James Kempf
Erik Guttman

Internet Draft
Document: draft-kempf-svrloc-rfc2614bis-00.txt
Expires: August 2002
Feburary 2002

An API for Service Location

Status of this Memo

Abstract

The Service Location Protocol (SLP) provides a way for clients to dynamically discovery network services. This document describes a standardized API for SLP in the C language. In addition, standardized file formats for configuration and serialized registrations are defined. This document defines a new API for SLP that supercedes the definition in RFC 2614.

Table of Contents

1.0 Introduction

The Service Location API is designed for standardized access to the Service Location Protocol (SLP) through a C language interface. The API facilitates writing portable client and service programs. In addition, standardized formats for configuration files and for serialized registration files are presented. These files allow system administrators to configure network parameters, to register legacy services that have not been SLP-enabled, and to portably exchange configuration and registration files. This document supercedes the SLP API definition in RFC 2614 [1] and corresponds to the protocol definition described in [8].

1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [2].

Please see [8] for specific SLP protocol-related terms.

SA Server

Many operating system platforms only allow a single process to listen on a particular port number for TCP. Since general purpose SAs are required to listen on TCP for SLP requests, implementations of the SLP supporting multiple SAs on such platforms need to arrange for a single process to do the listening. The advertising SAs communicate with that process through another mechanism, described here in Section XXX. The single listening process is called an SA server. SA servers share many characteristics with DAs, but they are not the same.

2.0 File Formats

This section describes the configuration and serialized registration file formats. Both files are defined in the UTF-8 character set [4], and they must not include a Byte Order Mark (BOM) at the beginning, to maximize compatibility with US-ASCII. The rules governing attribute tags and values in serialized registration files and configuration files are exactly the same as those for the wire format described in [8]. Attribute tags and string values require SLP reserved characters to be escaped. The SLP reserved characters are '(', ')', ',', '\', '!', '<', '=', '>', '~' and control

characters (characters with UTF codes less than 0x0020 and the
character 0x007f, which is US-ASCII DEL). The escapes are formed
exactly as for the wire protocol, i.e. a backslash followed by two
hex digits representing the character. For example, the escape for

',' is '\2c'. In addition, the characters "\n", "\r", "\t", and '_' are prohibited from attribute tags by the SLP wire syntax grammar [8]. Other characters may be escaped, and are processed into the corresponding characters upon input, exactly as for reserved characters.

In file encodings for attribute values, strings beginning with "\ff", an encoding for a nonUTF-8 character, are treated as opaques. Exactly as in the wire protocol, syntactically correct opaque encodings consist of a string beginning with "\ff" and containing *only* escaped characters that are transformed to bytes. Such strings are only syntactically correct as attribute values. In a string beginning with "\ff", if any characters are not escaped, it is a syntax error.

Escaped characters in URLs use the URL escape convention [3].

Property names and values in the configuration file have a few reserved characters that are involved in file's lexical definition, in addition to those involving attributes described above, for those property values that contain attribute list definitions. The characters '.' and '=' are reserved in property names and must be escaped. The characters ',', '(', and ')' are reserved in all property values, not just attribute list definitions, and must be escaped. In addition, scope names in the net.slp.configuredScopes property use the SLP wire format escape convention for SLP reserved characters. This simplifies implementation, since the same code can be used to unescape scope names as is used for formatting wire messages.

On platforms that only support US-ASCII and not UTF-8, the upper bit of bytes incoming from the configuration and registration files determines whether the character is US-ASCII or not. According to the standard UTF-8 encoding, the upper bit is zero if the character is US-ASCII and one if the character is multibyte and thus not US-ASCII. Platforms without intrinsic UTF-8 support are required to parse the multibyte character and store it in an appropriate internal format. Support for UTF-8 is required to implement the SLP protocol (see [8]), and can therefore be used in file processing as well.

The location and name of the configuration file is system-dependent, but implementations of the API are encouraged to locate it together with other configuration files and name it consistently.

2.1 Configuration File Format

The configuration file format consists of a newline delimited list of zero or more property definitions. Each property definition corresponds to a particular configurable SLP, network, or other

parameter in one or more of the three SLP agents. The file format
grammar in ABNF [6] syntax is:

```
         config-file   =  line-list
         line-list     =  line / line line-list
         line          =  property-line / comment-line
         comment-line  =  ( "#" / ";" ) 1*allchar newline
         property-line =  property newline
         property      =  tag "=" value-list
         tag           =  prop / prop "." tag
         prop          =  1*tagchar
         list          =  value / value "," list
         value         =  int / bool / attribute /
                          string / addr
         int           =  1*DIGIT
         bool          =  "true" / "false" / "TRUE" / "FALSE"
         newline       =  CR / ( CRLF )
         string        =  1*stringchar
         attribute     =  ; see the definition of attribute
                          ; list in Section 4.3.6 of [8].
         addr          =  fqdn / hostnumber
         fqdn          =  ALPHA / ALPHA *[ anum / "-" ] anum
         anum          =  ALPHA / DIGIT
         hostnumber    =  1*3DIGIT 3("." 1*3DIGIT)
         tagchar       =  DIGIT / ALPHA / tother / escape
         tother        =  %x21-%x2d / %x2f /
                          %x3a / %x3c-%x40 /
                          %x5b-%x60 / %7b-%7e
                          ; i.e., all characters except `.',
                          ; and `='.
         stringchar    =  DIGIT / ALPHA / sother / escape
         sother        =  %x21-%x29 / %x2a-%x2b /
                          %x2d-%x2f / %x3a-%x40 /
                          %x5b-%x60 / %7b-%7e
                          ; i.e., all characters except `,'
         allchar       =  DIGIT / ALPHA / HTAB / SP
         escape        =  "\" HEXDIG HEXDIG
                          ; Used for reserved characters
```

All properties can be changed through the SLPSetProperty() API function. However, changing certain properties has no effect on further execution in the API library, since these properties are only involved in conveying preconfigured information to the API library on startup and are not used afterwards. These properties are net.slp.configuredScopes, net.slp.configuredDAAddresses, and net.slp.enableBroadcast.

On multi-homed hosts, it may be necessary to have different network configuration properties for different interfaces. The net.slp.interfaces property indicates which network interfaces are SLP enabled. An API library implementation may support configuration customization on a per network interface basis by allowing the interface IP address or host name to be appended to the property name. In that case, the values of the property are only used for that particular interface, the generic property (or defaults if no generic property is set) applies to all others.

For example, if a configuration file has the following properties:

```
   net.slp.interfaces=125.196.42.41,125.196.42.42,125.196.42.43
   net.slp.multicastTTL.125.196.42.42=1
```

then the network interface on subnet 42 is restricted to a TTL of 1, while the interfaces on the other subnets have the default multicast TTL, 255.

The following subsections describe an area and its properties.

2.1.1 DA configuration

The following properties are used for DA configuration. They are ignored if the host is not configured as a DA:

```
   net.slp.isDA
      Type:    Boolean
      Default: FALSE
      Use:     A Boolean configuring the SLP server to act as a DA.
               If TRUE, run as a DA.

    net.slp.DAHeartBeat
      Type:    Unsigned 32 bit integer
      Default: 10800 seconds (3 hours)
      Use:     The number of seconds between transmission of
               unsolicited DAAdverts by the DA. This property
               corresponds to the protocol specification parameter
               CONFIG_DA_BEAT [8].

    net.slp.DAAttributes
      Type:    List of Attribute
      Default: Null
      Use:     A list of parenthesized attribute/value list pairs
               that the DA must advertise in DAAdverts.
```

2.1.2 Preconfiguration

The following properties convey statically configured or DHCP-configured information to all agents. Changing these properties using SLPSetProperty() has no effect on execution.

```
    net.slp.configuredScopes
      Type:    List of String
      Default: Null
      Use:     A list of statically configured or DHCP-configured
               scopes.

   net.slp.configuredDAAddresses
      Type:    List of Address
      Default: Null
      Use:     A list of statically configured or DHCP-configured DA
               IP addresses or DNS-resolvable host names.

   net.slp.enableBroadcast
```

```
      Type:    Boolean
      Default: FALSE
      Use:     If TRUE, enable all SLP agents to use broadcast
               instead of multicast, and disable multicast.
```

2.1.3 Tracing and Logging

The following properties are used to control tracing and logging of error and warning messages.

```
net.slp.traceDATraffic
   Type:    Boolean
   Default: FALSE
   Use:     If TRUE, print log messages about traffic to DAs.

net.slp.traceMsg
   Type:    Boolean
   Default: FALSE
   Use:     If TRUE, print log messages of all incoming and
            outgoing SLP messages.

net.slp.traceDrop
   Type:    Boolean
   Default: FALSE
   Use:     If TRUE, print log messages when a SLP message is
            dropped for any reason.

net.slp.traceReg
   Type:   Boolean
   Default FALSE
   Use:    If TRUE, dump all registred services when a
            registration or deregistration occurs.
```

2.1.4 Serialized Proxy Registrations

The following property controls processing of serialized registrations.

```
net.slp.serializedRegURL
   Type:    String
   Default: Null
   Use:     A URL pointing to a document containing serialized
            registrations that should be processed when the DA or
            SA server starts up.
```

2.1.5 Network Configuration Properties

The properties in this section allow various network configuration properties to be set.

```
net.slp.multicastTTL
   Type:    Positive integer less than or equal to 255
   Default: 255
   Use:     Multicast TTL.

net.slp.DAActiveDiscoveryInterval
```

```
      Type:    Unsigned 16 bit integer
      Default: 900
      Use:     The number of seconds between DA active discovery
               queries. The queries may be done periodically or in
               response to a particular SLP operation. This property
               corresponds to the protocol specification parameter
               CONFIG_DA_FIND [8]. If the property is set to zero,
               active discovery is turned off.

   net.slp.passiveDADetection
      Type:    Boolean
      Default: TRUE
      Use:     If FALSE, ignore any unsolicited DAAdverts that are
               received.

   net.slp.multicastMaximumWait
      Type:   Positive 32 bit integer.
      Default 15000 ms (15 sec.)
      Use:    Maximum number of milliseconds to multicast a request
               before giving up. This property corresponds to the
               CONFIG_MC_MAX parameter in the protocol specification
               [8].

   net.slp.multicastTimeouts
      Type:    List of positive 32 bit integer
      Default: 3000,3000,3000,3000,3000
      Use:     The timeouts, in milliseconds, to use for multiple
               attempts at multicast for UA requests. Each value
               specifies the time to wait before sending the next
               request, or until nothing new has been learned from
               two successive requests. The sum should equal
               net.slp.multicastMaximumWait.

   net.slp.DADiscoveryTimeouts
      Type:    List of positive 32 bit integer
      Default: 2000,2000,2000,2000,3000,4000
      Use:     The timeouts, in milliseconds, to use for multiple
               attempts at multicast for active DA discovery. Each
               value specifies the time to wait before sending the
               next request, or until nothing new has been learned
               from two successive requests. The sum should equal
               net.slp.multicastMaximumWait.

   net.slp.datagramTimeouts
      Type:    List of positive 32 bit integer
      Default: 3000,3000,3000,3000,3000
      Use:     The timeouts, in milliseconds, to use for
               retransmitting unicast UDP requests. The nth value
               gives the time to block waiting for a reply on the
               nth try to contact the DA. The sum of these values
               should equal the protocol specification property
               CONFIG_RETRY_MAX [8].

   net.slp.randomWaitBound
      Type:    Positive 32 bit integer
      Default: 1000 ms (1 sec.)
      Use:     The maximum value in milliseconds for all random wait
               parameters. This value corresponds to the protocol
               specification parameters CONFIG_START_WAIT,
               CONFIG_REG_PASSIVE, and CONFIG_REG_ACTIVE [8].
```

```
   net.slp.MTU
      Type:    Positive 16 bit integer
      Default: 1500
      Use:     Maximum datagram size for an SLP agent to send, and
               includes IP and UDP or TCP headers.

   net.slp.interfaces
      Type:    List of Address
      Default: System Default
      Use:     List of IP addresses for interfaces on the host on
               which the DA or SA server should listen on port 427
               for multicast, unicast UDP, and TCP messages.
```

2.1.6 SA Configuration

The following properties are used for SA or SA server configuration.

```
   net.slp.SAAttributes
      Type:    List of Attribute
      Default: "(service-type=" <list of service types> ")"
      Use:     A list of attribute definitions advertised by the SA
               in an SAAdvert. The list must contain the "service-
               type" attribute with value equal to all service types
               advertised by the SA.
```

2.1.7 UA Configuration

The following properties are used by the UA for configuration. They can be set dynamically through SLPSetProperty() to alter API library execution.

```
   net.slp.locale
      Type:    RFC 1766 Language Tag [7]
      Default: "en"
      Use:     The default locale used for language tags in SLP
               messages. This property is also used for SA and DA
               configuration.

   net.slp.maxResults
      Type:    Nonnegative 32 bit integer, and -1
      Default: -1
      Use:     The maximum number of results to report. A value of
               -1 indicates that all requests should be reported.

   net.slp.typeHint
      Type:    List of string
      Default: Null
      Use:     A list of service type names that are used when
               performing SA discovery

   net.slp.enableUnicastSARequest
```

```
      Type:    Boolean
      Default: FALSE
      Use:     If TRUE, the UA uses unicast to contact SAs directly
               rather than multicast, and does not use DAs even if
               DAs are available
```

2.2 Serialized Registration File

The serialized registration file contains a group of proxy registrations that a DA or SA server performs when it starts up. These registrations are primarily for older service programs that do not internally support SLP and cannot be converted, and for portably exchanging registrations between SLP implementations. The character encoding of the registrations is UTF-8.

The syntax of the serialized registration file, in ABNF format [6], is as follows:

```
   ser-file      =  reg-list
   reg-list      =  reg / reg reg-list
   reg           =  creg / ser-reg
   creg          =  comment-line ser-reg
   comment-line  =  ( "#" / ";" ) 1*allchar newline
   ser-reg       =  url-props [attr-list] newline
   url-props     =  surl "," lang "," ltime [ "," type ] newline
   surl          =  ;The registration's URL. See
                    ; [9] for syntax.
   lang          =  1*8ALPHA [ "-" 1*8ALPHA ]
                    ;RFC 1766 Language Tag see [7].
   ltime         =  1*5DIGIT
                    ; A positive 16-bit integer
                    ; giving the lifetime
                    ; of the registration.
   type          =  ; The service type name, see [8]
                    ; and [9] for syntax.
   attr-list     =  attr-def / attr-def attr-list
   attr-def      =  ( attr / keyword ) newline
   keyword       =  attr-id
   attr          =  attr-id "=" attr-val-list
   attr-id       =  ;Attribute id, see [8] for syntax.
   attr-val-list =  attr-val / attr-val "," attr-val-list
   attr-val      =  ;Attribute value, see [8] for syntax.
   allchar       =  char / WSP
   char          =  DIGIT / ALPHA / other
   other         =  %x21-%x2f / %x3a-%x40 /
                    %x5b-%x60 / %7b-%7e
                    ; All printable, nonwhitespace US-ASCII
                    ; characters.
   newline       =  CR / ( CRLF )
```

The syntax for attribute tags and attribute value lists is specified in [8]. DAs and SA servers that process serialized registrations must handle them exactly as if they were registered by an SA. In the url-props production, the type token is optional. If the type token is absent, the URL's scheme is used as the type. If the maximum lifetime is specified (65535 sec.), the advertisement is taken to be permanent, and is continually refreshed by the DA or SA server until it exits. The API library should respect any advertised DA minimum refresh interval values, and otherwise, should only register after half or more of the lifetime has expired. If the lifetime is other than the maximum, the advertisement times out after the lifetime expires. Advertisements are registered in the scopes with which the DA or SA server is configured.

2.3 Processing Serialized Registration and Configuration Files

Implementations are encouraged to make processing of configuration and serialized registration files as transparent as possible to clients of the API. Agents processing the configuration file and the serialized registration file must log any errors using the platform specific error reporting mechanism. An agent must not fail if a file format error occurs.

For configuration files, errors must be caught at the latest when the relevant configuration item is used. Errors may be caught at the earliest when the configuration file is loaded into the executing agent. The default value must be substituted when an error is caught. Configuration file loading must complete prior to the initiation of the first networking connection.

For serialized registration files, errors must be caught and reported when the file is loaded, and the offending registration must be rejected. Serialized registration must be complete before the DA or SA server accepts the first network request.

3.0 The API

The C language binding presents a minimal overhead implementation mapping directly into the protocol. To conform with standard C practice, all character strings passed to and returned through the API are null terminated, even though the SLP protocol does not use null terminated strings. Strings passed as parameters are in the multi-byte UTF-8 encoding but they must be passed as a type char*, a null terminated array of bytes. In the common case of US-ASCII, the usual one byte per character C strings work because the US-ASCII encoding is a subset of the UTF-8 encoding.

Unless otherwise noted, a NULL parameter value can be used to denote "no value." Some parameters may have restrictions. If any parameter fails to satisfy the restrictions on its value, the operation returns a PARAMETER_BAD error.

An exception is scope lists in the UA API. A NULL or empty string for a scope list parameter indicates "default the list". Section 4.3 describes how to construct the default list.

3.1 Constant Types

3.1.1 URL Lifetimes

Synopsis

```
   typedef enum {
     SLP_LIFETIME_DEFAULT = 10800,
     SLP_LIFETIME_MAXIMUM = 65535
   } SLPURLLifetime;
```

Description

The SLPURLLifetime enum type contains frequently used URL lifetime values, in seconds. SLP_LIFETIME_DEFAULT is 3 hours, while SLP_LIFETIME_MAXIMUM is about 18 hours and corresponds to the maximum size of the lifetime field in SLP messages. A registration made with SLP_LIFETIME_MAXIMUM causes the service advertisement to be automatically re-registered.

3.2 Error Codes

Synopsis

```
      typedef enum {
        SLP_LAST_CALL                     = 1,
        SLP_OK                            = 0,
        SLP_LANGUAGE_NOT_SUPPORTED        = -1,
        SLP_PARSE_ERROR                   = -2,
        SLP_INVALID_REGISTRATION          = -3,
        SLP_SCOPE_NOT_SUPPORTED           = -4,
        SLP_REFRESH_REJECTED              = -15,
        SLP_NOT_IMPLEMENTED               = -17,
        SLP_BUFFER_OVERFLOW               = -18,
        SLP_NETWORK_TIMED_OUT             = -19,
        SLP_NETWORK_INIT_FAILED           = -20,
        SLP_MEMORY_ALLOC_FAILED           = -21,
        SLP_PARAMETER_BAD                 = -22,
        SLP_NETWORK_ERROR                 = -23,
        SLP_INTERNAL_SYSTEM_ERROR         = -24,
        SLP_HANDLE_IN_USE                 = -25,
        SLP_TYPE_ERROR                    = -26
      } SLPError ;
```

Description

The SLPError enum contains error codes that are returned from API functions or passed as error parameters to callback functions.

The SLP protocol errors OPTION_NOT_UNDERSTOOD, VERSION_NOT_SUPPORTED, INTERNAL_ERROR, MSG_NOT_SUPPORTED, AUTHENTICATON_UNKNOWN, and DA_BUSY_NOW should be handled internally and not surfaced to clients through the API.

The error codes SLP_OK, SLP_LANGAUGE_NOT_UNDERSTOOD, SLP_PARSE_ERROR, SLP_SCOPE_NOT_SUPPORTED, and SLP_REFRESH_REJECTED correspond directly to the protocol error codes as described in [8]. In addition, SLP_PARSE_ERROR may be returned by the API library if the library itself detects any syntactic errors.

The remaining error codes indicate the following conditions:

SLP_LAST_CALL

The SLP_LAST_CALL code is passed to callback functions for both synchronous and asynchronous calls when the API library has no more data for them and therefore no further calls will be made to the callback on the currently outstanding operation. The callback can use this to signal the main body of the client code that no more data will be forthcoming on the operation, so that the main body of the client code can break out of data collection loops. The other callback parameters are all NULL. If an SLP request results in no return values, then only one call is made, with the error parameter set to SLP_LAST_CALL.

SLP_NETWORK_INIT_FAILED

The network failed to initialize properly.

SLP_NETWORK_TIMED_OUT

No reply can be obtained in the time specified by the configured timeout interval for a unicast request.

SLP_NETWORK_ERROR

Networking failed during normal operation.

SLP_BUFFER_OVERFLOW

An outgoing request overflowed the maximum network MTU size.

SLP_MEMORY_ALLOC_FAILED

The API failed to allocate memory.

SLP_PARAMETER_BAD

A bad parameter was passed into the API.

SLP_INTERNAL_SYSTEM_ERROR

A basic failure of the API, such as the failure of a system call, occurred.

SLP_HANDLE_IN_USE

An attempt was made to make an API call on an SLPHandle that already has an outstanding call on it.

SLP_TYPE_ERROR

If the API supports type checking of registrations against service type templates, this error is returned if the attributes in a registration do not match the service type template for the service.

More information on the causes of these errors may be available
through the platform specific system error reporting API.

3.3 SLPBoolean

Synopsis

```
      typedef enum {
        SLP_FALSE = 0,
        SLP_TRUE = 1
      } SLPBoolean;
```

Description

The SLPBoolean enum is used as a Boolean flag.

3.4 Structure Types

3.4.1   SLPSrvURL

Synopsis

```
      typedef struct srvurl {
        char *s_pcSrvType;
        char *s_pcHost;
        int   s_iPort;
        char *s_pcNetFamily;
        char *s_pcSrvPart;
      } SLPSrvURL;
```

Description

The SLPSrvURL structure is filled in by the SLPParseSrvURL() function when a service URL string is parsed. The fields correspond to different parts of the URL. Note that the structure is in conformance with the standard Berkeley sockets struct servent, with the exception that the pointer to an array of characters for aliases (s_aliases field) is replaced by the pointer to host name (s_pcHost field).

s_pcSrvType
   A pointer to a character string containing the service type name, including naming authority. The service type name includes the "service:" if the URL is of the service: scheme [8].

s_pcHost
   A pointer to a character string containing the host identification information.

s_iPort
   The port number, or zero if none. The port is only available if the transport is IP.

s_pcNetFamily
   A pointer to a character string containing the network address

      family identifier. Possible values are "ipx" for the IPX
      family, "at" for the Appletalk family, and "" (i.e. the empty
      string) for the IP address family.

   s_pcSrvPart

      The remainder of the URL, after the host identification.

The host and port should be sufficient to open a socket to the machine hosting the service, and the remainder of the URL should allow further differentiation of the service.

3.4.2 SLPHandle

Synopsis

```
      typedef void* SLPHandle;
```

Description

The SLPHandle type is returned by SLPOpen() and is a parameter to all SLP functions. It serves as a handle for all resources allocated on behalf of the process by the SLP library. The type is opaque, since the exact nature differs depending on the implementation.

3.5 Callback Types

The callback functions report the results of an SLP protocol operation. In addition to parameters for reporting the results of the operation, each callback parameter list contains an error code parameter and a cookie parameter. The error code parameter reports the error status of the ongoing (for asynchronous) or completed (for synchronous) operation. The cookie parameter allows the client code starting the operation to pass information down to the callback through the API function without using global variables. If the cookie is not set when the API function is called, the parameter is NULL.

The callback returns an SLPBoolean to indicate whether the API library should continue processing the operation. If the value returned from the callback is SLP_TRUE, asynchronous operations are terminated, synchronous operations ignore the return, since the operation is already complete.

Section 4.1 contains more detail on callback processing.

3.5.1 SLPRegReport

Synopsis

```
   typedef void SLPRegReport(SLPHandle hSLP,
                             SLPError  errCode,
                             void      *pvCookie);
```

Description

The SLPRegReport callback type is the type of the callback function to the SLPReg() and SLPDereg() functions.

Parameters

hSLP

The SLPHandle used to initiate the operation.

errCode
: The error code.

pvCookie
: The cookie.

3.5.2 SLPSrvTypeCallback

Synopsis

```
typedef SLPBoolean SLPSrvTypeCallback(SLPHandle   hSLP,
                                      const char* pcSrvTypes,
                                      SLPError    errCode,
                                      void        *pvCookie);
```

Description

The SLPSrvTypeCallback callback type is the type of the callback function parameter to the SLPFindSrvTypes() function.

Parameters

hSLP
: The SLPHandle used to initiate the operation.

pcSrvTypes
: A character buffer containing a comma separated, null terminated list of service types.

errCode
: The error code.

pvCookie
: The cookie.

Returns

The client code should return SLP_TRUE if more data is desired, otherwise SLP_FALSE.

3.5.3 SLPSrvURLCallback

Synopsis

```
typedef SLPBoolean SLPSrvURLCallback(SLPHandle      hSLP,
                                     const char*    pcSrvURL,
                                     unsigned short sLifetime,
                                     SLPError       errCode,
                                     void           *pvCookie);
```

Description

The SLPSrvURLCallback callback type is the type of the callback function parameter to the SLPFindSrvs() function.

Parameters

hSLP
The SLPHandle used to initiate the operation.

pcSrvURL
A character buffer containing the returned service URL.

sLifetime
An unsigned short giving the life time of the service advertisement, in seconds. The value must be an unsigned integer less than or equal to SLP_LIFETIME_MAXIMUM.

errCode
The error code.

pvCookie
The cookie.

Returns

The client code should return SLP_TRUE if more data is desired, otherwise SLP_FALSE.

3.5.4 SLPAttrCallback

Synopsis

```
typedef SLPBoolean SLPAttrCallback(SLPHandle   hSLP,
                                   const char* pcAttrList,
                                   SLPError    errCode,
                                   void        *pvCookie);
```

Description

The SLPAttrCallback type is the callback type of the callback function parameter to SLPFindAttrs() function.

Parameters

hSLP
The SLPHandle used to initiate the operation.

pcAttrList
A character buffer containing a comma separated, null terminated list of attribute id/value assignments, in SLP wire format, see [8] for details.

errCode
The error code.

```
      pvCookie
         The cookie.

Returns
```

The client code should return SLP_TRUE if more data is desired, otherwise SLP_FALSE.

3.6 Opening and Closing an SLPHandle

3.6.1 SLPOpen

Synopsis

```
   SLPError SLPOpen(const char *pcLang,
                    SLPBoolean isAsync,
                    SLPHandle  *phSLP);
```

Description

Returns a SLPHandle handle in the phSLP parameter for the language locale passed in as the pcLang parameter. If the isAsync parameter is TRUE, operations are performed asynchronously. The handle encapsulates the language locale for SLP requests issued through the handle, and any other resources required by the implementation. However, SLP properties are not encapsulated by the handle; they are global. The return value of the function is an SLPError code indicating the status of the operation.

An SLPHandle can only be used for one SLP API operation at a time. If the original operation was started asynchronously, any attempt to start an additional operation on the handle while the original operation is pending results in the return of an SLP_HANDLE_IN_USE error from the API function. If an implementation is unable to support an asynchronous (resp. synchronous) operation, due to memory constraints or lack of threading support, the SLP_NOT_IMPLEMENTED flag must be returned when the isAsync flag is SLP_TRUE (resp. SLP_FALSE).

Parameters

pcLang
The RFC 1766 Language Tag [7] for the natural language locale of requests and registrations issued on the handle.

isAsync
A SLPBoolean indicating whether the SLPHandle should be opened for asynchronous operation or not.

phSLP
A pointer to an SLPHandle, in which the open SLPHandle is returned. If an error occurs, the value upon return is NULL.

3.6.2 SLPClose

Synopsis

```
   void SLPClose(SLPHandle hSLP);
```

Description

Frees all resources associated with the handle. If the handle was invalid, the function returns silently. Any outstanding synchronous or asynchronous operations are cancelled immediately, so their callback functions will not be called any further.

Parameters

SLPHandle
   A SLPHandle handle returned from a call to SLPOpen().

3.7 SA API

3.7.1 SLPReg

Synopsis

```
SLPError SLPReg(SLPHandle            hSLP,
                const char           *pcSrvURL,
                const unsigned short usLifetime,
                const char           *pcSrvType,
                const char           *pcAttrs
                SLPBoolean           fresh,
                SLPRegReport         callback,
                void                 *pvCookie);
```

Description

Registers the URL in pcSrvURL having the lifetime usLifetime with the attribute list in pcAttrs. The pcAttrs list is a comma separated list of attribute assignments in the wire format (including escaping of reserved characters). The usLifetime parameter must be nonzero and less than or equal to SLP_LIFETIME_MAXIMUM. The pcSrvType parameter is a service type name and may be NULL or the empty string if the URL is a service: URL. The fresh parameter is ignored. The format for pcAttrs and pcScopeList can be found in [8]. Registrations and updates take place in the language locale of the hSLP handle.

The API library is required to perform the operation in all scopes obtained through configuration.

Parameters

hSLP
   The language specific SLPHandle on which to register the advertisement.

pcSrvURL
   The URL to register. May not be NULL or the empty string. This parameter must be a properly formatted URL [3]; otherwise, the

      SLP SrvReg returns a parse error and the callback is called
      with the SLP_PARSE_ERROR error code.

   usLifetime

An unsigned short giving the life time of the service advertisement, in seconds. The value must be an unsigned integer less than or equal to SLP_LIFETIME_MAXIMUM and greater than zero.

pcSrvType

The service type. If a service: URL is present in pcSrvURL and this parameter is NULL or an empty string, then the value of the the service type field in the SrvReg message is obtained from the service: URL's scheme [8].

pcAttrs

A comma separated list of attribute assignment expressions for the attributes of the advertisement. See [8] for the format. Use NULL or the empty string for no attributes.

fresh

Ignored.

callback

A callback to report the operation completion status.

pvCookie

Memory passed to the callback code from the client. May be NULL.

Returns

One of the SLPError codes is returned indicating the status of starting the operation.

3.7.2 **SLPDereg**

Synopsis

```
SLPError SLPDereg(SLPHandle    hSLP,
                  const char   *pcURL,
                  SLPRegReport callback,
                  void         *pvCookie);
```

Description

Deregisters the advertisement for URL pcURL in all scopes where the service is registered and all language locales. The deregistration is not just confined to the locale of the SLPHandle, it is in all locales. The API library is required to perform the operation in all scopes obtained through configuration.

Parameters

hSLP

The language specific SLPHandle to use for deregistering.

pcURL

The URL to deregister. May not be the empty string. This parameter must be a properly formatted URL [3]; otherwise, the

SLP SrvDeReg returns a parse error and the callback is called with the SLP_PARSE_ERROR error code.

callback
A callback to report the operation completion status.

pvCookie
Memory passed to the callback code from the client. May be NULL.

Returns

One of the SLPError codes is returned indicating the status of starting the operation.

3.7.3 SLPFindSrvTypes

Synopsis

```
SLPError SLPFindSrvTypes(SLPHandle           hSLP,
                         const char         *pcNamingAuthority,
                         const char         *pcScopeList,
                         SLPSrvTypeCallback callback,
                         void               *pvCookie);
```

Description

The SLPFindSrvType() function issues an SLP service type request for service types in the scopes indicated by the pcScopeList. The results are returned through the callback parameter. The service types are independent of language locale, but only for services registered in one of scopes and for the indicated naming authority. If the naming authority is "*", then results are returned for all naming authorities. If the naming authority is NULL or the empty string, then the default naming authority, IANA, is used. "IANA" is not a valid naming authority name, and it is a PARAMETER_BAD error to include it explicitly.

The service type names are returned with the naming authority intact. If the naming authority is the default (i.e. empty string) then it is omitted, as is the separating ".". Service type names from URLs of the service: scheme are returned with the "service:" prefix intact [8]. See [9] for more information on the syntax of service type names.

Parameters

hSLP
The SLPHandle on which to search for types.

pcNamingAuthority
The naming authority to search. Use "*" for all naming

      authorities and NULL or the empty string for the default
      naming authority.

   pcScopeList

The comma separated list of scope names to search for service types. Use NULL or the empty string for the default scope list.

callback

A callback function through which the results of the operation are reported.

pvCookie

Memory passed to the callback code from the client. May be NULL.

Returns

One of the SLPError codes is returned indicating the status of starting the operation.

3.7.4 SLPFindSrvs

Synopsis

```
SLPError SLPFindSrvs(SLPHandle         hSLP,
                     const char        *pcServiceType,
                     const char        *pcScopeList,
                     const char        *pcSearchFilter,
                     SLPSrvURLCallback callback,
                     void              *pvCookie);
```

Description

Issue the query for services on the locale-specific SLPHandle and return the results through the callback. The parameters determine the results.

Parameters

hSLP

The locale-specific SLPHandle on which to search for services.

pcServiceType

The service type name, including naming authority if any, for the request, such as can be discovered using SLPSrvTypes(). May not be NULL or the empty string.

pcScopeList

The comma separated list of scope names. Use NULL or the empty string for the default scope list.

pcSearchFilter

A query formulated of attribute pattern matching expressions in the form of an LDAPv3 Search Filter, see [5]. If this

filter is NULL or the empty string, all services of the
requested type in the specified scopes are returned. The
search filter should be a simple search filter as defined in
[8].

callback
A callback function through which the results of the operation are reported.

pvCookie
Memory passed to the callback code from the client. May be NULL.

Returns

One of the SLPError codes is returned indicating the status of starting the operation.

3.7.5 SLPFindAttrs

Synopsis

```
SLPError SLPFindAttrs(SLPHandle        hSLP,
                      const char      *pcURL,
                      const char      *pcScopeList,
                      const char      *pcAttrIds,
                      SLPAttrCallback callback,
                      void            *pvCookie);
```

Description

This function returns service attributes matching the attribute ids for the indicated URL. The attribute information returned is for the matching advertisement in the locale of the SLPHandle.
The result is filtered with an SLP attribute request filter string parameter, pcAttrIds, the syntax of which is described in [8]. If the filter string is NULL or the empty string, all attributes are returned.

Parameters

hSLP
The language specific SLPHandle on which to search for attributes.

pcURL
The URL. May not be NULL or the empty string. This parameter must be a properly formatted URL [3]; otherwise, the SLP AttrRqst returns a parse error and the callback is called with the SLP_PARSE_ERROR error code.

pcScopeList
The comma separated list of scope names. Use NULL or the empty string for the default scope list.

pcAttrIds
The filter string indicating which attribute values to return.

```
      Use NULL or the empty string to indicate all values. See [8]
      for the exact format of the filter string.

   callback
```

A callback function through which the results of the operation are reported.

pvCookie

Memory passed to the callback code from the client. May be NULL.

Returns

One of the SLPError codes is returned indicating the status of starting the operation.

3.8 Miscellaneous Functions

3.8.1 SLPGetRefreshInterval

Synopsis

```
unsigned short SLPGetRefreshInterval();
```

Description

Returns the maximum across all DAs of the min-refresh-interval attribute. This value satisfies the advertised refresh interval bounds for all DAs, and, if used by the SA as the minimum service advertisement lifetime, assures that no refresh registration will be rejected. If no DA advertises a min-refresh-interval attribute, a value of 0 is returned.

Returns

If no error, the maximum refresh interval value allowed by all DAs (a positive integer). If no DA advertises a min-refresh-interval attribute, returns 0. If an error occurs, returns an SLP error code.

3.8.2 SLPFindScopes

Synopsis

```
SLPError SLPFindScopes(SLPHandle hSLP,
                       char      **ppcScopeList);
```

Description

Sets ppcScopeList parameter to a pointer to a comma separated list including all available scope values. See Section 4.3 for a description of how the list is determined. If there is any order to the scopes, preferred scopes are listed before less desirable scopes. There is always at least one name in the list, the default scope, "DEFAULT".

Parameters

hSLP
   The SLPHandle on which to search for scopes.

ppcScopeList

On return, contains a pointer to a null terminated string with the comma-separated list of scopes. The memory should be freed by calling SLPFree().

Returns

If no error occurs, returns SLP_OK, otherwise, the appropriate error code.

3.8.3 **SLPParseSrvURL**

Synopsis

```
SLPError SLPParseSrvURL(char        *pcSrvURL
                        SLPSrvURL** ppSrvURL);
```

Description

The URL passed in as the argument is parsed into a SLPSrvURL structure and is return in the ppSrvURL pointer. If a parse error occurs, returns SLP_PARSE_ERROR as the value of the function. The input buffer pcSrvURL may be destructively modified during the parse and used to fill in the fields of the return structure. The structure returned in ppSrvURL should be freed with SLPFree().

If the URL has no service part, the s_pcSrvPart string is the empty string, "", i.e. not NULL. If pcSrvURL is not a service: URL, then the s_pcSrvType field in the returned data structure is the URL's scheme, which might not be the same as the service type under which the URL was registered. If the transport is IP, the s_pcTransport field is the empty string. If the transport is not IP or there is no port number, the s_iPort field is zero.

Parameters

pcSrvURL

The null terminated URL string to parse. It may be destructively modified to produce the output structure. This parameter must be a properly formatted URL; otherwise, function returns the SLP_PARSE_ERROR error code.

ppSrvURL

On return, contains a pointer to the SLPSrvURL structure with the parsed URL, or NULL if the parse failed. The memory should be freed by a call to SLPFree() when no longer needed.

Returns

If no error occurs, the return value is SLP_OK. Otherwise, the appropriate error code is returned.

3.8.4   SLPParseAttrs

Synopsis

```
   SLPError SLPParseAttrs(const char *pcAttrList,
                          const char *pcAttrId,
                         char       **ppcAttrVal);
```

Description

Parses an attribute list to obtain the attribute value of a specified attribute ID. SLP_PARSE_ERROR is returned if a value for pcAttrId can not be found. The attribute value string returned in ppcAttrVal must be freed with SLPFree().

Parameters

pcAttrList
: A comma separated list of attribute assignment expressions. See [8] for the format.

pcAttrId
: The string indicating which attribute value to return. May not be NULL or the empty string.

ppcAttrVal
: Upon return, a pointer to the buffer containing the attribute value. The returned memory should be freed by a call to SLPFree() when no longer needed.

Returns

If no error occurs, the return value is SLP_OK. Otherwise, the appropriate error code is returned. If this function is not implemented, the library should return SLP_NOT_IMPLEMENTED. If a parse error occurs, the library should return SLP_PARSE_ERROR.

3.8.5 **SLPEscape**

Synopsis

```
   SLPError SLPEscape(const char *pcInbuf,
                      char        **ppcOutBuf,
                      SLPBoolean isTag);
```

Description

Process the input string in pcInbuf and escape any SLP reserved characters. If the isTag parameter is SLPTrue, then look for bad tag characters and signal an error if any are found by returning the SLP_PARSE_ERROR code. The results are put into a buffer allocated by the API library and returned in the ppcOutBuf parameter. This buffer should be deallocated using SLPFree() when the memory is no longer needed.

Parameters

pcInbuf
: Pointer to he input buffer to process for escape characters.

```
      ppcOutBuf
         On output, contains a pointer to a copy of the input buffer
         with the SLP reserved characters escaped. Must be freed using
         SLPFree()when the memory is no longer needed.

      isTag
         When true, the input buffer is checked for bad tag characters.

Returns
```

Return SLP_PARSE_ERROR if any characters are bad tag characters and the isTag flag is true, otherwise SLP_OK, or the appropriate error code if another error occurs.

3.8.6 SLPUnescape

Synopsis

```
   SLPError SLPUnescape(const char *pcInbuf,
                        char        **ppcOutBuf,
                        SLPBoolean isTag);
```

Description

Process the input string in pcInbuf and unescape any SLP reserved characters. If the isTag parameter is SLPTrue, then look for bad tag characters and signal an error if any are found with the SLP_PARSE_ERROR code. No transformation is performed if the input string is an SLP opaque. The results are put into a buffer allocated by the API library and returned in the ppcOutBuf parameter. This buffer should be deallocated using SLPFree() when the memory is no longer needed.

Parameters

pcInbuf
Pointer to he input buffer to process for escape characters.

ppcOutBuf
On output, contains a pointer to a copy of the input buffer with the SLP reserved characters unescaped. Must be freed using SLPFree()when the memory is no longer needed.

isTag
When true, the input buffer is checked for bad tag characters.

Returns

Return SLP_PARSE_ERROR if any characters are bad tag characters and the isTag flag is true, otherwise SLP_OK, or the appropriate error code if another error occurs.

3.8.7 SLPFree

Synopsis

```
   void SLPFree(void* pvMem);
```

Description

Frees memory returned from SLPParseSrvURL(), SLPFindScopes(),SLPEscape(), SLPUnescape(), and SLPGetProperty().

Parameters

pvMem

      A pointer to the storage allocated by the
      SLPParseSrvURL(),SLPEscape(), SLPUnescape(), or
      SLPFindScopes() function. Ignored if NULL.

3.8.8   **SLPGetProperty**

Synopsis

```
   SLPError SLPGetProperty(const char *pcPropertyName,
                           char       **ppcPropertyValue);
```

Description

Upon return, the ppcPropertyValue parameter is set to a pointer to the property value string corresponding to pcPropertyName, or NULL if the pcPropertyName string does not name a valid SLP property. The ppcPropertyValue buffer should be deallocated using SLPFree() when the memory is no longer needed.

Parameters

pcPropertyName
: Null terminated string with the property name, from Section 2.1.

ppcPropertyValue
: On return, contains a pointer to a string with the property value, or NULL if the pcPropertyName parameter does not name a property.

Returns

Returns one of the following status codes: SLP_OK, SLP_MEMORY_ALLOC_FAILED, SLP_NOT_IMPLEMENTED, or SLP_PARAMETER_BAD. The latter is returned if the pcPropertyName parameter does not name a valid SLP property.

3.8.9   **SLPSetProperty**

Synopsis

```
   SLPError SLPSetProperty(const char *pcPropertyName,
                           const char *pcPropertyValue);
```

Description

Sets the value of the SLP property to the new value. The pcValue parameter should be the property value as a string.

Parameters

pcPropertyName

      Null terminated string with the property name, from Section
      2.1.

   pcPropertyValue

Null terminated string with the property value. Use NULL or the empty string to indicate that the property should be unset, and thus return to default.

Returns

Returns one of the following status codes: SLP_OK, SLP_MEMORY_ALLOC_FAILED, SLP_NOT_IMPLEMENTED, or SLP_PARAMETER_BAD. The latter is returned if the pcPropertyName parameter does not name a valid SLP property.

3.8.10 SLPGetExtensionInterface

Synopsis

```
   SLPError SLPGetExtensionInterface(SLPHandle  hSLP,
                                     const char *pcExtName,
                                     void       **ppExtInterface);
```

Description

Called with an initialized SLPHandle and the name of an SLP extension. On return, a pointer to the extension interface is in the ppExtInterface parameter, or NULL if there is no such extension. Exactly how the code for the extension is located, the exact format of the interface structure implementing access to the extension, how the interface code is made available (i.e.dynamically linked v.s. statically linked), and how names of extensions are formatted are implementation dependent issues.

Parameters

hSLP
The language specific SLPHandle to use for locating the extension interface.

pcExtName
The name of the extention to return.

ppExtInterface
On return, contains a pointer to a structure implementing the interface.

Returns

If no error occurs, the return value is SLP_OK. Otherwise, the appropriate error code is returned. If no extension is available corresponding to pcExtName, the return value is SLP_NOT_IMPLEMENTED and the ppExtInterface parameter is NULL.

3.8.11 SLPFreeExtensionInterface

Synopsis

```
   SLPError SLPFreeExtensionInterface(void **ppExtInterface);
```

Description

Free up memory and code associated with the interface accessed through ppExtInterface. Upon return, ppExtInterface is NULL and the memory for the interface is freed.

Parameters

ppExtInterface
A valid interface implementation obtained through SLPGetExtInterface()

Return

If no error occurs, the return value is SLP_OK. Otherwise, the appropriate error code is returned.

4.0 Implementation Considerations

This section discusses a number of implementation considerations.

4.1 Callback Semantics

There will always be at least one callback for every API operation: a callback with the error code set to SLP_LAST_CALL indicating that the request has completed. There may be more callbacks in certain if a result is returned. Any callback in which the error code is not set to SLP_LAST_CALL is a return report. If there are no results to report, the callback with SLP_LAST_CALL set is the only callback.

For the SA API, SLPSrvReg() and SLPSrvDereg() callbacks are only ever called once with a return report. If the SA API implementation performs DA forwarding directly, then it must wait until all DA replies are back before calling the callback. If the SA API implementation registers with an SA server, the SA server replies with a single SrvAck, the contents of which are reported through the callback.

For the UA API, only one callback containing a return report is ever made if a DA is in use for SLPFindSrvTypes(). If the UA multicasts a request or unicasts to multiple SAs, multiple calls to a callback with return reports may result for SLPFindSrvTypes() if multiple replies are received. The UA may also collate replies from multiple SAs and present them through a single callback. Only one return report callback invocation ever occurs for SLPFindAttrs(), and multiple callback reports are possible for SLPFindSrvs() regardless of how the request was transmitted if multiple URLs are received in the reply.

The callback function is called whenever the API library has results to report. The callback code is required to check the error code parameter before looking at the other parameters. If the error code is not SLP_OK, the other parameters may be invalid. The API library may terminate any outstanding operation on which an error occurs. The callback code can similarly indicate that the operation should be terminated by passing back SLP_FALSE.

Callback functions are not permitted to recursively call into the API on the same SLPHandle. If an attempt is made to recursively call into the API, the API function returns SLP_HANDLE_IN_USE. Prohibiting recursive callbacks on the same handle simplifies implementation of thread safe code, since locks held on the handle will not be in place during a second outcall on the handle. Handle creation should be fairly lightweight so a client program can easily support multiple outstanding calls.

The total number of results received can be controlled by setting the net.slp.maxResults parameter. Note that this parameter controls the number of results received, not the number of return messages. In the case of a multicast SrvRqst, for example, the number of return messages may be less than the number of results, since one message may contain multiple results.

There are five reasons why a call can terminate:

DA reply received

A reply from a DA has been received and therefore nothing more is expected, or the request timed out.

Unicast SA messages received

All messages were received in reply to a unicast request to one or several SAs, or one or more of the requests timed out.

Multicast terminated

The multicast convergence time has elapsed and the API library multicast code is giving up.

Multicast null results

Nothing new has been received during multicast for a while and the API library multicast code is returning the existing replies, if any.

Maximum results

The user has set the net.slp.maxResults property and that number of results has been collected and returned

4.2 Asynchronous Semantics

If a handle parameter to an API function is opened asynchronously, API function calls on the handle check the other parameters, open the appropriate operation and return immediately. If the handle parameter was opened synchronously, the API function call blocks

until all results are processed, and returns only after the callback
function has been called with the callback error code set to
SLP_LAST_CALL. If an error occurs in the process of starting the SLP
operation, an error code is returned from the API function. Errors

that occur as a result of the SLP operation are reported to the callback, and are not returned from the API function.

If asynchronous semantics are supported, the API library is required to be thread-safe. The API must be re-entrant in order to avoid interference between callbacks.

4.3 Scope and DA Configuration and Discovery

The API must conform to the scope and DA configuration rules described in Section 8 of [8]. Preconfigured scopes and DAs, whether through static configuration or DHCP configuration, must be available via the configuration properties net.slp.configuredScopes and net.slp.configuredDAAddresses.

Functions in the UA API have a scope parameter that determines the scopes used in UA requests. If that parameter is not NULL or the empty string, then the scopes in that parameter are used for the request. If that parameter is NULL or the empty string, the UA API library determines the scopes to use in the following fashion. If net.slp.configuredScopes is set, the listed scopes on net.slp.configuredScopes are used. If net.slp.configuredScopes is not set, the UA must use scopes obtained from any configured or discovered DAs, or scopes discovered through dynamic SA discovery, exactly as would be the case if the SLPFindScopes() function were called.

Dynamic scope and DA information is available at any time through the API functions. Calling SLPSrvRqst() with the service type parameter set to "service:directory-agent" returns all discoverable DAs, including any that were configured. Calling SLPFindScopes() returns all discoverable scopes including any that were configured. SLPFindScopes() uses the rules outlined in [8] to determine what sources to consult for scope information.

4.4 Multithreading

Implementations of the API are required to make API calls thread-safe. Access to data structures shared between threads must be coordinated to avoid corruption or invalid access. Implementations should also attempt to maximize the amount of concurrent thread access to the API library.

4.5 Type Checking for Registrations

Service templates [9] allow SLP registrations to be type checked for correctness. Implementations of the API may use service type information for type checking. If a type error occurs, the registration should terminate with SLP_TYPE_ERROR.

String encoded attribute values do not include explicit type
information. All UA implementations and those SA and DA
implementations that choose to support type checking should use the
type rules described in [9] in order to convert from the string
representation on the wire to an object typed appropriately.

4.6 Refreshing Registrations

SLP advertisements carry an explicit lifetime. After the lifetime expires, the DA flushes the registration from its cache. In some cases, an application may want to have the URL continue being registered for the entire time during which the application is executing. The API includes provision for clients to indicate whether they want URLs to be automatically refreshed: SLPReg() is called with the pLifetime parameter equivalent to SLP_LIFETIME_MAXIMUM (65535 seconds). Implementations of the SA API must provide automatic re-registration if a registration is made with the maximum lifetime. A client using this facility should explicitly deregister the service URL before exiting, since the API implementation may not be able to assure that the URL is deregistered when the application exits, although it times out in the DA eventually.

4.7 Character Set Encoding

Characters buffer parameters are represented in UTF-8 despite the defined type of char* or const char*. API functions are required to handle the full range of multi-byte UTF-8 characters because the SLP protocol requires it, but the API implementation can represent the characters internally in any convenient way. On the wire, all characters are converted to UTF-8 anyway.

Inside URLs, characters that are not allowed by URL syntax [3] must be escaped according to the URL escape character convention. Strings that are included in SLP messages may include SLP reserved characters and can be escaped by clients through convenience functions provided by the API. The character encoding used in escapes is UTF-8.

Due to constraints in SLP, no string parameter passed to the API may exceed 64K bytes in length. An API function that encounters a string longer than 64K should return SLP_PARSE_ERROR.

4.8 Error Handling

All errors encountered processing SLP messages should be logged, especially for the SA server and DA.

For the UA API, since no errors are returned for multicast requests, and only a single DA is ever used at a time, there is only one case where multiple invocations of a callback could result in one or more calls to callbacks with the error code set to something other than SLP_OK: a unicast request to multiple SAs. In all other cases, there is a single callback invocation in which the error code is set if an error occurs, in addition to the last call callback.

For the SA client API, a registration or deregistration to one DA among several may result in an error, but since only a single callback is ever made reporting return status for the SA API, the error code is only reported if no SrvAck indicating success was received.

Since registration with an SA server results in the same error conditions as with a DA, the SA server is not required to forward a SrvReg to any DAs if the registration fails. The SA server must return a SrvAck to the client with the error code properly set. The SA server is also not required to wait to return the SrvAck to the SA client until registration with DAs has completed, since any errors occurring with DAs are likely to be unrelated to the content of the registration if the registration succeeded with the SA server.

4.9 Modular Implementations

Subset implementations that do not support the full range of functionality must support every interface in order to maintain link compatibility between compliant API implementations and applications. If a particular operation is not supported, a NOT_IMPLEMENTED error must be returned. Applications that are expected to run on a wide variety of platforms should be prepared for subset API implementations by checking returned error codes.

4.10   Handling Special Service Types

The DA service type, "service:directory-agent", and SA service type, "service:service-agent", are used internally in the SLP framework to discover DAs and SAs. The mechanism of DA and SA discovery is not normally exposed to the API client; however, the client may have interest in discovering DAs and SAs independently of their role in discovering other services. For example, a network management application may want to determine which machines are running SLP DAs. To facilitate that, API implementations must handle requests to find services and attributes for these two service types so that API clients obtain the information they expect.

In particular, if the UA is using a DA, SrvRqst and AttrRqst for these service types must be multicast and not unicast to the DA, as is the case for other service types. If the requests are not multicast, the DA will respond with an empty reply to a request for the SA service type and with its URL only to a request for the DA service type. The UA would therefore not obtain a complete picture of the available DAs and SAs.

4.11 Syntax for String Parameters

Query strings, attribute registration lists, attribute deregistration lists, scope lists, and attribute selection lists follow the syntax described in [8] for the appropriate requests. The API directly reflects the strings passed in from clients into protocol requests, and directly reflects out strings returned from protocol replies to clients. As a consequence, clients are responsible for formatting request strings, including escaping and converting opaque values to escaped byte encoded strings. Similarly, on output, clients are required to unescape strings and convert escaped string encoded opaques to binary. The functions SLPEscape() and SLPUnescape() can be used for escaping SLP reserved characters, but perform no opaque processing.

Opaque values consist of a character buffer containing a UTF-8 encoded string, the first characters of which are the nonUTF-8 encoding "\ff". Subsequent characters are the escaped values for the original bytes in the opaque. The escape convention is relatively simple. An escape consists of a backslash followed by the two hexadecimal digits encoding the byte. An example is "\2c" for the byte 0x2c. Clients handle opaque processing themselves, since the algorithm is relatively simple and uniform.

4.12 Client Side Syntax Checking

Client side API implementations may do syntax checking of scope names, naming authority names, and service type names. Since the C API is designed to be a thin layer over the protocol, some low memory SA implementations may find extensive syntax checking on the client side to be burdensome. If syntax checking uncovers an error in a parameter, the SLP_PARAMETER_BAD error must be returned. If any parameter is NULL and is required to be nonNULL, SLP_PARAMETER_BAD is returned.

4.13 SLP Configuration Properties

The SLP configuration properties properties established in the configuration file are accessible through the SLPGetProperty() and SLPSetProperty()functions. The SLPSetProperty() function only modifies properties in the running process, not in the configuration file. Properties are global to the process, affecting all threads and all handles created with SLPOpen. Errors are checked when the property is used and, as with parsing the configuration file, are logged. Program execution continues without interruption by substituting the default for the erroneous parameter. With the exception of net.slp.locale, net.slp.typeHint, and net.slp.maxResults, clients of the API should rarely be required to override these properties, since they reflect properties of the SLP network that are not of concern to individual agents. If changes are required, system administrators should modify the configuration file.

4.14 Memory Management

The only API functions returning memory specifically requiring deallocation on the part of the client are SLPParseSrvURL(), SLPFindScopes(), SLPEscape(), and SLPUnescape(), and SLPGetProperty(). This memory should be freed using SLPFree() when no longer needed.

Memory passed to callbacks from the API library belongs to the library and MUST NOT be retained or freed by the client code. Otherwise, crashes are possible. Clients are required to copy data out of the callback parameters. No other use of the parameter memory in callback parameters is allowed.

4.15 Multi-homed Hosts

On a multi-homed host, routing may be disabled between interfaces.
The net.slp.interfaces property must only be set if there is no

routing between any of the interfaces or if broadcast is used instead of multicast. If the net.slp.interfaces is set, the DA (if any) and SAs on the host should respond to a DA or SA advertisement request with an IP address or host name on the list. Replies to requests should be made with service advertisements that are reachable through the interface on which the request arrived. If packets are routed between the interfaces, then the DA and SAs must only advertise on the default interface.

Note that even if unicast packets are not routed between the interfaces, multicast may be routed through another router. The danger in listening for multicast on multiple interfaces is that the DA or SA may receive the same multicast request via more than one interface. Since the IP address is different on each interface, the DA or SA cannot identify the request as having already being answered via the previous responder's list. The requesting agent will end up getting URLs that refer to the same DA or service but have different addresses or host names.

4.16 Unicast UA Requests

If the net.slp.enableUnicastSARequest property is TRUE, UAs are required to use unicast directly to discovered SAs rather than use multicast or DAs for the request. This allows the UA to receive errors directly from SAs that it otherwise wouldn't, for example, if the SA supports simple queries only but the UA issues a complex query. For SrvRqst and AttrRqst, prior to sending a request, the UA performs a multicast service request for SAs that advertise the service type of interest. The request is then unicast to the returned SAs. For SrvTypeRqst, the UA performs a service requests for all SAs, and either constructs the returned list of service types based on the "service-type" attribute definition in the SAs' attribute lists, or sends a SrvTypeRqst to each SA individually. The UA may cache the results of returned SAAdverts for some period of time to avoid having to perform the repeat multicast for SAAdverts. Unicasting of UA requests should be used with caution, in particular, it should not be used as a substitute for DAs. Deploying DAs is likely to result in better network performance and scalability.

4.17 UA Caching

In general, clients of the UA API should limit repeat queries until the lifetime of the service advertisement is about to expire. Because the base protocol and API lack any support for notification when a new service comes up, however, some applications may want to poll periodically for new services. Such polling could completely overwhelm the network with requests, especially if multicast is in use.

In order to regulate polling, the UA API library should cache the results of queries and return them when a repeat query arrives within some short time, say 10 seconds. The lifetime of the cache entries should be kept short in order to avoid stale information.

5.0 Deprecated Features

The following features were defined in RFC 2614 and have been deprecated in this update due to changes in the SLP protocol:

1) The property net.slp.securityEnabled is no longer supported. Security in SLP is now handled through IPSEC. Implementations should ignore this property if it is in the configuration file.
2) Scope lists have been dropped from the serialized registration file. Serialized registrations must be made in the configured scopes for the DA or SA server. Existing files must be edited to remove the scopes attribute definition, because it will otherwise be treated as a normal SLP attribute definition
3) The SLPDelAttrs() function is no longer supported. SLP no longer allows incremental update of service advertisements. Existing implementations of SLP should return the SLP_NOT_IMPLEMENTED error code from this function.
4) The SLPFindAttrs() function no longer takes a service type name. Attribute Request by Service Type has been dropped from SLP.
5) The error codes SLP_AUTHENTICATION_ABSENT, SLP_AUTHENTICATION_FAILED, and SLP_INVALID_UPDATE are no longer supported because these errors no longer occur in the protocol.

6.0 Example

This example illustrates how to discover a mailbox.
A POP3 server registers itself with the SLP framework. The attributes it registers are "USER", a list of all users whose mail is available through the POP3 server.
The POP3 server code is the following:

```
SLPHandle slph;
SLPRegReport errCallback = POPRegErrCallback;
/* Create an English SLPHandle, asynchronous processing. */
SLPError err = SLPOpen("en", SLP_TRUE, &slph);
if( err != SLP_OK ) {
  /* Deal with error. */
}

/* Create the service: URL and attribute parameters. */
const char* surl = "service:pop3://mail.netsurf.de"; /* the URL
*/
const char *pcAttrs = "(user=zaphod,trillian,roger,marvin)"
/* Perform the registration. */
err = SLPReg(slph,
             surl,
             SLP_LIFETIME_DEFAULT,
             ppcAttrs,
             errCallback,
             NULL);

if (err != SLP_OK ) {
  /*Deal with error.*/
}
```

The errCallback reports any errors:

```
   void
   POPRegErrCallback(SLPHandle hSLP,
                     SLPError errCode,
                     unsigned short usLifetime,
                     void* pvCookie) {
     if( errCode != SLP_OK ) {
       /* Report error through a dialog, message, etc. */
     }

     /*Use lifetime interval to update periodically. */
   }
```

The POP3 client locates the server for the user with the following
code:

```
   /*
   * The client calls SLPOpen(), exactly as above.
   */

   const char *pcSrvType   = "service:pop3"; /* the service type  */
   const char *pcScopeList = "default";      /* the scope         */
   const char *pcFilter    = "(user=roger)"; /* the search filter */
   SLPSrvURLCallback srvCallback =           /* the callback      */
                                   POPSrvURLCallback;
   err = SLPFindSrvs(slph,
                     pcSrvType, pcScopeList, pcFilter,
                     srvCallback, NULL);
   if( err != SLP_OK ) {
     /* Deal with error. */
   }
```

Within the callback, the client code can use the returned POP
service:

```
   SLPBoolean
   POPSrvURLCallback(SLPHandle hSLP,
                     const char* pcSrvURL,
                     unsigned short sLifetime,
                     SLPError errCode,
                     void* pvCookie) {

     if( errCode != SLP_OK ) {
       /* Deal with error. */
     }

      SLPSrvURL* pSrvURL;
      errCode = SLPParseSrvURL(pcSrvURL, &pSrvURL);
      if (err != SLP_OK ) {
        /* Deal with error. */
      } else {
        /* get the server's address */
        struct hostent *phe = gethostbyname(pSrvURL.s_pcHost);
        /* use hostname in pSrvURL to connect to the POP3 server
         *      . . .
         */
        SLPFreeSrvURL((void*)pSrvURL); /* Free the pSrvURL storage*/
     }

     return SLP_FALSE;                /* Done! */
   }
```

A client that wanted to discover all the users receiving mail at the server uses with the following query:

```
/*
 * The client calls SLPOpen(), exactly as above. We assume the
 * service: URL was retrieved into surl.
 */

   const char *pcScopeList = "default";   /* the scope            */
   const char *pcAttrFilter    = "use";   /* the attribute filter */
   SLPAttrCallback attrCallBack =         /* the callback         */
                                  POPUsersCallback

   err =
        SLPFindAttrs(slph,
                     surl,
                     pcScopeList, pcAttrFilter,
                     attrCallBack, NULL);
   if( err != SLP_OK ) {
     /* Deal with error. */
   }
```

The callback processes the attributes:

```
   SLPBoolean
   POPUsersCallback(const char* pcAttrList,
                    SLPError errCode,
                    void* pvCookie) {

     if( errCode != SLP_OK ) {
       /* Deal with error. */
     } else {
       /* Parse attributes. */
     }

     return SLP_FALSE;  /* Done! */
   }
```

7.0   **Security Considerations**

Security is handled by IPSEC and is not exposed to API clients. An adversary could delete valid service advertisements, provide false service information and deny UAs knowledge of existing services unless IPSEC is used to secure IP traffic between SLP agents, as described in [8].

8.0 Acknowledgements

The authors would like to thank Don Provan for his pioneering work during the initial stages of the RFC 2614 API definition. The contributions of Matt Peterson, Ira McDonald, and Jim Mayer were invaluable in preparing the current document.

9.0 References

[1] Kempf, J. and Guttman, E., "An API for Service Location," RFC 2614, June, 1999.
[2] Bradner, S., "Key Words for Use in RFCs to Indicate Requirement Levels," BCP 14, RFC 2119, March 1997.
[3] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," RFC 2396, August 1998.
[4] Yergeau, F., "UTF-8, a transformation format of ISO 10646," RFC 2279, January 1998.
[5] Howes, T., "The String Representation of LDAP Search Filters," RFC 2254 December 1997.
[6] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," RFC 2234, November 1997.
[7] Alvestrand, H., "Tags for the Identification of Languages," RFC 1766, March 1995.
[8] Guttman, E., and J. Kempf, "Service Location Protocol, Version 2," draft-guttman-rfc2608bis-01.txt, a work in progress.
[9] Guttman, E., Perkins, C. and J. Kempf, "Service Templates and Service: Schemes," RFC 2609, June 1999.

10.0 Editors' Addresses

Erik Guttman
Sun Microsystems, Inc.
Eichhoelzelstr. 7
74915 Waibstadt
GERMANY
Phone: +49 172 865 5497
Email: Erik.Guttman@Sun.Com

James Kempf
DoCoMo Labs, USA
180 Metro Drive, Suite 300
San Jose, CA, 95430
USA
Phone: +1 408 451 4711
Email: kempf@docomolabs-usa.com

11.0 Full Copyright Statement

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