
       endif;
       T.scheme = Base.scheme;
    endif;

    return (T, fragment);
```

The pseudocode above refers to a merge routine for merging a relative-path reference with the path of the base URI to obtain the target path. Although there are many ways to do this, we will describe a simple method using a separate string buffer:

1. All but the last segment of the base URI's path component is copied to the buffer. In other words, any characters after the last (right-most) slash character, if any, are excluded. If the base URI's path component is the empty string, then a single slash character ("/") is copied to the buffer.

2. The reference's path component is appended to the buffer string.

3. All occurrences of "./", where "." is a complete path segment, are removed from the buffer string.

4. If the buffer string ends with "." as a complete path segment, that "." is removed.

5. All occurrences of "/../", where is a complete path segment not equal to "..", are removed from the buffer string. Removal of these path segments is performed iteratively, removing the leftmost matching pattern on each iteration, until no matching pattern remains.

6. If the buffer string ends with "/..", where is a complete path segment not equal to "..", that "/.." is removed.

7. If the resulting buffer string still begins with one or more complete path segments of "..", then the reference is considered to be in error. Implementations may handle this error by retaining these components in the resolved path (i.e., treating them as part of the final URI), by removing them from the resolved path (i.e., discarding relative levels above the root), or by avoiding traversal of the reference.

8. The remaining buffer string is the target URI's path component.

Some systems may find it more efficient to implement the merge algorithm as a pair of path segment stacks being merged, rather than as a series of string pattern replacements.

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

The resulting target URI components and fragment can be recombined to provide the absolute form of the URI reference. Using pseudocode, this would be:

```
result = ""

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

if defined(T.authority) then
```

```
      append "//" to result;
      append T.authority to result;
   endif;

   append T.path to result;

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

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

   return result;
```

Note that we must be 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.

Resolution examples are provided in Appendix C.

6. URI 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 URI 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 separate 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, or 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 leftover relative path segments.

6.2.2.1 Case Normalization

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

6.2.2.2 Escape Normalization

The %-escape mechanism described in Section 2.4 is a frequent source of variance among otherwise identical URIs. One cause is the choice of upper-case or lower-case 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 upper-case 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 URI paths; if they are found, they can be removed by splitting the URI just after the "/" that starts the path, using the left half as the base URI and the right as a relative reference, and normalizing the URI by merging the two in in accordance with the relative URI processing algorithm (Section 5).

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 of hostport 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

   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 Good Practice When Using URIs

It is in the best interests of everyone to avoid false-negatives in comparing URIs, and to only require the minimum 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 lower-case characters.

   Always provide the hostname, if any, in lower-case characters.

   Only perform %-escaping where it is essential.

   Always use upper-case A-through-F characters when %-escaping.

   Use the UTF-8 character-to-octet mapping, whenever possible.

   Prevent /./ and /../ from appearing in absolute URI paths.

The choices listed above are motivated by observations that a high proportion of deployed software already use these techniques in practice 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, that the same information will be retievable 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 namespace or maintains it over time. Such a guarantee can only be obtained from the person(s) controlling that namespace 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 associated with a resource, 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 in fact running a different protocol. 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 using any URI that specifies a TCP port number other than the default for the protocol, 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 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 authority 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. Acknowledgements

This document is derived from RFC 2396 [RFC2396], RFC 1808 [RFC1808], and RFC 1738 [RFC1738]; the acknowledgements in those specifications 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, Dan Connolly, Adam M. Costello, Jason Diamond, Martin Duerst, Henry Holtzman, Graham Klyne, Dan Kohn, Bruce Lilly, Michael Mealling, Julian Reschke, Tomas Rokicki, Miles Sabin, Ronald Tschalaer, Marc Warne, Henry Zongaro, and Zefram 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.

Non-normative 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.

[RFC2373] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 2373, July 1998.

[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.

[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.

[HTML] Raggett, D., Le Hors, A. and I. Jacobs, "Hypertext Markup Language (HTML 4.01) Specification", December 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 BNF for URI

To be filled-in later.

Appendix B. Parsing a URI Reference with a Regular Expression

As described in Section 4.3, the generic URI syntax is not sufficient to disambiguate the components of some forms of URI. Since the "greedy algorithm" described in that section is identical to the disambiguation method used by POSIX regular expressions, it is natural and commonplace to use a regular expression for parsing the potential four components and fragment identifier of a URI reference.

The following line is the regular expression for breaking-down a 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 5.2.

Appendix C. Examples of Resolving Relative URI References

Within an object with a well-defined base URI of

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

the relative URI would be resolved as follows:

C.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	=	(current document)#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

C.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 start of the current document.

<>	=	(current document)

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
```

In practice, some implementations strip leading relative symbolic elements (".", "..") after applying a relative URI calculation, based on the theory that compensating for obvious author errors is better than allowing the request to fail. Thus, the above two references will be interpreted as "http://a/g" by some implementations.

Similarly, parsers must avoid treating "." and ".." as special when they are not complete components of a relative path.

```
/./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. 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 backwards compatibility.

```
http:g        =  http:g           ; for validating parsers
```

```
                    /  http://a/b/c/g   ; for backwards compatibility
```


Appendix D. Embedding the Base URI in HTML documents

It is useful to consider an example of how the base URI of a document can be embedded within the document's content. In this appendix, we describe how documents written in the Hypertext Markup Language (HTML) [HTML] can include an embedded base URI. This appendix does not form a part of the URI specification and should not be considered as anything more than a descriptive example.

HTML defines a special element "BASE" which, when present in the "HEAD" portion of a document, signals that the parser should use the BASE element's "HREF" attribute as the base URI for resolving any relative URI. The "HREF" attribute must be an absolute URI. Note that, in HTML, element and attribute names are case-insensitive. For example:

```
<!doctype html public "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML><HEAD>
<TITLE>An example HTML document</TITLE>
<BASE href="http://www.example.com/Test/a/b/c">
</HEAD><BODY>
... <A href="../x">a hypertext anchor</A> ...
</BODY></HTML>
```

A parser reading the example document should interpret the given relative URI "../x" as representing the absolute URI

```
<http://www.example.com/Test/a/x>
```

regardless of the context in which the example document was obtained.

Appendix E. Recommendations for Delimiting 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, linebreaks, 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 F. Abbreviated URIs

The URI syntax was designed for unambiguous reference to network 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 abbreviated URI references. That is, a reference consisting of only the authority and path portions of the identified resource, 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 abbreviated URI references, 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 an abbreviated URI has the same syntax as a relative URI path, abbreviated URI references cannot be used in contexts where relative URIs are expected. This limits the use of abbreviated URIs to places where there is no defined base URI, such as dialog boxes and off-line advertisements.

Appendix G. Summary of Non-editorial Changes

G.1 Additions

IPv6 literals have been added to the list of possible identifiers for the host portion of a server component, as described by [RFC2732], with the addition of "[" and "]" to the reserved, uric, and uric-no-slash sets. Square brackets are now specified as reserved for the authority component, allowed within the opaque part of an opaque URI, and not allowed in the hierarchical syntax except for 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 continuing to be defined in terms of uric.

Since [RFC2732] defers to [RFC2373] for definition of an IPv6 literal address, which unfortunately has an incorrect ABNF description of IPv6address, we created a new ABNF rule for IPv6address that matches the text representations defined by Section 2.2 of [RFC2373]. 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 on URI normalization and comparison has been completely rewritten and extended using input from Tim Bray and discussion within the W3C Technical Architecture Group.

G.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. Likewise, absoluteURI and relativeURI have been changed to absolute-URI and relative-URI, respectively, for consistency.

The ABNF of hier-part and relative-URI (Section 3) 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 ABNF of qualified has been simplified to remove a parsing ambiguity without changing the allowed syntax.

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 the reference contains an empty path and a defined query component, then the target URI inherits the base URI's path component.

Index

Intellectual Property Statement

Full Copyright Statement

Acknowledgement

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