Network Working Group
Request for Comments: 2396
Updates: 1808, 1738
Category: Standards Track

T. Berners-Lee
MIT/LCS
R. Fielding
U.C. Irvine
L. Masinter
Xerox Corporation

August 1998

Uniform Resource Identifiers (URI): Generic Syntax

Status of this Memo

Copyright Notice

IESG Note

This paper describes a "superset" of operations that can be applied to URI. It consists of both a grammar and a description of basic functionality for URI. To understand what is a valid URI, both the grammar and the associated description have to be studied. Some of the functionality described is not applicable to all URI schemes, and some operations are only possible when certain media types are retrieved using the URI, regardless of the scheme used.

Abstract

A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource. This document defines the generic syntax of URI, including both absolute and relative forms, and guidelines for their use; it revises and replaces the generic definitions in RFC 1738 and RFC 1808.

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

1. Introduction

Uniform Resource Identifiers (URI) provide a simple and extensible means for identifying a resource. This specification of URI syntax and semantics is derived from concepts introduced by the World Wide Web global information initiative, whose use of such objects dates from 1990 and is described in "Universal Resource Identifiers in WWW" [RFC1630]. The specification of URI 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 updates and merges "Uniform Resource Locators" [RFC1738] and "Relative Uniform Resource Locators" [RFC1808] in order to define a single, generic syntax for all URI. It excludes those portions of RFC 1738 that defined the specific syntax of individual URL schemes; those portions will be updated as separate documents, as will the process for registration of new URI schemes. This document does not discuss the issues and recommendation for dealing with characters outside of the US-ASCII character set [ASCII]; those recommendations are discussed in a separate document.

All significant changes from the prior RFCs are noted in Appendix G.

1.1 Overview of URI

URI are characterized by the following definitions:

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

Resource
: A resource can be anything that has identity. Familiar examples include an electronic document, an image, a service (e.g., "today's weather report for Los Angeles"), and a collection of other resources. Not all resources are network "retrievable"; e.g., human beings, corporations, and bound books in a library can also be considered resources.

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

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

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

1.2. URI, URL, and URN

A URI can be further classified as a locator, a name, or both. The term "Uniform Resource Locator" (URL) refers to the subset of URI that identify resources via a representation of their primary access mechanism (e.g., their network "location"), rather than identifying the resource by name or by some other attribute(s) of that resource. The term "Uniform Resource Name" (URN) refers to the subset of URI that are required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable.

The URI scheme (Section 3.1) defines the namespace of the URI, and thus may further restrict the syntax and semantics of identifiers using that scheme. This specification defines those elements of the URI syntax that are either required of all URI schemes or are common to many URI schemes. It thus defines the syntax and semantics that are needed to implement a scheme-independent parsing mechanism for URI references, such that the scheme-dependent handling of a URI can be postponed until the scheme-dependent semantics are needed. We use the term URL below when describing syntax or semantics that only apply to locators.

Although many URL schemes are named after protocols, this does not imply that the only way to access the URL’s resource is via the named protocol. Gateways, proxies, caches, and name resolution services might be used to access some resources, independent of the protocol of their origin, and the resolution of some URL may require the use of more than one protocol (e.g., both DNS and HTTP are typically used to access an "http" URL’s resource when it can’t be found in a local cache).

A URN differs from a URL in that it's primary purpose is persistent labeling of a resource with an identifier. That identifier is drawn from one of a set of defined namespaces, each of which has its own set name structure and assignment procedures. The "urn" scheme has been reserved to establish the requirements for a standardized URN namespace, as defined in "URN Syntax" [RFC2141] and its related specifications.

Most of the examples in this specification demonstrate URL, since they allow the most varied use of the syntax and often have a hierarchical namespace. A parser of the URI syntax is capable of parsing both URL and URN references as a generic URI; once the scheme is determined, the scheme-specific parsing can be performed on the generic URI components. In other words, the URI syntax is a superset of the syntax of all URI schemes.

1.3. Example URI

The following examples illustrate URI that are in common use.

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

gopher://spinaltap.micro.umn.edu/00/Weather/California/Los%20Angeles
-- gopher scheme for Gopher and Gopher+ Protocol services

http://www.math.uio.no/faq/compression-faq/part1.html
-- http scheme for Hypertext Transfer Protocol services

mailto:mduerst@ifi.unizh.ch
-- mailto scheme for electronic mail addresses

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

telnet://melvyl.ucop.edu/
-- telnet scheme for interactive services via the TELNET Protocol

1.4. Hierarchical URI and Relative Forms

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

Some URI schemes support a hierarchical naming system, where the hierarchy of the name is denoted by a "/" delimiter separating the components in the scheme. This document defines a scheme-independent 'relative' form of URI reference that can be used in conjunction with a 'base' URI (of a hierarchical scheme) to produce another URI. The syntax of hierarchical URI is described in Section 3; the relative URI calculation is described in Section 5.

1.5. URI Transcribability

The URI syntax was designed with global transcribability as one of its main concerns. A URI is a sequence of characters from a very limited set, i.e. the letters of the basic Latin alphabet, digits, and a few special characters. A URI may be represented in a variety of ways: e.g., ink on paper, pixels on a screen, or a sequence of octets in a coded character set. The interpretation of a URI depends only on the characters used and not how those characters are represented in a network protocol.

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

There are several design concerns revealed by the scenario:

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

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

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

These design concerns are not always in alignment. For example, it is often the case that the most meaningful name for a URI component would require characters that cannot be typed into some systems. The ability to transcribe the resource identifier from one medium to another was considered more important than having its URI consist of the most meaningful of components. In local and regional contexts

and with improving technology, users might benefit from being able to use a wider range of characters; such use is not defined in this document.

1.6. Syntax Notation and Common Elements

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

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

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

The second convention is a BNF-like grammar, used to define the formal URI syntax. The grammar is that of [RFC822], except that "|" is used to designate alternatives. Briefly, rules are separated from definitions by an equal "=", indentation is used to continue a rule definition over more than one line, literals are quoted with "", parentheses "(" and ")" are used to group elements, optional elements are enclosed in "[" and "]" brackets, and elements may be preceded with <n>* to designate n or more repetitions of the following element; n defaults to 0.

Unlike many specifications that use a BNF-like grammar to define the bytes (octets) allowed by a protocol, the URI grammar is defined in terms of characters. Each literal in the grammar corresponds to the character it represents, rather than to the octet encoding of that character in any particular coded character set. How a URI is represented in terms of bits and bytes on the wire is dependent upon the character encoding of the protocol used to transport it, or the charset of the document which contains it.

The following definitions are common to many elements:

```
alpha    = lowalpha | upalpha

lowalpha = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
           "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
           "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

upalpha  = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
           "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
           "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
```

```
digit    = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
           "8" | "9"

alphanum = alpha | digit
```

The complete URI syntax is collected in Appendix A.

2. URI Characters and Escape Sequences

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

```
uric          = reserved | unreserved | escaped
```

Within a URI, characters are either used as delimiters, or to represent strings of data (octets) within the delimited portions. Octets are either represented directly by a character (using the US-ASCII character for that octet [ASCII]) or by an escape encoding. This representation is elaborated below.

2.1 URI and non-ASCII characters

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

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

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

A URI scheme may define a mapping from URI characters to octets; whether this is done depends on the scheme. Commonly, within a delimited component of a URI, a sequence of characters may be used to represent a sequence of octets. For example, the character "a" represents the octet 97 (decimal), while the character sequence "%", "0", "a" represents the octet 10 (decimal).

There is a second translation for some resources: the sequence of octets defined by a component of the URI is subsequently used to represent a sequence of characters. A 'charset' defines this mapping. There are many charsets in use in Internet protocols. For example, UTF-8 [UTF-8] defines a mapping from sequences of octets to sequences of characters in the repertoire of ISO 10646.

In the simplest case, the original character sequence contains only characters that are defined in US-ASCII, and the two levels of mapping are simple and easily invertible: each 'original character' is represented as the octet for the US-ASCII code for it, which is, in turn, represented as either the US-ASCII character, or else the "%" escape sequence for that octet.

For original character sequences that contain non-ASCII characters, however, the situation is more difficult. Internet protocols that transmit octet sequences intended to represent character sequences are expected to provide some way of identifying the charset used, if there might be more than one [RFC2277]. However, there is currently no provision within the generic URI syntax to accomplish this identification. An individual URI scheme may require a single charset, define a default charset, or provide a way to indicate the charset used.

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

2.2. Reserved Characters

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

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

The "reserved" syntax class above refers to those characters that are allowed within a URI, but which may not be allowed within a particular component of the generic URI syntax; they are used as delimiters of the components described in Section 3.

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

2.3. Unreserved Characters

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

```
unreserved  = alphanum | mark

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

Unreserved characters can be escaped without changing the semantics of the URI, but this should not be done unless the URI is being used in a context that does not allow the unescaped character to appear.

2.4. Escape Sequences

Data must be escaped if it does not have a representation using an unreserved character; this includes data that does not correspond to a printable character of the US-ASCII coded character set, or that corresponds to any US-ASCII character that is disallowed, as explained below.

2.4.1. Escaped Encoding

An escaped octet is encoded as a character triplet, consisting of the percent character "%" followed by the two hexadecimal digits representing the octet code. For example, "%20" is the escaped encoding for the US-ASCII space character.

```
escaped     = "%" hex hex
hex         = digit | "A" | "B" | "C" | "D" | "E" | "F" |
                      "a" | "b" | "c" | "d" | "e" | "f"
```

2.4.2. When to Escape and Unescape

A URI is always in an "escaped" form, since escaping or unescaping a completed URI might change its semantics. Normally, the only time escape encodings can safely be made is when the URI is being created from its component parts; each component may have its own set of characters that are reserved, so only the mechanism responsible for generating or interpreting that component can determine whether or

not escaping a character will change its semantics. Likewise, a URI must be separated into its components before the escaped characters within those components can be safely decoded.

In some cases, data that could be represented by an unreserved character may appear escaped; for example, some of the unreserved "mark" characters are automatically escaped by some systems. If the given URI scheme defines a canonicalization algorithm, then unreserved characters may be unescaped according to that algorithm. For example, "%7e" is sometimes used instead of "~" in an http URL path, but the two are equivalent for an http URL.

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

2.4.3. Excluded US-ASCII Characters

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

The control characters in the US-ASCII coded character set are not used within a URI, both because they are non-printable and because they are likely to be misinterpreted by some control mechanisms.

```
control     = <US-ASCII coded characters 00-1F and 7F hexadecimal>
```

The space character is excluded because significant spaces may disappear and insignificant spaces may be introduced when URI are transcribed or typeset or subjected to the treatment of word-processing programs. Whitespace is also used to delimit URI in many contexts.

```
space       = <US-ASCII coded character 20 hexadecimal>
```

The angle-bracket "<" and ">" and double-quote (") characters are excluded because they are often used as the delimiters around URI in text documents and protocol fields. The character "#" is excluded because it is used to delimit a URI from a fragment identifier in URI references (Section 4). The percent character "%" is excluded because it is used for the encoding of escaped characters.

```
delims      = "<" | ">" | "#" | "%" | <">
```

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

```
unwise      = "{" | "}" | "|" | "\" | "^" | "[" | "]" | "`"
```

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

3. URI Syntactic Components

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

```
<scheme>:<scheme-specific-part>
```

An absolute URI contains the name of the scheme being used (<scheme>) followed by a colon (":") and then a string (the <scheme-specific-part>) whose interpretation depends on the scheme.

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

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

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

```
absoluteURI   = scheme ":" ( hier_part | opaque_part )
```

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

```
hier_part     = ( net_path | abs_path ) [ "?" query ]

net_path      = "//" authority [ abs_path ]

abs_path      = "/"  path_segments
```

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

```
opaque_part   = uric_no_slash *uric

uric_no_slash = unreserved | escaped | ";" | "?" | ":" | "@" |
                "&" | "=" | "+" | "$" | ","
```

We use the term <path> to refer to both the <abs_path> and <opaque_part> constructs, since they are mutually exclusive for any given URI and can be parsed as a single component.

3.1. Scheme Component

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

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

```
scheme        = alpha *( alpha | digit | "+" | "-" | "." )
```

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

3.2. Authority Component

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

```
authority     = server | reg_name
```

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

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

3.2.1. Registry-based Naming Authority

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

```
reg_name      = 1*( unreserved | escaped | "$" | "," |
                    ";" | ":" | "@" | "&" | "=" | "+" )
```

3.2.2. Server-based Naming Authority

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

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

where <userinfo> may consist of a user name and, optionally, scheme-specific information about how to gain authorization to access the server. The parts "<userinfo>@" and ":<port>" may be omitted.

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

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

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

Some URL schemes use the format "user:password" in the userinfo field. This practice is NOT RECOMMENDED, because the passing of authentication information in clear text (such as URI) has proven to be a security risk in almost every case where it has been used.

The host is a domain name of a network host, or its IPv4 address as a set of four decimal digit groups separated by ".". Literal IPv6 addresses are not supported.

```
hostport      = host [ ":" port ]
host          = hostname | IPv4address
hostname      = *( domainlabel "." ) toplabel [ "." ]
domainlabel   = alphanum | alphanum *( alphanum | "-" ) alphanum
toplabel      = alpha | alpha *( alphanum | "-" ) alphanum
```

```
   IPv4address   = 1*digit "." 1*digit "." 1*digit "." 1*digit
   port          = *digit
```

Hostnames take the form described in Section 3 of [RFC1034] and Section 2.1 of [RFC1123]: a sequence of domain labels separated by ".", each domain label starting and ending with an alphanumeric character and possibly also containing "-" characters. The rightmost domain label of a fully qualified domain name will never start with a digit, thus syntactically distinguishing domain names from IPv4 addresses, and may be followed by a single "." if it is necessary to distinguish between the complete domain name and any local domain. To actually be "Uniform" as a resource locator, a URL hostname should be a fully qualified domain name. In practice, however, the host component may be a local domain literal.

Note: A suitable representation for including a literal IPv6 address as the host part of a URL is desired, but has not yet been determined or implemented in practice.

The port is the network port number for the server. Most schemes designate protocols that have a default port number. Another port number may optionally be supplied, in decimal, separated from the host by a colon. If the port is omitted, the default port number is assumed.

3.3. Path Component

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

```
   path          = [ abs_path | opaque_part ]

   path_segments = segment *( "/" segment )
   segment       = *pchar *( ";" param )
   param         = *pchar

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

The path may consist of a sequence of path segments separated by a single slash "/" character. Within a path segment, the characters "/", ";", "=", and "?" are reserved. Each path segment may include a sequence of parameters, indicated by the semicolon ";" character. The parameters are not significant to the parsing of relative references.

3.4. Query Component

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

```
query         = *uric
```

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

4. URI References

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

```
URI-reference = [ absoluteURI | relativeURI ] [ "#" fragment ]
```

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

4.1. Fragment Identifier

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

```
fragment      = *uric
```

The semantics of a fragment identifier is a property of the data resulting from a retrieval action, regardless of the type of URI used in the reference. Therefore, the format and interpretation of fragment identifiers is dependent on the media type [RFC2046] of the retrieval result. The character restrictions described in Section 2

for URI also apply to the fragment in a URI-reference. Individual media types may define additional restrictions or structure within the fragment for specifying different types of "partial views" that can be identified within that media type.

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

4.2. Same-document References

A URI reference that does not contain a URI is a reference to the current document. In other words, an empty URI reference within a document is interpreted as a reference to the start of that document, and a reference containing only a fragment identifier is a reference to the identified fragment of that document. Traversal of such a reference should not result in an additional retrieval action. However, if the URI reference occurs in a context that is always intended to result in a new request, as in the case of HTML's FORM element, then an empty URI reference represents the base URI of the current document and should be replaced by that URI when transformed into a request.

4.3. Parsing a URI Reference

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

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

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

5. Relative URI References

It is often the case that a group or "tree" of documents has been constructed to serve a common purpose; the vast majority of URI in these documents point to resources within the tree rather than

outside of it. Similarly, documents located at a particular site are much more likely to refer to other resources at that site than to resources at remote sites.

Relative addressing of URI allows document trees to be partially independent of their location and access scheme. For instance, it is possible for a single set of hypertext documents to be simultaneously accessible and traversable via each of the "file", "http", and "ftp" schemes if the documents refer to each other using relative URI. Furthermore, such document trees can be moved, as a whole, without changing any of the relative references. Experience within the WWW has demonstrated that the ability to perform relative referencing is necessary for the long-term usability of embedded URI.

The syntax for relative URI takes advantage of the <hier_part> syntax of <absoluteURI> (Section 3) in order to express a reference that is relative to the namespace of another hierarchical URI.

```
   relativeURI   = ( net_path | abs_path | rel_path ) [ "?" query ]
```

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

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

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

```
   rel_path      = rel_segment [ abs_path ]

   rel_segment   = 1*( unreserved | escaped |
                       ";" | "@" | "&" | "=" | "+" | "$" | "," )
```

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

Authors should be aware that a path segment which contains a colon character cannot be used as the first segment of a relative URI path (e.g., "this:that"), because it would be mistaken for a scheme name.

It is therefore necessary to precede such segments with other segments (e.g., "./this:that") in order for them to be referenced as a relative path.

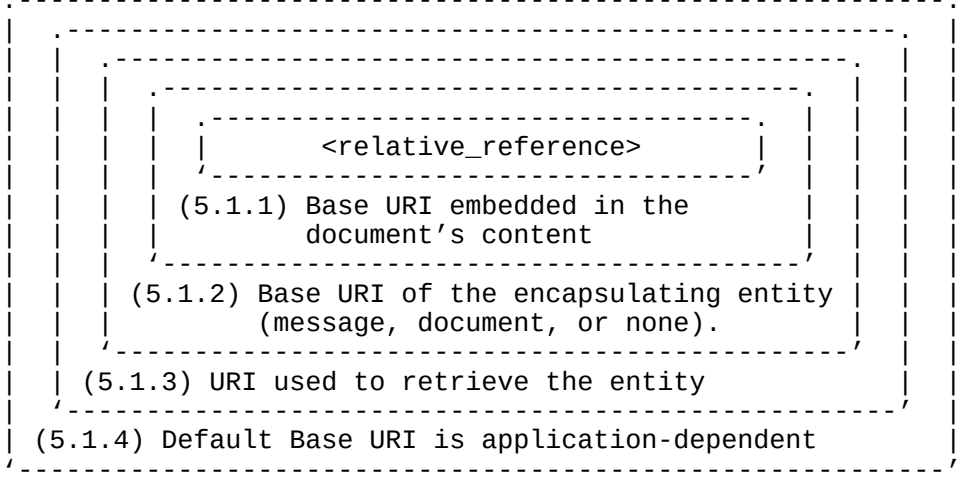
It is not necessary for all URI within a given scheme to be restricted to the <hier_part> syntax, since the hierarchical properties of that syntax are only necessary when relative URI are used within a particular document. Documents can only make use of relative URI when their base URI fits within the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

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

5.1. Establishing a Base URI

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

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



5.1.1. Base URI within Document Content

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

It is beyond the scope of this document to specify how, for each media type, the base URI can be embedded. It is assumed that user agents manipulating such media types will be able to obtain the appropriate syntax from that media type's specification. An example of how the base URI can be embedded in the Hypertext Markup Language (HTML) [RFC1866] is provided in Appendix D.

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

5.1.2. Base URI from the Encapsulating Entity

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

5.1.3. Base URI from the Retrieval URI

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

5.1.4. Default Base URI

If none of the conditions described in Sections 5.1.1--5.1.3 apply, then the base URI is defined by the context of the application. Since this definition is necessarily application-dependent, failing

to define the base URI using one of the other methods may result in the same content being interpreted differently by different types of application.

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

5.2. Resolving Relative References to Absolute Form

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

The base URI is established according to the rules of Section 5.1 and parsed into the four main components as described in Section 3. Note that only the scheme component is required to be present in the base URI; the other components may be empty or undefined. A component is undefined if its preceding separator does not appear in the URI reference; the path component is never undefined, though it may be empty. The base URI's query component is not used by the resolution algorithm and may be discarded.

For each URI reference, the following steps are performed in order:

1) The URI reference is parsed into the potential four components and fragment identifier, as described in Section 4.3.

2) If the path component is empty and the scheme, authority, and query components are undefined, then it is a reference to the current document and we are done. Otherwise, the reference URI's query and fragment components are defined as found (or not found) within the URI reference and not inherited from the base URI.

3) If the scheme component is defined, indicating that the reference starts with a scheme name, then the reference is interpreted as an absolute URI and we are done. Otherwise, the reference URI's scheme is inherited from the base URI's scheme component.

   Due to a loophole in prior specifications [RFC1630], some parsers allow the scheme name to be present in a relative URI if it is the same as the base URI scheme. Unfortunately, this can conflict with the correct parsing of non-hierarchical URI. For backwards compatibility, an implementation may work around such references by removing the scheme if it matches that of the base URI and the scheme is known to always use the <hier_part> syntax. The parser

can then continue with the steps below for the remainder of the reference components. Validating parsers should mark such a misformed relative reference as an error.

4) If the authority component is defined, then the reference is a network-path and we skip to step 7. Otherwise, the reference URI's authority is inherited from the base URI's authority component, which will also be undefined if the URI scheme does not use an authority component.

5) If the path component begins with a slash character ("/"), then the reference is an absolute-path and we skip to step 7.

6) If this step is reached, then we are resolving a relative-path reference. The relative path needs to be merged with the base URI's path. Although there are many ways to do this, we will describe a simple method using a separate string buffer.

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

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

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

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

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

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

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

h) The remaining buffer string is the reference URI's new path component.

7) The resulting URI components, including any inherited from the base URI, are recombined to give the absolute form of the URI reference. Using pseudocode, this would be

```
result = ""

if scheme is defined then
    append scheme to result
    append ":" to result

if authority is defined then
    append "//" to result
    append authority to result

append path to result

if query is defined then
    append "?" to result
    append query to result

if fragment is defined then
    append "#" to result
    append fragment to result

return result
```

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

The above algorithm is intended to provide an example by which the output of implementations can be tested -- implementation of the algorithm itself is not required. For example, some systems may find it more efficient to implement step 6 as a pair of segment stacks being merged, rather than as a series of string pattern replacements.

Note: Some WWW client applications will fail to separate the reference's query component from its path component before merging the base and reference paths in step 6 above. This may result in a loss of information if the query component contains the strings "/../" or "/./".

Resolution examples are provided in Appendix C.

6. URI Normalization and Equivalence

In many cases, different URI strings may actually identify the identical resource. For example, the host names used in URL are actually case insensitive, and the URL <http://www.XEROX.com> is equivalent to <http://www.xerox.com>. In general, the rules for equivalence and definition of a normal form, if any, are scheme dependent. When a scheme uses elements of the common syntax, it will also use the common syntax equivalence rules, namely that the scheme and hostname are case insensitive and a URL with an explicit ":port", where the port is the default for the scheme, is equivalent to one where the port is elided.

7. Security Considerations

A URI does not in itself pose a security threat. Users should beware that there is no general guarantee that a URL, which at one time located a given resource, will continue to do so. Nor is there any guarantee that a URL will not locate a different resource at some later point in time, due to the lack of any constraint on how a given authority apportions its namespace. Such a guarantee can only be obtained from the person(s) controlling that namespace and the resource in question. A specific URI scheme may include additional semantics, such as name persistence, if those semantics are required of all naming authorities for that scheme.

It is sometimes possible to construct a URL such that an attempt to perform a seemingly harmless, idempotent operation, such as the retrieval of an entity associated with the resource, will in fact cause a possibly damaging remote operation to occur. The unsafe URL is typically constructed by specifying a port number other than that reserved for the network protocol in question. The client unwittingly contacts a site that is in fact running a different protocol. The content of the URL contains instructions that, when interpreted according to this other protocol, cause an unexpected operation. An example has been the use of a gopher URL to cause an unintended or impersonating message to be sent via a SMTP server.

Caution should be used when using any URL that specifies a port number other than the default for the protocol, especially when it is a number within the reserved space.

Care should be taken when a URL contains escaped delimiters for a given protocol (for example, CR and LF characters for telnet protocols) that these are not unescaped before transmission. This might violate the protocol, but avoids the potential for such

characters to be used to simulate an extra operation or parameter in that protocol, which might lead to an unexpected and possibly harmful remote operation to be performed.

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

8. Acknowledgements

This document was derived from RFC 1738 [RFC1738] and RFC 1808 [RFC1808]; the acknowledgements in those specifications still apply. In addition, contributions by Gisle Aas, Martin Beet, Martin Duerst, Jim Gettys, Martijn Koster, Dave Kristol, Daniel LaLiberte, Foteos Macrides, James Marshall, Ryan Moats, Keith Moore, and Lauren Wood are gratefully acknowledged.

9. References

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

[RFC1630] Berners-Lee, T., "Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", RFC 1630, June 1994.

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

[RFC1866] Berners-Lee T., and D. Connolly, "HyperText Markup Language Specification -- 2.0", RFC 1866, November 1995.

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

[RFC822] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, August 1982.

[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.

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

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

[RFC1034] Mockapetris, P., "Domain Names - Concepts and Facilities", STD 13, RFC 1034, November 1987.

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

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

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

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

10. Authors' Addresses

Tim Berners-Lee
World Wide Web Consortium
MIT Laboratory for Computer Science, NE43-356
545 Technology Square
Cambridge, MA 02139

Fax: +1(617)258-8682
EMail: timbl@w3.org

Roy T. Fielding
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425

Fax: +1(949)824-1715
EMail: fielding@ics.uci.edu

Larry Masinter
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94034

Fax: +1(415)812-4333
EMail: masinter@parc.xerox.com

A. Collected BNF for URI

```
      URI-reference = [ absoluteURI | relativeURI ] [ "#" fragment ]
      absoluteURI   = scheme ":" ( hier_part | opaque_part )
      relativeURI   = ( net_path | abs_path | rel_path ) [ "?" query ]

      hier_part     = ( net_path | abs_path ) [ "?" query ]
      opaque_part   = uric_no_slash *uric

      uric_no_slash = unreserved | escaped | ";" | "?" | ":" | "@" |
                      "&" | "=" | "+" | "$" | ","

      net_path      = "//" authority [ abs_path ]
      abs_path      = "/"  path_segments
      rel_path      = rel_segment [ abs_path ]

      rel_segment   = 1*( unreserved | escaped |
                          ";" | "@" | "&" | "=" | "+" | "$" | "," )

      scheme        = alpha *( alpha | digit | "+" | "-" | "." )

      authority     = server | reg_name

      reg_name      = 1*( unreserved | escaped | "$" | "," |
                          ";" | ":" | "@" | "&" | "=" | "+" )

      server        = [ [ userinfo "@" ] hostport ]
      userinfo      = *( unreserved | escaped |
                         ";" | ":" | "&" | "=" | "+" | "$" | "," )

      hostport      = host [ ":" port ]
      host          = hostname | IPv4address
      hostname      = *( domainlabel "." ) toplabel [ "." ]
      domainlabel   = alphanum | alphanum *( alphanum | "-" ) alphanum
      toplabel      = alpha | alpha *( alphanum | "-" ) alphanum
      IPv4address   = 1*digit "." 1*digit "." 1*digit "." 1*digit
      port          = *digit

      path          = [ abs_path | opaque_part ]
      path_segments = segment *( "/" segment )
      segment       = *pchar *( ";" param )
      param         = *pchar
      pchar         = unreserved | escaped |
                      ":" | "@" | "&" | "=" | "+" | "$" | ","

      query         = *uric

      fragment      = *uric
```

```
uric          = reserved | unreserved | escaped
reserved      = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" |
                "$" | ","
unreserved    = alphanum | mark
mark          = "-" | "_" | "." | "!" | "~" | "*" | "'" |
                "(" | ")"

escaped       = "%" hex hex
hex           = digit | "A" | "B" | "C" | "D" | "E" | "F" |
                        "a" | "b" | "c" | "d" | "e" | "f"

alphanum      = alpha | digit
alpha         = lowalpha | upalpha

lowalpha = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
           "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
           "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
upalpha  = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
           "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
           "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
digit    = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
           "8" | "9"
```

B. Parsing a URI Reference with a Regular Expression

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

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

```
^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?
 12            3  4          5       6  7        8 9
```

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

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

results in the following subexpression matches:

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

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

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

and, going in the opposite direction, we can recreate a URI reference from its components using the algorithm in step 7 of Section 5.2.

C. Examples of Resolving Relative URI References

Within an object with a well-defined base URI of

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

the relative URI would be resolved as follows:

C.1. Normal Examples

```
g:h           =  g:h
g             =  http://a/b/c/g
./g           =  http://a/b/c/g
g/            =  http://a/b/c/g/
/g            =  http://a/g
//g           =  http://g
?y            =  http://a/b/c/?y
g?y           =  http://a/b/c/g?y
#s            =  (current document)#s
g#s           =  http://a/b/c/g#s
g?y#s         =  http://a/b/c/g?y#s
;x            =  http://a/b/c/;x
g;x           =  http://a/b/c/g;x
g;x?y#s       =  http://a/b/c/g;x?y#s
.             =  http://a/b/c/
./            =  http://a/b/c/
..            =  http://a/b/
../           =  http://a/b/
../g          =  http://a/b/g
../..         =  http://a/
../../        =  http://a/
../../g       =  http://a/g
```

C.2. Abnormal Examples

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

An empty reference refers to the start of the current document.

```
<>            =  (current document)
```

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

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

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

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

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

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

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

All client applications remove the query component from the base URI before resolving relative URI. However, some applications fail to separate the reference’s query and/or fragment components from a relative path before merging it with the base path. This error is rarely noticed, since typical usage of a fragment never includes the hierarchy ("/") character, and the query component is not normally used within relative references.

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

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

```
   http:g        =  http:g           ; for validating parsers
                 |  http://a/b/c/g   ; for backwards compatibility
```

D. Embedding the Base URI in HTML documents

It is useful to consider an example of how the base URI of a document can be embedded within the document's content. In this appendix, we describe how documents written in the Hypertext Markup Language (HTML) [RFC1866] can include an embedded base URI. This appendix does not form a part of the URI specification and should not be considered as anything more than a descriptive example.

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

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

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

```
<http://www.ics.uci.edu/Test/a/x>
```

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

E. Recommendations for Delimiting URI in Context

URI are often transmitted through formats that do not provide a clear context for their interpretation. For example, there are many occasions when URI are included in plain text; examples include text sent in electronic mail, USENET news messages, and, most importantly, printed on paper. In such cases, it is important to be able to delimit the URI from the rest of the text, and in particular from punctuation marks that might be mistaken for part of the URI.

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

http://test.com/

These wrappers do not form part of the URI.

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

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

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

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

The prefix "URL:" (with or without a trailing space) was recommended as a way to used to help distinguish a URL from other bracketed designators, although this is not common in practice.

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

For example, the text:

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

contains the URI references

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

F. Abbreviated URLs

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

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

or simply the DNS hostname on its own. Such references are primarily intended for human interpretation rather than machine, with the assumption that context-based heuristics are sufficient to complete the URL (e.g., most hostnames beginning with "www" are likely to have a URL prefix of "http://"). Although there is no standard set of heuristics for disambiguating abbreviated URL references, many client implementations allow them to be entered by the user and heuristically resolved. It should be noted that such heuristics may change over time, particularly when new URL schemes are introduced.

Since an abbreviated URL has the same syntax as a relative URL path, abbreviated URL references cannot be used in contexts where relative URLs are expected. This limits the use of abbreviated URLs to places where there is no defined base URL, such as dialog boxes and off-line advertisements.

G. Summary of Non-editorial Changes

G.1. Additions

Section 4 (URI References) was added to stem the confusion regarding "what is a URI" and how to describe fragment identifiers given that they are not part of the URI, but are part of the URI syntax and parsing concerns. In addition, it provides a reference definition for use by other IETF specifications (HTML, HTTP, etc.) that have previously attempted to redefine the URI syntax in order to account for the presence of fragment identifiers in URI references.

Section 2.4 was rewritten to clarify a number of misinterpretations and to leave room for fully internationalized URI.

Appendix F on abbreviated URLs was added to describe the shortened references often seen on television and magazine advertisements and explain why they are not used in other contexts.

G.2. Modifications from both RFC 1738 and RFC 1808

Changed to URI syntax instead of just URL.

Confusion regarding the terms "character encoding", the URI "character set", and the escaping of characters with %<hex><hex> equivalents has (hopefully) been reduced. Many of the BNF rule names regarding the character sets have been changed to more accurately describe their purpose and to encompass all "characters" rather than just US-ASCII octets. Unless otherwise noted here, these modifications do not affect the URI syntax.

Both RFC 1738 and RFC 1808 refer to the "reserved" set of characters as if URI-interpreting software were limited to a single set of characters with a reserved purpose (i.e., as meaning something other than the data to which the characters correspond), and that this set was fixed by the URI scheme. However, this has not been true in practice; any character that is interpreted differently when it is escaped is, in effect, reserved. Furthermore, the interpreting engine on a HTTP server is often dependent on the resource, not just the URI scheme. The description of reserved characters has been changed accordingly.

The plus "+", dollar "$", and comma "," characters have been added to those in the "reserved" set, since they are treated as reserved within the query component.

The tilde "~" character was added to those in the "unreserved" set, since it is extensively used on the Internet in spite of the difficulty to transcribe it with some keyboards.

The syntax for URI scheme has been changed to require that all schemes begin with an alpha character.

The "user:password" form in the previous BNF was changed to a "userinfo" token, and the possibility that it might be "user:password" made scheme specific. In particular, the use of passwords in the clear is not even suggested by the syntax.

The question-mark "?" character was removed from the set of allowed characters for the userinfo in the authority component, since testing showed that many applications treat it as reserved for separating the query component from the rest of the URI.

The semicolon ";" character was added to those stated as being reserved within the authority component, since several new schemes are using it as a separator within userinfo to indicate the type of user authentication.

RFC 1738 specified that the path was separated from the authority portion of a URI by a slash. RFC 1808 followed suit, but with a fudge of carrying around the separator as a "prefix" in order to describe the parsing algorithm. RFC 1630 never had this problem, since it considered the slash to be part of the path. In writing this specification, it was found to be impossible to accurately describe and retain the difference between the two URI

<foo:/bar> and <foo:bar>

without either considering the slash to be part of the path (as corresponds to actual practice) or creating a separate component just to hold that slash. We chose the former.

G.3. Modifications from RFC 1738

The definition of specific URL schemes and their scheme-specific syntax and semantics has been moved to separate documents.

The URL host was defined as a fully-qualified domain name. However, many URLs are used without fully-qualified domain names (in contexts for which the full qualification is not necessary), without any host (as in some file URLs), or with a host of "localhost".

The URL port is now *digit instead of 1*digit, since systems are expected to handle the case where the ":" separator between host and port is supplied without a port.

The recommendations for delimiting URI in context (Appendix E) have been adjusted to reflect current practice.

G.4. Modifications from RFC 1808

RFC 1808 (Section 4) defined an empty URL reference (a reference containing nothing aside from the fragment identifier) as being a reference to the base URL. Unfortunately, that definition could be interpreted, upon selection of such a reference, as a new retrieval action on that resource. Since the normal intent of such references is for the user agent to change its view of the current document to the beginning of the specified fragment within that document, not to make an additional request of the resource, a description of how to correctly interpret an empty reference has been added in Section 4.

The description of the mythical Base header field has been replaced with a reference to the Content-Location header field defined by MHTML [RFC2110].

RFC 1808 described various schemes as either having or not having the properties of the generic URI syntax. However, the only requirement is that the particular document containing the relative references have a base URI that abides by the generic URI syntax, regardless of the URI scheme, so the associated description has been updated to reflect that.

The BNF term <net_loc> has been replaced with <authority>, since the latter more accurately describes its use and purpose. Likewise, the authority is no longer restricted to the IP server syntax.

Extensive testing of current client applications demonstrated that the majority of deployed systems do not use the ";" character to indicate trailing parameter information, and that the presence of a semicolon in a path segment does not affect the relative parsing of that segment. Therefore, parameters have been removed as a separate component and may now appear in any path segment. Their influence has been removed from the algorithm for resolving a relative URI reference. The resolution examples in Appendix C have been modified to reflect this change.

Implementations are now allowed to work around misformed relative references that are prefixed by the same scheme as the base URI, but only for schemes known to use the <hier_part> syntax.

H. Full Copyright Statement