Nested Uniform Resource Identifiers

Manuel Urueña and David Larrabeiti

Department of Telematic Engineering
Universidad Carlos III de Madrid
E-28911 Leganés, Madrid (Spain)
Email: {muruenya,dlla}@it.uc3m.es
Telephone: (+34) 91 624 87 95
Fax: (+34) 91 624 87 49

Abstract—Uniform Resource Identifiers (URIs) are very popular among non-technical people, to identify services and users in Internet, because they hide the underlying complexity of IP addresses and port numbers with a simple syntax. However, as currently defined, URIs are not extensible enough to support other name resolution mechanisms than the Domain Name System (DNS), nor newer transport protocols. This paper defines a backward-compatible syntax for URIs, that allows the location part of any URI to be defined with a URI itself. This nested URI syntax is more flexible, as it makes it possible for current applications to employ dynamic Service Discovery protocols, and to support multiple transport protocols.

I. INTRODUCTION

From the average user point of view, Uniform Resource Identifiers (URIs) [1] are the preferred “addresses” of the Internet, as their simple syntax is able to hide the underlying technical complexity such as IP addresses, transport protocol ports or DNS queries. Moreover, web sites URLs and mail addresses have become well-known terms, even for non-technical people. Therefore many network protocols are defining its own URI schemes, because of their growing popularity.

Although nowadays “Uniform Resource Identifier” is the recommended term for any of their uses [2], many people still employ alternative terms like Uniform Resource Names (URN) or Uniform Resource Locators (URL), to specify whether the URI refers to an abstract identifier, like an ISBN book code (URN), or to an addressable service, like a web page (URL). Therefore, although this paper always employs the generic term URI, its main applications are related to URIs referring to locations; thus the term URL could be used instead.

A URI is composed by a hierarchical sequence of components, whose generic syntax [1] is shown in figure 1. In the URL/URI of a client-server application: the scheme identifies the application protocol to be employed (e.g. http), the authority part defines the location of the peer entity (e.g. www.example.com:80), while the optional path, query and fragment sections usually refer to the piece of information to be handled by the protocol (e.g. index.html#top).

Therefore, in a TCP/IP network, the authority section must identify the transport address of the target server, which is composed by: an IP network address and a transport protocol port number. However, most URIs do not include the IP

address explicitly, but the DNS hostname of the server. Also, the service access port is not usually specified, as most of the application protocols have a default port number.

URI syntax allows these and other simplifications to create shorter strings, as URIs were designed to be easily read and written by human beings, while remaining flexible enough to be employed in multiple contexts and to identify many types of resources.

When URIs were initially specified, they allowed two kinds of location identifiers: IPv4 addresses, and DNS domain names. Later on, IPv6 addresses were included. This made the authority section syntax a little more complex, because a square bracket delimiter has to be added in order to differentiate IPv6 hexadecimal strings from IPv4 addresses or DNS names. Figure 2 shows three sample URIs that specify resource locations by: a DNS hostname ([www.example.com]), an IPv4 address (10.117.139.166), an IPv6 address (2001:DB8::2c0:9fff:fe18:31d4), and a port number (80).

Therefore, although current generic URI syntax specifica-
tion allows future IP addresses to be defined, at the present time a resource location can be only specified by one of these three mechanisms. Thus, services have to be identified by the server they reside in (that is, with its hostname or IP address), and the only name-resolution mechanism available for URIs is the Domain Name System (DNS)\(^1\).

However, nowadays there are many other mechanisms to locate a resource. For example, load balancing frameworks such as Reliable Server Pooling (Rserpool) \([3]\), or service discovery protocols like Service Location Protocol (SLP) \([4]\), Universal Plug’n’Play (UPnP) \([5]\), eXtensible Service Discovery Framework (XSDF) \([6]\), or DNS-based by querying SRV Resource Records \([7]\).

All these techniques could be also seen as name-resolution mechanisms as, at the end, they return a valid IP address of the server where the service is running. Although, unlike DNS, they do not use hostnames as the search key but an abstract service identifier (e.g. “printer”). Most of them are also more dynamic than DNS, thus they are able to locate available services at any unknown network.

Therefore, as the current URI syntax depends on DNS as the only resolution mechanism, current applications can neither employ any of these service discovery protocols, nor any other that could be defined at the future, unless applications implement them by themselves, or an alternative mechanism based on URIs is in place to choose the name-resolution mechanism to be employed.

Another limitation of URIs is in the identification of the transport layer to be used, as only the port number can be specified. This might be considered a minor limitation nowadays because the Transmission Control Protocol (TCP) is, by far, the most popular transport protocol, and many network applications, like web servers or mail agents, only support TCP. However, applications may employ several transport protocols, and many of the first IP services, including DNS, were defined to support both TCP and UDP.

Moreover, there are several new transport protocols that could replace their 20-years-old counterparts, such as the Datagram Congestion Control Protocol (DCCP) \([8]\) or the Stream Control Transmission Protocol (SCTP) \([9]\), which provides many advanced features like multiple channels, multi-homing support, dynamic address management, etc. Therefore, the possibility of specifying a transport protocol inside a URI seems to be an important tool for the future development of Internet services and protocol architectures.

Of course, non-technical users are not interested in neither which transport protocol nor name-resolution mechanism are employed while surfing the web. However, overcoming these limitations could be very useful for researchers, migration scenarios and network debugging, whereas average-users would remain unaware of these fully detailed URIs if they are embedded inside web pages or applications.

\(^{1}\)Actually, hostnames may be resolved by other means like local files or NIS, although they must be DNS names

II. Possible Solutions to URI Limitations

Currently, transport and name resolution are implicit methods associated to a URI when the URI Scheme \([10]\) is defined for that application. Therefore, a possible future solution to deal with these limitations could be to define a new URI scheme whenever an application is updated to support a new transport protocol or an alternative name-resolution mechanism (i.e. a similar approach to the \texttt{https} scheme for HTTP over SSL).

However, this alternative has several drawbacks:

- **Applications should be modified to recognize the new scheme.** Although this could be not regarded as an issue because the application has to be modified anyway to support the new transport protocol, this could be not true for alternative name-resolution mechanisms, or transport protocols that, for example, are capable of emulating the API of TCP.
- **Combinatorial explosion.** When an application needs to support several service discovery protocols for name-resolution AND different transport protocols, it should have to define and handle all the possible scheme combinations, like: \texttt{lpr-slp-tcp, lpr-upnp-tcp, lpr-srv-tcp, lpr-slp-sctp}...

As the name-resolution mechanism and transport protocol to be used are orthogonal choices, it makes sense to allow each one to be made separately to avoid the latter problem. Therefore, as a second alternative solution, it could be possible to define standard resolver and transport protocol options for the URI’s \texttt{query} section, which would lead to URIs like: \texttt{lpr://printer:515/queue?resolver=SLP\&proto=SCTP}...

However, this second mechanism still requires applications to be updated each time new query options are to be supported, but also to re-define the URI schemes that do not include a \texttt{query} section, because it is an optional part of the current generic URI syntax.

III. Nested URIs

The previous possible solutions would not require the current URI specification to be changed, as they are based on the extensible sections of the general URI syntax. However, the simplest way to handle these limitations, related to the transport address identifiers, seems to be the \texttt{authority} section itself; in particular the \texttt{host} and \texttt{port} components.

Unfortunately, the \texttt{port} section is not extensible at all, while the \texttt{host} part can only be expanded to support alternative IP addresses between square brackets, as IPv6 hexadecimal ones, or future formats by defining a “\texttt{v?}.” prefix \([1]\).

Therefore, it is necessary to modify the current URI syntax definition in order to support other name-resolution mechanisms or to include transport protocol identifiers at the \texttt{authority} section.

However, it is desirable that the new \texttt{authority} syntax should be compatible with the other sections and even with the previous one. Also, it should be easily read and written by non-technical users, as common URIs are. Moreover, the new
syntax should be extensible in order to support future name-
resolution mechanisms and transport protocols.

Then, Why not to employ URIs themselves? They are simple,
its syntax is well known and, with the proper encapsulation
mechanism, they can be compatible with previous syntax as,
by definition, they do not contain reserved characters.

For example, the following URIs identify IPv4, IPv6, and
even IPX addresses, a DNS hostname [11], a SLP service [12],
and transport protocol ports for TCP and SCTP, with an easily
readable and extensible representation:

```
ipv4:10.117.139.166
ipv6:2001:DB8::2c0:9fff:fe18:31d4
ipx:00000001:00081A0D01C2
dns:www.example.com
service:printer
tcp:80
sctp:80
```

When needed, these location URIs may be used to define the
host and/or port sections of a URI, although enclosed between
square brackets ("[", "]") in order to identify them as nested
URIs (Fig. 3).

Note that this definition is not fully recursive but it only
allows one nesting level to ease URI parsing. Therefore
location URIs must not include other location URIs inside.

As described in the next section, the usage of square
brackets separators allows current applications to support the
host part of nested-URI syntax without being updated, as
URI parsing libraries usually do not process square bracket-
enclosed IPv6 addresses but return them as plain text strings.
Thus, nested-URIs would be successfully processed by most
of the current URI parsing routines.

This syntax is also backward-compatible with the standard
one [1], although nested-URI parsing code should take into
account that IPv6 addresses are also enclosed between square
brackets. As none of the standard URI schemes [10] can be
mistaken as an IPv6 address, the simplest way to support
legacy IPv6 addresses is to fallback to an IPv6 address parser
when no nested-URI schemes are found.

There are other alternatives to obtain full backwards-
compatibility between nested-URIs and the standard syntax.
For example, it could be possible to use a different
delimiter pair (e.g. "[",""), or the IPvFuture format (e.g.
[v0:service:printer]) to enclose nested-URIs, although
these solutions would lead to a more complex URI syntax, and
also require URI parsing routines of current applications to be
updated.

However, the above discussion only refers to nested-URIs
in the host part, as supporting nested-URIs in the port section
requires URI parsing libraries to be updated.

As an example of the new syntax, figure 4 shows the same
URIs of figure 2, this time employing nested URI locations.
However former URIs are still valid with the new syntax, and
even preferred, as they are shorter to type.

The real gain of the nested URI syntax comes from the
enhanced expressivity. For example, figure 5 shows three URIs
referring to a printer. The first one is almost a regular URI
but specifies that the SCTP transport protocol should be used,
instead of TCP.

The second one looks like the first example of figure 4, but it
specifies that the DNS server should be employed to discover
printers at the example.com domain, instead of specifying
the hostname of the printer. This is done by querying the
SRV Resource Record [7], that contains a list of printer’s
hostnames, that are associated with a weight in order to
perform load balancing among them.

The last URI employs SLP [4] in order to discover printers
at the local LAN. But, unlike the DNS SRV method, SLP
is a dynamic protocol, thus only the available printers will
be returned. Even more, SLP Service Requests could include
search predicates to filter Services. Therefore, this URI would
choose an available color printer.

Figure 6 shows more advanced uses of nested URIs. For
example, the IPv6 URI could specify the scope of the address,
that in the case of link local addresses is the interface identifier
(eth0).

```
http://[dns:www.example.com]:[tcp:80]/index.html
http://[ipv4:10.117.139.166]:[tcp:80]/index.html
http://[ipv6:2001:DB8::2c0:9fff:fe18:31d4]:[tcp:80]/
index.html
telnet://[ipv6:fe80::2c0:9fff:fe18:31d4#eth0]
http://[dns/www.example.com:type=AAAA]/index.html
ftp://[dns://192.168.1.31:53/_drivers._sctp.example.com?
type=SRV]:[sctp:21]/acmelaser2000
```

Fig. 4. Simple nested URI examples

Fig. 6. Advanced examples with nested URIs
DNS hostname-resolution could also benefit from this nested-URI syntax, as it can include options for the DNS query. For example to choose the type of Resource Record to be returned. Therefore, the second location URI does force the use of IPv6 addresses only. The last URI is left as an exercise for the reader.

Another big advantage of the nested URI solution, compared to the other alternatives is that it does not require applications to be modified in order to support it, as described in the next section.

IV. IMPLEMENTATION DESIGN

In order to discuss about the implementation details to support nested URIs, it is necessary to understand how applications employ URIs to access network services.

First network applications heavily depended on IPv4 addresses and DNS semantics, like the gethostbyname() function, that was a direct mapping of a DNS query. However when IPv6 started being defined, the application level was also analyzed [13] in order to ease the development of IP-independent applications. As a result of this review process, the gethostbyname() function was declared obsolete and was replaced by getaddrinfo() (Fig. 8), that hides the details of name-resolution and IP addresses into protocol-agnostic structures. This function is also much more service-oriented, as it also handles transport port resolution.

Figure 10 shows the recommended [13] C source code for developing an IP-independent TCP client application, by means of the getaddrinfo() function.

For example, if this application supports a URI, like lpr://printer1.example.com:515, to select the target server, the connection process should look like the one in figure 7:

1) The application parses the input URI to identify the host and port parts, and the rest of the URI components, that might control the application’s behavior.
2) The location information obtained at the previous step (SERVER_NODE = "printer1.example.com", SERVICE = "515") is passed to the getaddrinfo() function to be resolved, specifying that any family of IP addresses could be accepted (AF_UNSPEC), but the recommended transport protocol should be connection-oriented, like TCP (SOCK_STREAM).
3) As the SERVER_NODE string is neither an IPv4 nor an IPv6 address, then the DNS resolver at the client machine queries its recursive DNS server.
4) The DNS server returns the AAAA and A Resource Records (10.117.139.166) associated to the specified hostname.
5) The getaddrinfo() call returns an addrinfo structure containing all the transport addresses (sockaddr), built from the specified port number and IP addresses obtained from the DNS query.
6) Finally, a socket is created and connected to the first server transport address, in order to run the application-level protocol session over the the established TCP connection.

Therefore, the getaddrinfo() library is able to handle all the URI location part, as it returns all the information needed by the applications to establish the transport connection with the peer server. Thus, application code does not need to
understand whether the URI refers to a DNS hostname or a literal IP address, and it just passes the opaque string down to the name-resolver library.

This behavior is the key to seamlessly support nested URIs without the need for applications to be modified. Instead, it is only necessary to update the getaddrinfo() library to understand nested URI syntax and to employ the appropriate name-resolution mechanism to deal with it.

For example, figure 9 shows an enhanced system that executes the same application of the previous example (Fig. 10), fed with the following nested-URI:

```
lpr://[service:printer]
```

As done before, the application extracts the `host` ("[service:printer]"), and `port` (NULL) parts of the input URI, and it passes them to the getaddrinfo() call, as if it were a conventional URI.

As the nested URIs are delimited by square brackets, current URI parser libraries should identify the `host` section simply as an IPv6 literal address. However, a nested `port` URI could be treated as an error, and therefore some URI parsing libraries might be updated to support nested-port URIs.

The name-address resolution library does recognize the nested-URI (or just an IP address or hostname, and thus previous URI does also work) and, depending on the initial scheme ("service:"), the location URI is resolved by the appropriate mechanism.

In this case, a SLP User Agent [4] should be employed, although if the specified scheme does not have an associated name resolution mechanism an `EAI_NONAME` error code should be returned by the getaddrinfo() call.

The SLP User Agent sends a `ServiceRequest` to the Directory Agent of the local network (or, if not present, to all the Service Agents by means of multicast), in order to discover all the printers in the LAN. As the Service Agents of the printers have previously registered their service information at the central Directory Agent, the DA is able to reply with the list of all the `available` printers back to the User Agent.

This is an important difference between DNS-based service discovery and a dynamic protocol like SLP, because a static resolution mechanism like DNS returns all the configured printers, no matter they are disabled or not; hence the application connection could fail later. SLP, on the other hand, only returns the available printers, according to the updates received periodically.

When the SLP nested-URI is fully resolved, the resulting transport addresses (i.e., including the port number that is also learnt via SLP) are returned to the application, that is able to access the selected printer.

This mechanism also enables application protocols to easily support different transport protocols. For example, in this case the transport protocol to be employed could be SCTP, even if the application expects a TCP connection, because SCTP is
able to mimic [14] the TCP socket API functionalities (i.e., this reasoning could also be applied to DCCP/UDP), while offering advanced capabilities as multi-homing support.

V. CONCLUSION
Although Uniform Resource Identifier (URI) usage has become very popular among protocol designers because of its flexibility, and also for non-technical users because of its readability, its current syntax [1] has two important limitations:

1) Resources can be located only by the IP address or the hostname of the server providing it.
2) URIs may include a port number, but there is no way to identify the transport protocol that should be employed to access a given service.

Therefore, URI usage forces application protocols to employ the DNS as the only name-resolution mechanism, and to support just one transport protocol. For this reason Service Discovery protocols like SLP [4], or new transport protocols like SCTP [9] cannot be easily deployed, even when they provide many advantages over their 20-year old counterparts: DNS and TCP.

This work has defined a backwards-compatible generic URI syntax, that allows users to choose the name-resolution mechanism and the transport protocol that should be used, by specifying them with nested URIs. This mechanism has many advantages over other possible alternatives, as nested URIs are easily readable, they do not require applications to support new schemes or options, and it even enhances typical URI usage to become even more flexible. For example, this additional flexibility may allow name-resolution mechanisms to be parametrized (e.g. to force a DNS query to return AAAA Resource Records, or to support IPv6 address scopes).

This paper has also studied the requisites to support nested-URIs in existing systems and applications. The conclusion is that IP-independent applications that employ the getaddrinfo() call could support nested-URIs without being modified, as they are unaware of the name-resolution mechanism employed to obtain the server transport addresses.

Therefore, it should be enough to upgrade the name-address resolver libraries and some URI parser routines, in order to enable nested-URI support to any existing application that follows current recommended practices. This way, next-generation applications will be able to benefit from state-of-the-art transport protocols, as well as load balancing and dynamic service discovery protocols.

ACKNOWLEDGMENT
The authors would like to thank Carlos Jesús Bernardos, Pablo Serrano, Norberto Fernández, Pablo Basanta and Ignacio Soto for their valuable comments and support.

This work was partially funded by the Spanish MEC under project CAPITAL TEC2004-05622-C04-03/TCM. This paper reflects the view of the authors and not necessarily the view of the project.

REFERENCES