Network Working Group
Request for Comments: 3986
STD: 66
Updates: 1738
Obsoletes: 2732, 2396, 1808
Category: Standards Track

T. Berners-Lee
W3C/MIT
R. Fielding
Day Software
L. Masinter
Adobe Systems

January 2005

Uniform Resource Identifier (URI): Generic Syntax

Status of This Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies 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, allowing an implementation to 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.

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 these identifiers dates from 1990 and is described in "Universal Resource Identifiers in WWW" [RFC1630]. The syntax 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 obsoletes [RFC2732], which introduced syntax for an IPv6 address. It excludes 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 [BCP35]. Advice for designers of new URI schemes can be found in [RFC2718]. All significant changes from RFC 2396 are noted in Appendix D.

This specification uses the terms "character" and "coded character set" in accordance with the definitions provided in [BCP19], and "character encoding" in place of what [BCP19] refers to as a "charset".

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. 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

This specification does not limit the scope of what might be a resource; rather, the term "resource" is used in a general sense for whatever might be identified by a URI. Familiar examples include an electronic document, an image, a source of information with a consistent purpose (e.g., "today's weather report for Los Angeles"), a service (e.g., an HTTP-to-SMS gateway), 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, the types of a relationship (e.g., "parent" or "employee"), or numeric values (e.g., zero, one, and infinity).

Identifier

An identifier embodies the information required to distinguish what is being identified from all other things within its scope of identification. Our use of the terms "identify" and "identifying" refer to this purpose of distinguishing one resource from all other resources, regardless of how that purpose is accomplished (e.g., by name, address, or context). These terms should not be mistaken as an assumption that an identifier defines or embodies the identity of what is referenced, though that may be the case for some identifiers. Nor should it be assumed that a system using URIs will access the resource identified: in many cases, URIs are used to denote resources without any intention that they be accessed. Likewise, the "one" resource identified might not be singular in nature (e.g., a resource might be a named set or a mapping that varies over time).

A URI is an identifier consisting of a sequence of characters matching the syntax rule named <URI> in Section 3. It enables uniform identification of resources via a separately defined extensible set of naming schemes (Section 3.1). How that identification is accomplished, assigned, or enabled is delegated to each scheme specification.

This specification does not place any limits on the nature of a resource, the reasons why an application might seek to refer to a resource, or the kinds of systems that might use URIs for the sake of identifying resources. This specification does not require that a URI persists in identifying the same resource over time, though that is a common goal of all URI schemes. Nevertheless, nothing in this

specification prevents an application from limiting itself to particular types of resources, or to a subset of URIs that maintains characteristics desired by that application.

URIs have a global scope and are interpreted consistently regardless of context, though the result of that interpretation may be in relation to the end-user's context. For example, "http://localhost/" has the same interpretation for every user of that reference, even though the network interface corresponding to "localhost" may be different for each end-user: interpretation is independent of access. However, an action made on the basis of that reference will take place in relation to the end-user's context, which implies that an action intended to refer to a globally unique thing must use a URI that distinguishes that resource from all other things. URIs that identify in relation to the end-user's local context should only be used when the context itself is a defining aspect of the resource, such as when an on-line help manual refers to a file on the end-user's file system (e.g., "file:///etc/hosts").

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 needed to implement a scheme-independent parsing mechanism for URI references, by which 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 a definition for the range of syntax allowed for all URIs, including those schemes that have yet to be defined. This decouples the evolution of identification schemes from the evolution of protocols, data formats, and implementations that make use of URIs.

A parser of the generic URI syntax can parse 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 example URIs illustrate several URI schemes and variations in their common syntax components:

ftp://ftp.is.co.za/rfc/rfc1808.txt

http://www.ietf.org/rfc/rfc2396.txt

ldap://[2001:db8::7]/c=GB?objectClass?one

mailto:John.Doe@example.com

news:comp.infosystems.www.servers.unix

tel:+1-816-555-1212

telnet://192.0.2.16:80/

urn:oasis:names:specification:docbook:dtd:xml:4.1.2

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) has been used historically to refer to both 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, and to any other URI with the properties of a name.

An individual scheme does not have 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 on any quality of the scheme. Future specifications and related documentation should use the general term "URI" rather than the more restrictive terms "URL" and "URN" [RFC3305].

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 character encoding octets. The interpretation of a URI depends only on the characters used and not on 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 able to be entered into a computer, within the constraints imposed by keyboards (and related input devices) across languages and locales.

- A URI often has 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 by this specification. Percent-encoded octets (Section 2.1) may be used within a URI to represent characters outside the range of the US-ASCII coded character set if this

representation is allowed by the scheme or by the protocol element in which the URI is referenced. Such a definition should specify the character encoding used to map those characters to octets prior to being percent-encoded for the URI.

1.2.2. Separating Identification from Interaction

A common misunderstanding of URIs is that they are only used to refer to accessible resources. The URI itself only provides identification; access to the resource is neither guaranteed nor implied by the presence of a URI. Instead, any operation 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 words such as "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; this resolution may require several iterations. To use that access mechanism to perform an action on the URI's resource is to "dereference" the URI.

When URIs are used within information retrieval 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 representation metadata describing those octets, that constitutes a record of the state of the resource at the time when 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. Depending on the protocols used to perform the retrieval, additional information might be supplied about the resource (resource metadata) and its relation to other resources.

URI references in information retrieval systems are designed to be late-binding: the result of an access is generally determined when it is accessed and may vary over time or due to other aspects of the interaction. These references are created in order to 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 these URIs 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. 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 order of decreasing significance 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 generic syntax uses the slash ("/"), question mark ("?"), and number sign ("#") characters to delimit 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, wherein the vast majority of URI references in these documents point to resources within the tree rather than outside 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 with relative references. Furthermore, such document trees can be moved, as a whole, without changing any of the relative references.

A relative reference (Section 4.2) refers to a resource by describing the difference within a hierarchical name space between the reference context and the target URI. The reference resolution algorithm,

presented in Section 5, defines how such a reference is transformed to the target URI. As relative references can only be used within the context of a hierarchical URI, designers of new URI schemes should use a syntax consistent with the generic syntax's hierarchical components unless there are compelling reasons to forbid relative referencing within that scheme.

> NOTE: Previous specifications used the terms "partial URI" and "relative URI" to denote a relative reference to a URI. As some readers misunderstood those terms to mean that relative URIs are a subset of URIs rather than a method of referencing URIs, this specification simply refers to them as relative references.

All URI references are parsed by generic syntax parsers when used. However, because hierarchical processing has no effect on an absolute URI used in a reference unless it contains one or more dot-segments (complete path segments of "." or "..", as described in Section 3.3), URI scheme specifications can define opaque identifiers by disallowing use of slash characters, question mark characters, and the URIs "scheme:." and "scheme:..".

1.3. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC2234], including the following core ABNF syntax rules defined by that specification: ALPHA (letters), CR (carriage return), DIGIT (decimal digits), DQUOTE (double quote), HEXDIG (hexadecimal digits), LF (line feed), and SP (space). The complete URI syntax is collected in Appendix A.

2. Characters

The URI syntax provides a method of encoding data, presumably for the sake of identifying a resource, as a sequence of characters. The URI characters are, in turn, frequently encoded as octets for transport or presentation. This specification does not mandate any particular character encoding for mapping between URI characters and the octets used to store or transmit those characters. When a URI appears in a protocol element, the character encoding is defined by that protocol; without such a definition, a URI is assumed to be in the same character encoding as the surrounding text.

The ABNF notation defines its terminal values to be non-negative integers (codepoints) based on the US-ASCII coded character set [ASCII]. Because a URI is a sequence of characters, we must invert that relation in order to understand the URI syntax. Therefore, the

integer values used by the ABNF must be mapped back to their corresponding characters via US-ASCII in order to complete the syntax rules.

A URI is composed from a limited set of characters consisting of digits, letters, and a few graphic symbols. A reserved subset of those characters may be used to delimit syntax components within a URI while the remaining characters, including both the unreserved set and those reserved characters not acting as delimiters, define each component's identifying data.

2.1. Percent-Encoding

A percent-encoding mechanism is used to represent a data octet in a component when that octet's corresponding character is outside the allowed set or is being used as a delimiter of, or within, the component. A percent-encoded 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 percent-encoding for the binary octet "00100000" (ABNF: %x20), which in US-ASCII corresponds to the space character (SP). Section 2.4 describes when percent-encoding and decoding is applied.

```
pct-encoded = "%" HEXDIG HEXDIG
```

The uppercase hexadecimal digits 'A' through 'F' are equivalent to the lowercase digits 'a' through 'f', respectively. If two URIs differ only in the case of hexadecimal digits used in percent-encoded octets, they are equivalent. For consistency, URI producers and normalizers should use uppercase hexadecimal digits for all percent-encodings.

2.2. Reserved Characters

URIs include components and subcomponents that are delimited by characters in the "reserved" set. These characters are called "reserved" because they may (or may not) be defined as delimiters by the generic syntax, by each scheme-specific syntax, or by the implementation-specific syntax of a URI's dereferencing algorithm. If data for a URI component would conflict with a reserved character's purpose as a delimiter, then the conflicting data must be percent-encoded before the URI is formed.

```
reserved    = gen-delims / sub-delims

gen-delims  = ":" / "/" / "?" / "#" / "[" / "]" / "@"

sub-delims  = "!" / "$" / "&" / "'" / "(" / ")"
            / "*" / "+" / "," / ";" / "="
```

The purpose of reserved characters is to provide a set of delimiting characters that are distinguishable from other data within a URI. URIs that differ in the replacement of a reserved character with its corresponding percent-encoded octet are not equivalent. Percent-encoding a reserved character, or decoding a percent-encoded octet that corresponds to a reserved character, will change how the URI is interpreted by most applications. Thus, characters in the reserved set are protected from normalization and are therefore safe to be used by scheme-specific and producer-specific algorithms for delimiting data subcomponents within a URI.

A subset of the reserved characters (gen-delims) is used as delimiters of the generic URI components described in Section 3. A component's ABNF syntax rule will not use the reserved or gen-delims rule names directly; instead, each syntax rule lists the characters allowed within that component (i.e., not delimiting it), and any of those characters that are also in the reserved set are "reserved" for use as subcomponent delimiters within the component. Only the most common subcomponents are defined by this specification; other subcomponents may be defined by a URI scheme's specification, or by the implementation-specific syntax of a URI's dereferencing algorithm, provided that such subcomponents are delimited by characters in the reserved set allowed within that component.

URI producing applications should percent-encode data octets that correspond to characters in the reserved set unless these characters are specifically allowed by the URI scheme to represent data in that component. If a reserved character is found in a URI component and no delimiting role is known for that character, then it must be interpreted as representing the data octet corresponding to that character's encoding in US-ASCII.

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, hyphen, period, underscore, and tilde.

```
unreserved  = ALPHA / DIGIT / "-" / "." / "_" / "~"
```

URIs that differ in the replacement of an unreserved character with its corresponding percent-encoded US-ASCII octet are equivalent: they identify the same resource. However, URI comparison implementations do not always perform normalization prior to comparison (see Section 6). For consistency, percent-encoded octets in the ranges of ALPHA (%41-%5A and %61-%7A), DIGIT (%30-%39), hyphen (%2D), period (%2E), underscore (%5F), or tilde (%7E) should not be created by URI producers and, when found in a URI, should be decoded to their corresponding unreserved characters by URI normalizers.

2.4. When to Encode or Decode

Under normal circumstances, the only time when octets within a URI are percent-encoded is during the process of producing the URI from its component parts. This is when an implementation determines which of the reserved characters are to be used as subcomponent delimiters and which can be safely used as data. Once produced, a URI is always in its percent-encoded form.

When a URI is dereferenced, the components and subcomponents significant to the scheme-specific dereferencing process (if any) must be parsed and separated before the percent-encoded octets within those components can be safely decoded, as otherwise the data may be mistaken for component delimiters. The only exception is for percent-encoded octets corresponding to characters in the unreserved set, which can be decoded at any time. For example, the octet corresponding to the tilde ("~") character is often encoded as "%7E" by older URI processing implementations; the "%7E" can be replaced by "~" without changing its interpretation.

Because the percent ("%") character serves as the indicator for percent-encoded octets, it must be percent-encoded as "%25" for that octet to be used as data within a URI. Implementations must not percent-encode or decode the same string more than once, as decoding an already decoded string might lead to misinterpreting a percent data octet as the beginning of a percent-encoding, or vice versa in the case of percent-encoding an already percent-encoded string.

2.5. Identifying Data

URI characters provide identifying data for each of the URI components, serving as an external interface for identification between systems. Although the presence and nature of the URI production interface is hidden from clients that use its URIs (and is thus beyond the scope of the interoperability requirements defined by this specification), it is a frequent source of confusion and errors in the interpretation of URI character issues. Implementers have to be aware that there are multiple character encodings involved in the

production and transmission of URIs: local name and data encoding, public interface encoding, URI character encoding, data format encoding, and protocol encoding.

Local names, such as file system names, are stored with a local character encoding. URI producing applications (e.g., origin servers) will typically use the local encoding as the basis for producing meaningful names. The URI producer will transform the local encoding to one that is suitable for a public interface and then transform the public interface encoding into the restricted set of URI characters (reserved, unreserved, and percent-encodings). Those characters are, in turn, encoded as octets to be used as a reference within a data format (e.g., a document charset), and such data formats are often subsequently encoded for transmission over Internet protocols.

For most systems, an unreserved character appearing within a URI component is interpreted as representing the data octet corresponding to that character's encoding in US-ASCII. Consumers of URIs assume that the letter "X" corresponds to the octet "01011000", and even when that assumption is incorrect, there is no harm in making it. 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 [STD63] (or some other superset of the US-ASCII character encoding) at an internal interface, thereby providing more meaningful identifiers than those resulting from simply percent-encoding the original octets.

For example, consider an information service that provides data, stored locally using an EBCDIC-based file system, to clients on the Internet through an HTTP server. When an author creates a file with the name "Laguna Beach" on that file system, the "http" URI corresponding to that resource is expected to contain the meaningful string "Laguna%20Beach". If, however, that server produces URIs by using an overly simplistic raw octet mapping, then the result would be a URI containing "%D3%81%87%A4%95%81@%C2%85%81%83%88". An internal transcoding interface fixes this problem by transcoding the local name to a superset of US-ASCII prior to producing the URI. Naturally, proper interpretation of an incoming URI on such an interface requires that percent-encoded octets be decoded (e.g., "%20" to SP) before the reverse transcoding is applied to obtain the local name.

In some cases, the internal interface between a URI component and the identifying data that it has been crafted to represent is much less direct than a character encoding translation. For example, portions of a URI might reflect a query on non-ASCII data, or numeric

coordinates on a map. Likewise, a URI scheme may define components with additional encoding requirements that are applied prior to forming the component and producing the URI.

When a new URI scheme defines a component that represents textual data consisting of characters from the Universal Character Set [UCS], the data should first be encoded as octets according to the UTF-8 character encoding [STD63]; then only those octets that do not correspond to characters in the unreserved set should be percent-encoded. For example, the character A would be represented as "A", the character LATIN CAPITAL LETTER A WITH GRAVE would be represented as "%C3%80", and the character KATAKANA LETTER A would be represented as "%E3%82%A2".

3. Syntax Components

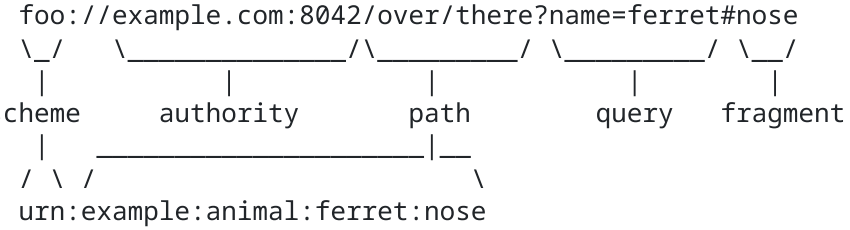
The generic URI syntax consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment.

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

   hier-part   = "//" authority path-abempty
               / path-absolute
               / path-rootless
               / path-empty
```

The scheme and path components are required, though the path may be empty (no characters). When authority is present, the path must either be empty or begin with a slash ("/") character. When authority is not present, the path cannot begin with two slash characters ("//"). These restrictions result in five different ABNF rules for a path (Section 3.3), only one of which will match any given URI reference.

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 schemes are case-insensitive, the canonical form is lowercase and documents that specify schemes must do so with 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 produce 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 [BCP35]. The scheme registry maintains the mapping between scheme names and their specifications. Advice for designers of new URI schemes can be found in [RFC2718]. URI scheme specifications must define their own syntax so that all strings matching their scheme-specific syntax will also match the <absolute-URI> grammar, as described in Section 4.3.

When presented with a URI that violates one or more scheme-specific restrictions, the scheme-specific 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, which might indicate that the URI has been constructed to mislead the user (Section 7.6).

3.2. Authority

Many URI schemes include a hierarchical element for a naming authority so 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 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 ]
```

URI producers and normalizers should omit the ":" delimiter that separates host from port if the port component is empty. Some schemes do not allow the userinfo and/or port subcomponents.

If a URI contains an authority component, then the path component must either be empty or begin with a slash ("/") character. Non-validating parsers (those that merely separate a URI reference into its major components) will often ignore the subcomponent structure of authority, treating it as an opaque string from the double-slash to the first terminating delimiter, until such time as the URI is dereferenced.

3.2.1. User Information

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

```
   userinfo    = *( unreserved / pct-encoded / sub-delims / ":" )
```

Use of the format "user:password" in the userinfo field is deprecated. Applications should not render as clear text any data after the first colon (":") character found within a userinfo subcomponent unless the data after the colon is the empty string (indicating no password). Applications may choose to ignore or reject such data when it is received as part of a reference and should reject the storage of such data in unencrypted form. The passing of authentication information in clear text has proven to be a security risk in almost every case where it has been used.

Applications that render a URI for the sake of user feedback, such as in graphical hypertext browsing, should render userinfo in a way that is distinguished from the rest of a URI, when feasible. Such rendering will assist the user in cases where the userinfo has been misleadingly crafted to look like a trusted domain name (Section 7.6).

3.2.2. Host

The host subcomponent of authority is identified by an IP literal encapsulated within square brackets, an IPv4 address in dotted-decimal form, or a registered name. The host subcomponent is case-insensitive. The presence of a host subcomponent 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 producer. In other cases, the data within the host component identifies a registered name that has nothing to do with an Internet host. We use the name "host" for the ABNF rule because that is its most common purpose, not its only purpose.

```
host        = IP-literal / IPv4address / reg-name
```

The syntax rule for host is ambiguous because it does not completely distinguish between an IPv4address and a reg-name. In order to disambiguate the syntax, we apply the "first-match-wins" algorithm: If host matches the rule for IPv4address, then it should be considered an IPv4 address literal and not a reg-name. Although host is case-insensitive, producers and normalizers should use lowercase for registered names and hexadecimal addresses for the sake of uniformity, while only using uppercase letters for percent-encodings.

A host identified by an Internet Protocol literal address, version 6 [RFC3513] or later, is distinguished by enclosing the IP literal within square brackets ("[" and "]"). This is the only place where square bracket characters are allowed in the URI syntax. In anticipation of future, as-yet-undefined IP literal address formats, an implementation may use an optional version flag to indicate such a format explicitly rather than rely on heuristic determination.

```
IP-literal = "[" ( IPv6address / IPvFuture  ) "]"

IPvFuture  = "v" 1*HEXDIG "." 1*( unreserved / sub-delims / ":" )
```

The version flag does not indicate the IP version; rather, it indicates future versions of the literal format. As such, implementations must not provide the version flag for the existing IPv4 and IPv6 literal address forms described below. If a URI containing an IP-literal that starts with "v" (case-insensitive), indicating that the version flag is present, is dereferenced by an application that does not know the meaning of that version flag, then the application should return an appropriate error for "address mechanism not supported".

A host identified by an IPv6 literal address is represented inside the square brackets without a preceding version flag. The ABNF provided here is a translation of the text definition of an IPv6 literal address provided in [RFC3513]. This syntax does not support IPv6 scoped addressing zone identifiers.

A 128-bit IPv6 address is divided into eight 16-bit pieces. Each piece is represented numerically in case-insensitive hexadecimal, using one to four hexadecimal digits (leading zeroes are permitted). The eight encoded pieces are given most-significant first, separated by colon characters. Optionally, the least-significant two pieces may instead be represented in IPv4 address textual format. A sequence of one or more consecutive zero-valued 16-bit pieces within the address may be elided, omitting all their digits and leaving exactly two consecutive colons in their place to mark the elision.

```
   IPv6address =                            6( h16 ":" ) ls32
               /                       "::" 5( h16 ":" ) ls32
               / [               h16 ] "::" 4( h16 ":" ) ls32
               / [ *1( h16 ":" ) h16 ] "::" 3( h16 ":" ) ls32
               / [ *2( h16 ":" ) h16 ] "::" 2( h16 ":" ) ls32
               / [ *3( h16 ":" ) h16 ] "::"    h16 ":"   ls32
               / [ *4( h16 ":" ) h16 ] "::"              ls32
               / [ *5( h16 ":" ) h16 ] "::"              h16
               / [ *6( h16 ":" ) h16 ] "::"

   ls32        = ( h16 ":" h16 ) / IPv4address
               ; least-significant 32 bits of address

   h16         = 1*4HEXDIG
               ; 16 bits of address represented in hexadecimal
```

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 7.4, 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 a registered name is a sequence of characters usually intended for lookup within a locally defined host or service name registry, though the URI's scheme-specific semantics may require that a specific registry (or fixed name table) be used instead. The most common name registry mechanism is the Domain Name System (DNS). A registered name intended for lookup in the DNS uses the syntax

defined in Section 3.5 of [RFC1034] and Section 2.1 of [RFC1123]. Such a name consists of 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 in DNS may be followed by a single "." and should be if it is necessary to distinguish between the complete domain name and some local domain.

```
   reg-name    = *( unreserved / pct-encoded / sub-delims )
```

If the URI scheme defines a default for host, then that default applies when the host subcomponent is undefined or when the registered name is empty (zero length). For example, the "file" URI scheme is defined so that no authority, an empty host, and "localhost" all mean the end-user's machine, whereas the "http" scheme considers a missing authority or empty host invalid.

This specification does not mandate a particular registered name lookup technology and therefore does not restrict the syntax of reg-name beyond what is necessary for interoperability. Instead, it delegates the issue of registered name syntax conformance to the operating system of each application performing URI resolution, and that operating system decides what it will allow for the purpose of host identification. A URI resolution implementation might use DNS, host tables, yellow pages, NetInfo, WINS, or any other system for lookup of registered names. However, a globally scoped naming system, such as DNS fully qualified domain names, is necessary for URIs intended to have global scope. URI producers should use names that conform to the DNS syntax, even when use of DNS is not immediately apparent, and should limit these names to no more than 255 characters in length.

The reg-name syntax allows percent-encoded octets in order to represent non-ASCII registered names in a uniform way that is independent of the underlying name resolution technology. Non-ASCII characters must first be encoded according to UTF-8 [STD63], and then each octet of the corresponding UTF-8 sequence must be percent-encoded to be represented as URI characters. URI producing applications must not use percent-encoding in host unless it is used to represent a UTF-8 character sequence. When a non-ASCII registered name represents an internationalized domain name intended for resolution via the DNS, the name must be transformed to the IDNA encoding [RFC3490] prior to name lookup. URI producers should provide these registered names in the IDNA encoding, rather than a percent-encoding, if they wish to maximize interoperability with legacy URI resolvers.

3.2.3. Port

The port subcomponent 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
```

A scheme may define a default port. For example, the "http" scheme defines a default port of "80", corresponding to its reserved TCP port number. The type of port designated by the port number (e.g., TCP, UDP, SCTP) is defined by the URI scheme. URI producers and normalizers should omit the port component and its ":" delimiter if port is empty or if its value would be the same as that of the scheme's default.

3.3. Path

The path component contains data, usually organized in hierarchical form, that, along with data in the non-hierarchical query component (Section 3.4), serves to identify a resource within the scope of the URI's scheme and naming authority (if any). The path is terminated by the first question mark ("?") or number sign ("#") character, or by the end of the URI.

If a URI contains an authority component, then the path component must either be empty or begin with a slash ("/") character. If a URI does not contain an authority component, then the path cannot begin with two slash characters ("//"). In addition, a URI reference (Section 4.1) may be a relative-path reference, in which case the first path segment cannot contain a colon (":") character. The ABNF requires five separate rules to disambiguate these cases, only one of which will match the path substring within a given URI reference. We use the generic term "path component" to describe the URI substring matched by the parser to one of these rules.

```
path          = path-abempty    ; begins with "/" or is empty
              / path-absolute   ; begins with "/" but not "//"
              / path-noscheme   ; begins with a non-colon segment
              / path-rootless   ; begins with a segment
              / path-empty      ; zero characters

path-abempty  = *( "/" segment )
path-absolute = "/" [ segment-nz *( "/" segment ) ]
path-noscheme = segment-nz-nc *( "/" segment )
path-rootless = segment-nz *( "/" segment )
path-empty    = 0<pchar>
```

```
segment       = *pchar
segment-nz    = 1*pchar
segment-nz-nc = 1*( unreserved / pct-encoded / sub-delims / "@" )
              ; non-zero-length segment without any colon ":"

pchar         = unreserved / pct-encoded / sub-delims / ":" / "@"
```

A 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). Use of the slash character to indicate hierarchy is only required when a URI will be used as the context for relative references. For example, the URI <mailto:fred@example.com> has a path of "fred@example.com", whereas the URI <foo://info.example.com?fred> has an empty path.

The path segments "." and "..", also known as dot-segments, 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) to indicate relative position within the hierarchical tree of names. This is similar to their role within some operating systems' file directory structures to indicate the current directory and parent directory, respectively. However, unlike in a file system, these dot-segments are only interpreted within the URI path hierarchy and are removed as part of the resolution process (Section 5.2).

Aside from dot-segments in hierarchical paths, a path segment is considered opaque by the generic syntax. URI producing applications often use the reserved characters allowed in a segment to delimit scheme-specific or dereference-handler-specific subcomponents. For example, the semicolon (";") and equals ("=") reserved characters are often used to delimit parameters and parameter values applicable to that segment. The comma (",") reserved character is often used for similar purposes. For example, one URI producer might use a segment such as "name;v=1.1" to indicate a reference to version 1.1 of "name", whereas another might use a segment such as "name,1.1" to indicate the same. Parameter types may be defined by scheme-specific semantics, but in most cases the syntax of a parameter is specific to the implementation of the URI's dereferencing algorithm.

3.4. Query

The query component contains non-hierarchical data that, along with data in the path component (Section 3.3), serves to identify a resource within the scope of the 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 ("?") may represent data within the query component. Beware that some older, erroneous implementations may not handle such data correctly when it is used as the base URI for relative references (Section 5.1), apparently because they fail to distinguish query data from path data when looking for hierarchical separators. However, as 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 avoid percent-encoding those characters.

3.5. Fragment

The fragment identifier component of a URI allows indirect identification of a secondary resource by reference to a primary resource and additional identifying information. 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 defined or described by those representations. A fragment identifier component is indicated by the presence of a number sign ("#") character and terminated by the end of the URI.

```
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 a potentially retrieved representation, even though such a retrieval is only performed if the URI is dereferenced. If no such representation exists, then the semantics of the fragment are considered unknown and are effectively unconstrained. Fragment identifier semantics are independent of the URI scheme and thus cannot be redefined by scheme specifications.

Individual media types may define their own restrictions on or structures 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 has multiple representations, as is often the case for resources whose representation is selected based on attributes of the retrieval request (a.k.a., content negotiation), then whatever is identified by the fragment should be consistent across all of those representations. Each representation should either define the fragment so that it corresponds to the same secondary resource, regardless of how it is represented, or should leave the fragment undefined (i.e., not found).

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 or will ever be accessed.

Fragment identifiers have a special role in information retrieval systems as the primary form of client-side indirect referencing, allowing an author to specifically identify aspects of an existing resource that are only indirectly provided by the resource owner. As such, the fragment identifier is not used in the scheme-specific processing of a URI; instead, the fragment identifier is separated from the rest of the URI prior to a dereference, and thus the identifying information within the fragment itself is dereferenced solely by the user agent, regardless of the URI scheme. Although this separate handling is often perceived to be a loss of information, particularly for accurate redirection of references as resources move over time, it also serves to prevent information providers from denying reference authors the right to refer to information within a resource selectively. Indirect referencing also provides additional flexibility and extensibility to systems that use URIs, as new media types are easier to define and deploy than new schemes of identification.

The characters slash ("/") and question mark ("?") are allowed to represent data within the fragment identifier. Beware that some older, erroneous implementations may not handle this data correctly when it is used as the base URI for relative references (Section 5.1).

4. Usage

When applications make reference to a URI, they do not always use the full form of reference defined by the "URI" syntax rule. To save space and take advantage of hierarchical locality, many Internet protocol elements and media type formats allow an abbreviation of a URI, whereas 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

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

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

A URI-reference is either a URI or a relative reference. If the URI-reference's prefix does not match the syntax of a scheme followed by its colon separator, then the URI-reference is a relative reference.

A URI-reference is typically parsed first into the five URI components, in order to determine what components are present and whether the reference is relative. 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 Reference

A relative reference takes advantage of the hierarchical syntax (Section 1.2.3) to express a URI reference relative to the name space of another hierarchical URI.

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

relative-part = "//" authority path-abempty
              / path-absolute
              / path-noscheme
              / path-empty
```

The URI referred to by a relative reference, also known as the target URI, 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, as 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 a base URI for later use by relative references calls for an absolute-URI syntax rule that does not allow a fragment.

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

URI scheme specifications must define their own syntax so that all strings matching their scheme-specific syntax will also match the <absolute-URI> grammar. Scheme specifications will not define fragment identifier syntax or usage, regardless of its applicability to resources identifiable via that scheme, as fragment identification is orthogonal to scheme definition. However, scheme specifications are encouraged to include a wide range of examples, including examples that show use of the scheme's URIs with fragment identifiers when such usage is appropriate.

4.4. Same-Document Reference

When a URI reference 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 a retrieval action, the target of that reference is defined to be within the same entity (representation, document, or message) as the reference; therefore, a dereference should not result in a new retrieval action.

Normalization of the base and target URIs prior to their comparison, as described in Sections 6.2.2 and 6.2.3, is allowed but rarely performed in practice. Normalization may increase the set of same-document references, which may be of benefit to some caching applications. As such, reference authors should not assume that a slightly different, though equivalent, reference URI will (or will not) be interpreted as a same-document reference by any given application.

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 a DNS registered name on its own. Such references are primarily intended for human interpretation rather than for machines, with the assumption that context-based heuristics are sufficient to complete the URI (e.g., most registered names 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.

Although this practice of using suffix references is common, it should be avoided whenever possible and should never be used in situations where long-term references are expected. The heuristics noted above will change over time, particularly when a new URI scheme becomes popular, and are often incorrect when used out of context. Furthermore, they can lead to security issues along the lines of those described in [RFC1535].

As 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 places where there is no defined base URI, such as dialog boxes and off-line advertisements.

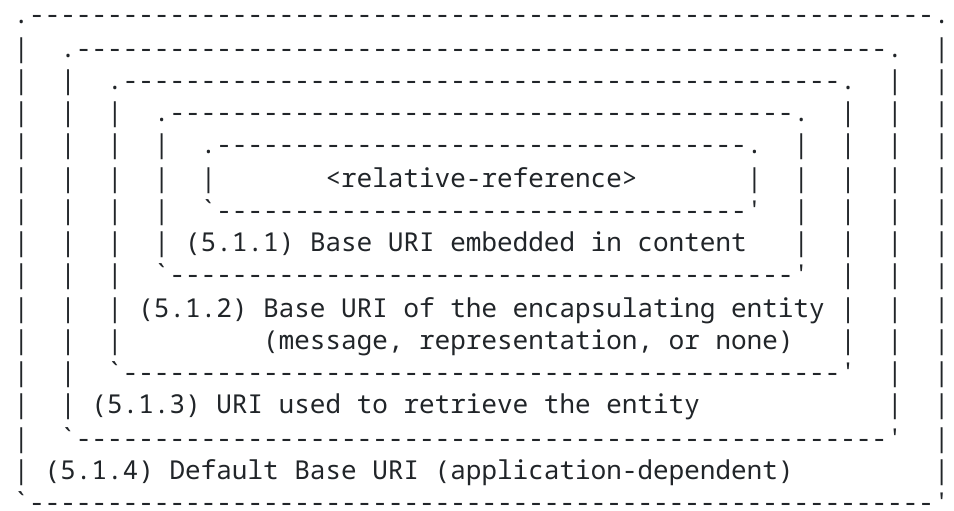
5. Reference Resolution

This section defines the process of resolving a URI reference within a context that allows relative references so that the result is a string matching the <URI> syntax rule of Section 3.

5.1. Establishing a Base URI

The term "relative" implies that a "base URI" exists against which the relative reference is applied. Aside from fragment-only references (Section 4.4), relative references are only usable when a base URI is known. A base URI must be established by the parser prior to parsing URI references that might be relative. A base URI must conform to the <absolute-URI> syntax rule (Section 4.3). If the base URI is obtained from a URI reference, then that reference must be converted to absolute form and stripped of any fragment component prior to its use as a base URI.

The base URI of a reference can be established in one of four ways, discussed 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 follows:



5.1.1. Base URI Embedded in Content

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

It is beyond the scope of this specification to specify how, for each media type, a base URI can be embedded. The appropriate syntax, when available, is described by the data format specification associated with each media type.

5.1.2. Base URI from the Encapsulating Entity

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

A mechanism for embedding a base URI within MIME container types (e.g., the message and multipart types) is defined by MHTML [RFC2557]. Protocols that do not use the MIME message header syntax, but that do allow some form of tagged metadata to be included within messages, may define their own syntax for defining a base URI as part of a message.

5.1.3. Base URI from the Retrieval URI

If no base URI is embedded and the representation is not encapsulated within some other entity, then, if a URI was used to retrieve the representation, 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., the URI that resulted in the actual retrieval of the representation) is the base URI.

5.1.4. Default Base URI

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

A sender of a representation containing relative references is responsible for ensuring that a base URI for those references can be established. Aside from fragment-only references, relative references can only be used reliably in situations where the base URI is well defined.

5.2. Relative Resolution

This section describes an algorithm for converting a URI reference that might be relative to a given base URI into the parsed components of the reference's target. The components can then be recomposed, as described in Section 5.3, to form the target URI. This algorithm provides definitive results that can be used to test the output of other implementations. Applications may implement relative reference resolution by using some other algorithm, provided that the results match what would be given by this one.

5.2.1. Pre-parse the Base URI

The base URI (Base) is established according to the procedure 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 a base URI; the other components may be empty or undefined. A component is undefined if its associated delimiter does not appear in the URI reference; the path component is never undefined, though it may be empty.

Normalization of the base URI, as described in Sections 6.2.2 and 6.2.3, is optional. A URI reference must be transformed to its target URI before it can be normalized.

5.2.2. Transform References

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;
```

5.2.3. Merge Paths

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 has a defined authority component and an empty path, then return a string consisting of "/" concatenated with the reference's path; 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., excluding any characters after the right-most "/" in the base URI path, or excluding the entire base URI path if it does not contain any "/" characters).

5.2.4. Remove Dot Segments

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 two string buffers.

1. The input buffer is initialized with the now-appended path components and the output buffer is initialized to the empty string.

2. While the input buffer is not empty, loop as follows:

   A. If the input buffer begins with a prefix of "../" or "./", then remove that prefix from the input buffer; otherwise,

   B. if the input buffer begins with a prefix of "/./" or "/.", where "." is a complete path segment, then replace that prefix with "/" in the input buffer; otherwise,

   C. if the input buffer begins with a prefix of "/../" or "/..", where ".." is a complete path segment, then replace that prefix with "/" in the input buffer and remove the last segment and its preceding "/" (if any) from the output buffer; otherwise,

   D. if the input buffer consists only of "." or "..", then remove that from the input buffer; otherwise,

   E. move the first path segment in the input buffer to the end of the output buffer, including the initial "/" character (if any) and any subsequent characters up to, but not including, the next "/" character or the end of the input buffer.

3. Finally, the output buffer is returned as the result of remove_dot_segments.

Note that dot-segments are intended for use in URI references to express an identifier relative to the hierarchy of names in the base URI. The remove_dot_segments algorithm respects that hierarchy by removing extra dot-segments rather than treat them as an error or leaving them to be misinterpreted by dereference implementations.

The following illustrates how the above steps are applied for two examples of merged paths, showing the state of the two buffers after each step.

```
   STEP   OUTPUT BUFFER          INPUT BUFFER

    1 :                          /a/b/c/./../../g
    2E:   /a                     /b/c/./../../g
    2E:   /a/b                   /c/./../../g
    2E:   /a/b/c                 /./../../g
    2B:   /a/b/c                 /../../g
    2C:   /a/b                   /../g
    2C:   /a                     /g
    2E:   /a/g

   STEP   OUTPUT BUFFER          INPUT BUFFER

    1 :                          mid/content=5/../6
    2E:   mid                    /content=5/../6
    2E:   mid/content=5          /../6
    2C:   mid                    /6
    2E:   mid/6
```

Some applications may find it more efficient to implement the remove_dot_segments algorithm by using two segment stacks rather than strings.

> Note: Beware that some older, erroneous implementations will fail to separate a reference's query component from its path component prior to merging the base and reference paths, resulting in an interoperability failure if the query component contains the strings "/../" or "/./".

5.3. Component Recomposition

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 a representation with a well defined base URI of

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

a relative reference is transformed to its target URI 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/d;p?q"
"."             =  "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 that above.

Parsers must be careful in handling cases where there are more ".." segments in a relative-path reference 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 reference 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 the path component before merging it with the base path and removing dot-segments. This error is rarely noticed, as 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 reference 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 whether 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 to reduce duplication of request actions and response storage.

URI comparison is performed for some particular purpose. Protocols or implementations that compare URIs for different purposes will often be subject to differing design trade-offs in regards to how much effort should be spent in reducing aliased identifiers. This section describes various 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

Because URIs exist to identify resources, presumably they should be considered equivalent when they identify the same resource. However, this definition of equivalence is not of much practical use, as there is no way for an implementation to compare two resources unless it has full knowledge or control of them. 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, URI comparison is not sufficient to determine whether two URIs identify different resources. For example, an owner of two different domain names could decide to serve the same resource from both, resulting in two different URIs. Therefore, comparison methods are designed to minimize false negatives while strictly avoiding false positives.

In testing for equivalence, applications should not directly compare relative references; the references should be converted to their respective target URIs before comparison. When URIs are compared to select (or avoid) a network action, such as retrieval of a representation, fragment components (if any) should be excluded from the 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 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 practices 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, when 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 strings for equality is normally based on pair comparison of the characters that make up the strings, starting from the first and proceeding until both strings are exhausted and all characters are found to be equal, until a pair of characters compares unequal, or until one of the strings is exhausted before the other.

This character comparison requires 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 in a Java String object (UTF-16), bit-for-bit comparisons applied naively will produce errors. It is better to speak of equality on a character-for-character basis rather than on a byte-for-byte or bit-for-bit basis. In practical terms, character-by-character comparisons should be done codepoint-by-codepoint after conversion to a common character encoding.

False negatives are caused by the production and use of URI aliases. Unnecessary aliases can be reduced, regardless of the comparison method, by consistently providing URI references in an already-normalized form (i.e., a form identical to what would be produced after normalization is applied, as described below).

Protocols and data formats often limit some URI comparisons to simple string comparison, based on the theory that people and implementations will, in their own best interest, be consistent in providing URI references, or at least consistent enough to negate any efficiency that might be obtained from further normalization.

6.2.2. Syntax-Based Normalization

Implementations may use logic based on the definitions provided by this specification to reduce the probability of false negatives. This 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/%7Bfoo%7D
eXAMPLE://a/./b/../b/%63/%7bfoo%7d
```

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, percent-encoding normalization, and removal of dot-segments.

6.2.2.1. Case Normalization

For all URIs, the hexadecimal digits within a percent-encoding triplet (e.g., "%3a" versus "%3A") are case-insensitive and therefore should be normalized to use uppercase letters for the digits A-F.

When a URI uses components of the generic syntax, the component syntax equivalence rules always apply; namely, that the scheme and host are case-insensitive and therefore should be normalized to lowercase. For example, the URI <HTTP://www.EXAMPLE.com/> is equivalent to <http://www.example.com/>. The other generic syntax components are assumed to be case-sensitive unless specifically defined otherwise by the scheme (see Section 6.2.3).

6.2.2.2. Percent-Encoding Normalization

The percent-encoding mechanism (Section 2.1) is a frequent source of variance among otherwise identical URIs. In addition to the case normalization issue noted above, some URI producers percent-encode octets that do not require percent-encoding, resulting in URIs that are equivalent to their non-encoded counterparts. These URIs should be normalized by decoding any percent-encoded octet that corresponds to an unreserved character, as described in Section 2.3.

6.2.2.3. Path Segment Normalization

The complete path segments "." and ".." are intended only for use within relative references (Section 4.1) and are removed as part of the reference resolution process (Section 5.2). However, some deployed implementations incorrectly assume that reference resolution is not necessary when the reference is already a URI and thus fail to remove dot-segments when they occur in non-relative paths. URI normalizers should remove dot-segments by applying the remove_dot_segments algorithm to the path, as described in Section 5.2.4.

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. Implementations may use scheme-specific rules, at further processing cost, to reduce the probability of false negatives. For example, because the "http" scheme makes use of an authority component, has a default port of "80", and defines an empty path to be equivalent to "/", the following four URIs are equivalent:

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

In general, a URI that uses the generic syntax for authority with an empty path should be normalized to a path of "/". Likewise, an explicit ":port", for which the port is empty or the default for the scheme, is equivalent to one where the port and its ":" delimiter are elided and thus should be removed by scheme-based normalization. For example, the second URI above is the normal form for the "http" scheme.

Another case where normalization varies by scheme is in the handling of an empty authority component or empty host subcomponent. For many scheme specifications, an empty authority or host is considered an error; for others, it is considered equivalent to "localhost" or the end-user's host. When a scheme defines a default for authority and a URI reference to that default is desired, the reference should be normalized to an empty authority for the sake of uniformity, brevity, and internationalization. If, however, either the userinfo or port subcomponents are non-empty, then the host should be given explicitly even if it matches the default.

Normalization should not remove delimiters when their associated component is empty unless licensed to do so by the scheme

specification. For example, the URI "http://example.com/?" cannot be assumed to be equivalent to any of the examples above. Likewise, the presence or absence of delimiters within a userinfo subcomponent is usually significant to its interpretation. The fragment component is not subject to any scheme-based normalization; thus, two URIs that differ only by the suffix "#" are considered different regardless of the scheme.

Some schemes define additional subcomponents that consist of case-insensitive data, giving an implicit license to normalizers to convert this data to a common case (e.g., all lowercase). For example, URI schemes that define a subcomponent of path to contain an Internet hostname, such as the "mailto" URI scheme, cause that subcomponent to be case-insensitive and thus subject to case normalization (e.g., "mailto:Joe@Example.COM" is equivalent to "mailto:Joe@example.com", even though the generic syntax considers the path component to be case-sensitive).

Other scheme-specific normalizations are possible.

6.2.4. Protocol-Based Normalization

Substantial effort to reduce the incidence of false negatives is often cost-effective for web spiders. Therefore, they 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. This kind of technique is only appropriate when equivalence is clearly indicated by both the result of accessing the resources and the common conventions of their scheme's dereference algorithm (in this case, use of redirection by HTTP origin servers to avoid problems with relative references).

7. Security Considerations

A URI does not in itself pose a security threat. However, as 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 once a URI has been used to retrieve 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 namespace or maintains it over time. Such guarantees 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 so 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. 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 running a different protocol service, and data within the URI contains instructions that, when interpreted according to this other protocol, cause an unexpected operation. A frequent example of such abuse has been the use of a protocol-based scheme with a port component of "25", thereby fooling user agent software into sending an unintended or impersonating message via an SMTP server.

Applications should prevent dereference of a URI that specifies a TCP port number within the "well-known port" range (0 - 1023) unless the protocol being used to dereference that URI is compatible with the protocol expected on that well-known port. Although IANA maintains a registry of well-known ports, applications should make such restrictions user-configurable to avoid preventing the deployment of new services.

When a URI contains percent-encoded octets that match the delimiters for a given resolution or dereference protocol (for example, CR and LF characters for the TELNET protocol), these percent-encodings must not be decoded before transmission across that protocol. Transfer of the percent-encoding, which might violate the protocol, is less harmful than allowing decoded octets to be interpreted as additional operations or parameters, perhaps triggering an unexpected and possibly harmful remote operation.

7.3. Back-End Transcoding

When a URI is dereferenced, the data within it is often parsed by both the user agent and one or more servers. In HTTP, for example, a typical user agent will parse a URI into its five major components, access the authority's server, and send it the data within the authority, path, and query components. A typical server will take that information, parse the path into segments and the query into key/value pairs, and then invoke implementation-specific handlers to respond to the request. As a result, a common security concern for server implementations that handle a URI, either as a whole or split into separate components, is proper interpretation of the octet data represented by the characters and percent-encodings within that URI.

Percent-encoded octets must be decoded at some point during the dereference process. Applications must split the URI into its components and subcomponents prior to decoding the octets, as otherwise the decoded octets might be mistaken for delimiters. Security checks of the data within a URI should be applied after decoding the octets. Note, however, that the "%00" percent-encoding (NUL) may require special handling and should be rejected if the application is not expecting to receive raw data within a component.

Special care should be taken when the URI path interpretation process involves the use of a back-end file system or related system functions. File systems typically assign an operational meaning to special characters, such as the "/", "\", ":", "[", and "]" characters, and to special device names like ".", "..", "...", "aux", "lpt", etc. In some cases, merely testing for the existence of such a name will cause the operating system to pause or invoke unrelated system calls, leading to significant security concerns regarding denial of service and unintended data transfer. It would be impossible for this specification to list all such significant characters and device names. Implementers should research the reserved names and characters for the types of storage device that may be attached to their applications and restrict the use of data obtained from URI components accordingly.

7.4. 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 that 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; 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 this filtering is performed, literals should be converted to numeric form and filtered based on the numeric value, and not on a prefix or suffix of the string form.

7.5. Sensitive Information

URI producers should not provide a URI that contains a username or password that is intended to be secret. URIs are frequently displayed by browsers, stored in clear text bookmarks, and logged by user agent history and intermediary applications (proxies). A password appearing within the userinfo component is deprecated and should be considered an error (or simply ignored) except in those rare cases where the 'password' parameter is intended to be public.

7.6. Semantic Attacks

Because the userinfo subcomponent is rarely used and appears before the host in the authority component, it can be used to construct a URI 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

```
ftp://cnn.example.com&story=breaking_news@10.0.0.1/top_story.htm
```

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

A misleading URI, such as that 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 distinguishing the various components of the URI when they are rendered, such as by using a different color or tone to render userinfo if any is present, though there is no panacea. More information on URI-based semantic attacks can be found in [Siedzik].

8. IANA Considerations

URI scheme names, as defined by <scheme> in Section 3.1, form a registered namespace that is managed by IANA according to the procedures defined in [BCP35]. No IANA actions are required by this document.

9. Acknowledgements

This specification is derived from RFC 2396 [RFC2396], RFC 1808 [RFC1808], and RFC 1738 [RFC1738]; the acknowledgements 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 Gisle Aas, Reese Anschultz, Daniel Barclay, Tim Bray, Mike Brown, Rob Cameron, Jeremy Carroll, Dan Connolly, Adam M. Costello, John Cowan, Jason Diamond, Martin Duerst, Stefan Eissing, Clive D.W. Feather, Al Gilman, Tony Hammond, Elliotte Harold, Pat Hayes, Henry Holtzman, Ian B. Jacobs, Michael Kay, John C. Klensin, Graham Klyne, Dan Kohn, Bruce Lilly, Andrew Main, Dave McAlpin, Ira McDonald, Michael Mealling, Ray Merkert, Stephen Pollei, Julian Reschke, Tomas Rokicki, Miles Sabin, Kai Schaetzl, Mark Thomson, Ronald Tschalaer, Norm Walsh, Marc Warne, Stuart Williams, and Henry Zongaro are gratefully acknowledged.

10. References

10.1. 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.

[STD63] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[UCS] International Organization for Standardization, "Information Technology - Universal Multiple-Octet Coded Character Set (UCS)", ISO/IEC 10646:2003, December 2003.

10.2. Informative References

[BCP19] Freed, N. and J. Postel, "IANA Charset Registration Procedures", BCP 19, RFC 2978, October 2000.

[BCP35] Petke, R. and I. King, "Registration Procedures for URL Scheme Names", BCP 35, RFC 2717, November 1999.

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

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

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

[RFC1535] Gavron, E., "A Security Problem and Proposed Correction With Widely Deployed DNS Software", RFC 1535, October 1993.

[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.

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

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

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

[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.

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

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

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

[RFC2557] Palme, J., Hopmann, A., and N. Shelness, "MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2557, March 1999.

[RFC2718] Masinter, L., Alvestrand, H., Zigmond, D., and R. Petke, "Guidelines for new URL Schemes", RFC 2718, November 1999.

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

[RFC3305] Mealling, M. and R. Denenberg, "Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations", RFC 3305, August 2002.

[RFC3490] Faltstrom, P., Hoffman, P., and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003.

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

[Siedzik] Siedzik, R., "Semantic Attacks: What's in a URL?", April 2001, <http://www.giac.org/practical/gsec/Richard_Siedzik_GSEC.pdf>.

Appendix A. Collected ABNF for URI

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

hier-part     = "//" authority path-abempty
              / path-absolute
              / path-rootless
              / path-empty

URI-reference = URI / relative-ref

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

relative-ref  = relative-part [ "?" query ] [ "#" fragment ]

relative-part = "//" authority path-abempty
              / path-absolute
              / path-noscheme
              / path-empty

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

authority     = [ userinfo "@" ] host [ ":" port ]
userinfo      = *( unreserved / pct-encoded / sub-delims / ":" )
host          = IP-literal / IPv4address / reg-name
port          = *DIGIT

IP-literal    = "[" ( IPv6address / IPvFuture  ) "]"

IPvFuture     = "v" 1*HEXDIG "." 1*( unreserved / sub-delims / ":" )

IPv6address   =                            6( h16 ":" ) ls32
              /                       "::" 5( h16 ":" ) ls32
              / [               h16 ] "::" 4( h16 ":" ) ls32
              / [ *1( h16 ":" ) h16 ] "::" 3( h16 ":" ) ls32
              / [ *2( h16 ":" ) h16 ] "::" 2( h16 ":" ) ls32
              / [ *3( h16 ":" ) h16 ] "::"    h16 ":"   ls32
              / [ *4( h16 ":" ) h16 ] "::"              ls32
              / [ *5( h16 ":" ) h16 ] "::"              h16
              / [ *6( h16 ":" ) h16 ] "::"

h16           = 1*4HEXDIG
ls32          = ( h16 ":" h16 ) / IPv4address
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

   reg-name      = *( unreserved / pct-encoded / sub-delims )

   path          = path-abempty    ; begins with "/" or is empty
                 / path-absolute   ; begins with "/" but not "//"
                 / path-noscheme   ; begins with a non-colon segment
                 / path-rootless   ; begins with a segment
                 / path-empty      ; zero characters

   path-abempty  = *( "/" segment )
   path-absolute = "/" [ segment-nz *( "/" segment ) ]
   path-noscheme = segment-nz-nc *( "/" segment )
   path-rootless = segment-nz *( "/" segment )
   path-empty    = 0<pchar>

   segment       = *pchar
   segment-nz    = 1*pchar
   segment-nz-nc = 1*( unreserved / pct-encoded / sub-delims / "@" )
                 ; non-zero-length segment without any colon ":"

   pchar         = unreserved / pct-encoded / sub-delims / ":" / "@"

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

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

   pct-encoded   = "%" HEXDIG HEXDIG

   unreserved    = ALPHA / DIGIT / "-" / "." / "_" / "~"
   reserved      = gen-delims / sub-delims
   gen-delims    = ":" / "/" / "?" / "#" / "[" / "]" / "@"
   sub-delims    = "!" / "$" / "&" / "'" / "(" / ")"
                 / "*" / "+" / "," / ";" / "="
```

Appendix B. Parsing a URI Reference with a Regular Expression

As 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 five components as

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

Going in the opposite direction, we can recreate a URI reference from its components by 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 email, USENET news, and on printed 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, URIs are delimited in a variety of ways, but usually within double-quotes "http://example.com/", angle brackets <http://example.com/>, or just by using whitespace:

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

These wrappers do not form part of the URI.

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

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 it, the interpreter of a URI containing a line break immediately after a hyphen should ignore all 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 reference that contains embedded 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://foo.example.com/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://foo.example.com/rfc/
> http://www.ics.uci.edu/pub/ietf/uri/historical.html#WARNING

Appendix D. Changes from RFC 2396

D.1. Additions

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

IPv6 (and later) literals have been added to the list of possible identifiers for the host portion of an authority component, as described by [RFC2732], with the addition of "[" and "]" to the reserved set and a version flag to anticipate future versions of IP literals. Square brackets are now specified as reserved within the authority component and are not allowed outside their use as delimiters for an IP literal 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.

As [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.

Section 6, on URI normalization and comparison, has been completely rewritten and extended by using input from Tim Bray and discussion within the W3C Technical Architecture Group.

D.2. Modifications

The ad-hoc BNF syntax of RFC 2396 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. In addition, a number of syntax rules have been eliminated or simplified to make the overall grammar more comprehensible. Specifications that refer to the obsolete grammar rules may be understood by replacing those rules according to the following table:

obsolete rule	translation
absoluteURI	absolute-URI
relativeURI	relative-part ["?" query]
hier_part	("//" authority path-abempty / path-absolute) ["?" query]
opaque_part	path-rootless ["?" query]
net_path	"//" authority path-abempty
abs_path	path-absolute
rel_path	path-rootless
rel_segment	segment-nz-nc
reg_name	reg-name
server	authority
hostport	host [":" port]
hostname	reg-name
path_segments	path-abempty
param	*<pchar excluding ";">
uric	unreserved / pct-encoded / ";" / "?" / ":" / "@" / "&" / "=" / "+" / "$" / "," / "/"
uric_no_slash	unreserved / pct-encoded / ";" / "?" / ":" / "@" / "&" / "=" / "+" / "$" / ","
mark	"-" / "_" / "." / "!" / "~" / "*" / "'" / "(" / ")"
escaped	pct-encoded
hex	HEXDIG
alphanum	ALPHA / DIGIT

Use of the above obsolete rules for the definition of scheme-specific syntax is deprecated.

Section 2, on characters, has been rewritten to explain what characters are reserved, when they are reserved, and why they are reserved, even when they are not used as delimiters by the generic syntax. The mark characters that are typically unsafe to decode, including the exclamation mark ("!"), asterisk ("*"), single-quote ("'"), and open and close parentheses ("(" and ")"), have been moved to the reserved set in order to clarify the distinction between reserved and unreserved and, hopefully, to answer the most common question of scheme designers. Likewise, the section on percent-encoded characters has been rewritten, and URI normalizers are now given license to decode any percent-encoded octets

corresponding to unreserved characters. In general, the terms "escaped" and "unescaped" have been replaced with "percent-encoded" and "decoded", respectively, to reduce confusion with other forms of escape mechanisms.

The ABNF for URI and URI-reference has been redesigned to make them more friendly to LALR parsers and to reduce complexity. As a result, the layout form of syntax description has been removed, along with the uric, uric_no_slash, opaque_part, net_path, abs_path, rel_path, path_segments, rel_segment, and mark rules. All references to "opaque" URIs have been replaced with a better description of how the path component may be opaque to hierarchy. The relativeURI rule has been replaced with relative-ref to avoid unnecessary confusion over whether they are a subset of URI. The ambiguity regarding the parsing of URI-reference as a URI or a relative-ref with a colon in the first segment has been eliminated through the use of five separate path matching rules.

The fragment identifier has been moved back into the section on generic syntax components and within the URI and relative-ref rules, though it remains excluded from absolute-URI. The number sign ("#") character has been moved back to the reserved set as a result of reintegrating the fragment syntax.

The ABNF has been corrected to allow the path component 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 with the "about:" scheme used internally by many WWW browser implementations. The ambiguity regarding the boundary between authority and path has been eliminated through the use of five separate path matching rules.

Registry-based naming authorities that use the generic syntax are now defined within the host rule. This change allows current implementations, where whatever name provided is simply fed to the local name resolution mechanism, to be consistent with the specification. It also removes the need to re-specify DNS name formats here. Furthermore, it allows the host component to contain percent-encoded octets, which is necessary to enable internationalized domain names to be provided in URIs, processed in their native character encodings at the application layers above URI processing, and passed to an IDNA library as a registered name in the UTF-8 character encoding. The server, hostport, hostname, domainlabel, toplabel, and alphanum rules have been removed.

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

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

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

- o The determination of whether a URI reference is a same-document reference has been decoupled from the URI parser, simplifying the URI processing interface within applications in a way consistent with the internal architecture of deployed URI processing implementations. The determination is now based on comparison to the base URI after transforming a reference to absolute form, rather than on the format of the reference itself. This change may result in more references being considered "same-document" under this specification than there would be under the rules given in RFC 2396, especially when normalization is used to reduce aliases. However, it does not change the status of existing same-document references.

- o 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 to 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

Authors' Addresses

Tim Berners-Lee
World Wide Web Consortium
Massachusetts Institute of Technology
77 Massachusetts Avenue
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
5251 California Ave., Suite 110
Irvine, CA 92617
USA

Phone: +1-949-679-2960
Fax: +1-949-679-2972
EMail: fielding@gbiv.com
URI: http://roy.gbiv.com/

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/

Full Copyright Statement

Intellectual Property

Acknowledgement

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