Network Working Group
Request for Comments: 2614
Category: Informational

J. Kempf
E. Guttman
Sun Microsystems

June 1999

An API for Service Location

Status of This Memo

Abstract

The Service Location Protocol (SLP) provides a new way for clients to dynamically discovery network services. With SLP, it is simple to offer highly available services that require no user configuration or assistance from network administrators prior to use. This document describes standardized APIs for SLP in C and Java. The APIs are modular and are designed to allow implementations to offer just the feature set needed. In addition, standardized file formats for configuration and serialized registrations are defined, allowing SLP agents to set network and other parameters in a portable way. The serialized file format allows legacy services to be registered with SLP directory agents in cases where modifying the legacy service program code is difficult or impossible, and to portably exchange a registration database.

Table of Contents

1. Introduction

The Service Location API is designed for standardized access to the Service Location Protocol (SLP). The APIs allow client and service programs to be be written or modified in a very simple manner to provide dynamic service discovery and selection. Bindings in the C and Java languages are defined in this document. In addition, standardized formats for configuration files and for serialized registration files are presented. These files allow SLP agents to configure network parameters, to register legacy services that have not been SLP enabled, and to portably exchange registration databases.

1.1. Goals

The overall goal of the API is to enable source portability of applications that use the API between different implementations of SLP. The result should facilitate the adoption of SLP, and conversion of clients and service programs to SLP.

The goals of the C binding are to create a minimal but complete access to the functionality of the SLP protocol, allowing for simple memory management and limited code size.

The Java API provides for modular implementations (where unneeded features can be omitted) and an object oriented interface to the complete set of SLP data and functionality.

The standardized configuration file and serialized file formats provide a simple syntax with complete functional coverage of the protocol, but without system dependent properties and secure information.

1.2. 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 [1].

Service Location Protocol (SLP)

The underlying protocol allowing dynamic and scalable service discovery. This protocol is specified in the Service Location Protocol Version 2 [7].

SLP framework

When a 'Service Location framework' is mentioned, it refers to both the SLP implementation and interface implementation; i.e. whatever provides the SLP functionality to user level programs. This includes remote agents.

Directory Agent (DA)

A service that automatically gathers service advertisements from SAs in order to provide them to UAs.

User Agent (UA)

This is the Service Location process or library that allows SLP requests to be made on behalf of a client process. UAs automatically direct requests to DAs when they exist. In their absence, UAs make requests to SAs.

Service Agent (SA)

This is the Service Location process or library that allows service software to register and deregister itself with the SLP framework. SAs respond to UA service requests, detect DAs and register service advertisements with them.

SA Server

Many operating system platforms only allow a single process to listen on a particular port number. Since SAs are required to listen on a multicast address for SLP service requests, implementations of the SLP framework on such platforms that want to support multiple SAs on one machine need to arrange for a single process to do the listening while the advertising SAs communicate with that process through another mechanism. The single listening process is called an SA server. SA servers share many characteristics with DAs, but they are not the same.

Service Advertisement

A URL possibly combined with service attributes. These are made available to UAs by SAs, either directly or via a DA.

Locale

The language localization that applies to strings passed into or returned from the SLP API. The Locale is expressed using a Language Tag [6]. All attribute strings are associated with a

particular locale. The locale is completely orthogonal to the ANSI C locale. The SLP locale is mapped into the Java locale in the Java API.

Service Template

A document that describes the syntax of the URL for a given service type and a definition of all service attributes including the meaning, defaults, and constraints on values the attributes may take. See [8] for more information on service templates.

The service: URL

A service of a particular type announces its availability with a service: URL that includes its service access point (domain name or IP address, and possibly its port number) and optionally basic configuration parameters. The syntax of the service: URL is defined in the service template. Other URL's can be used in service advertisements if desired.

Service Attributes

The attributes associated with a given service. The values that can be assigned to service attributes are defined by the service template.

Scope

A string used to control the availability of service advertisements. Every SLP Agent is configured with one or more scope strings. Scopes are assigned by site administrators to group services for many purposes, but chiefly as a means of scalability. DAs store only services advertised having a scope string matching the scopes with which they are configured.

Naming Authority (NA)

This is a 'suffix' to the service type string. It completely changes the meaning of the service type. NAs are used for private definitions of well known Service Types and experimental Service Type extensions. The default NA is "IANA", which must not be explicitly included. Service types with the IANA naming authority are registered with the Internet Assigned Numbers Authority (see [8] for more information on the registration procedure).

2. File Formats

This section describes the configuration and serialized registration file formats. Both files are defined in the UTF-8 character set [3].

Attribute tags and values in the serialized registration file 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. [7]

In serialized registration files, escaped 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 in the serialized registration file as attribute values. In other cases, whenever an escape is encountered and the character is not an SLP reserved character, an error is signaled.

Escaped characters in URLs in serialized registration files use the URL escape convention. [2].

Property names and values in the configuration file have a few reserved characters that are involved in file's lexical definition. The characters '.' and '=' are reserved in property names and must be escape. The characters ',', '(', and ')' are reserved in property values and must be escaped. In addition, scope names in the net.slp.useScopes 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 in processing the serialized registration file or 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 US-ASCII. 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 [7]), 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 [5] 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
value-list    =  value / value "," value-list
value         =  int / bool /
                 "(" value-list ")" / string
int           =  1*DIGIT
bool          =  "true" / "false" / "TRUE" / "FALSE"
newline       =  CR / ( CRLF )
string        =  1*stringchar
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
```

With the exception of net.slp.useScopes, net.slp.DAAddresses, and net.slp.isBroadcastOnly, all other properties can be changed through property accessors in the C and Java APIs. The property accessors

only change the property values in the running agent program and do not affect the values in the configuration file. The net.slp.useScopes and net.slp.DAAddresses properties are read-only because they control the agent's view of the scopes and DAs and are therefore critical to the function of the API scope discovery algorithm. Attempts to modify them are unlikely to yield productive results, and could harm the ability of the agent to find scopes and use DAs. The net.slp.isBroadcastOnly property is read-only because the API library needs to configure networking upon start up and changing this property might invalidate the configuration. Whether the local network uses broadcast or multicast is not likely to change during the course of the program's execution.

The properties break down into the following subsections describes an area and its properties.

2.1.1. DA configuration

Important configuration properties for DAs are included in this section. These are:

net.slp.isDA

A boolean indicating if the SLP server is to act as a DA. If false, not run as a DA. Default is false.

net.slp.DAHeartBeat

A 32 bit integer giving the number of seconds for the DA heartbeat. Default is 3 hours (10800 seconds). This property corresponds to the protocol specification parameter CONFIG_DA_BEAT [7]. Ignored if isDA is false.

net.slp.DAAttributes

A comma-separated list of parenthesized attribute/value list pairs that the DA must advertise in DAAdverts. The property must be in the SLP attribute list wire format, including escapes for reserved characters. [7]

2.1.2. Static Scope Configuration

These properties allow various aspects of scope handling to be configured.

net.slp.useScopes

A value-list of strings indicating the only scopes a UA or SA is allowed to use when making requests or registering, or the scopes a DA must support. If not present for the DA and SA, then in the absence of scope information from DHCP, the default scope "DEFAULT" is used. If not present for the UA, and there is no scope information available from DHCP, then the user scoping model is in force. Active and passive DA discovery or SA discovery are used for scope discovery, and the scope "DEFAULT" is used if no other information is available. If a DA or SA gets another scope in a request, a SCOPE_NOT_SUPPORTED error should be returned, unless the request was multicast, in which case it should be dropped. If a DA gets another scope in a registration, a SCOPE_NOT_SUPPORTED error must be returned. Unlike other properties, this property is "read-only", so attempts to change it after the configuration file has been read are ignored. See Section 3.12 for the algorithm the API uses in determining what scope information to present.

net.slp.DAAddresses

A value-list of IP addresses or DNS resolvable host names giving the SLPv2 DAs to use for statically configured UAs and SAs. Ignored by DAs (unless the DA is also an SA server). Default is none. Unlike other properties, this property is "read-only", so attempts to change it after the configuration file has been read are ignored.

The following grammar describes the property:

```
addr-list     =  addr / addr "," addr-list
addr          =  fqdn / hostnumber
fqdn          =  ALPHA / ALPHA *[ anum / "-" ] anum
anum          =  ALPHA / DIGIT
hostnumber    =  1*3DIGIT 3("." 1*3DIGIT)
```

An example is:

```
sawah,mandi,sambal
```

IP addresses can be used instead of host names in networks where DNS is not deployed, but network administrators are reminded that using IP addresses will complicate machine

renumbering, since the SLP configuration property files in statically configured networks will have to be changed. Similarly, if host names are used, implementors must be careful that a name service is available before SLP starts, in other words, SLP cannot be used to find the name service.

2.1.3. Tracing and Logging

This section allows tracing and logging information to be printed by the various agents.

net.slp.traceDATraffic

A boolean controlling printing of messages about traffic with DAs. Default is false.

net.slp.traceMsg

A boolean controlling printing of details on SLP messages. The fields in all incoming messages and outgoing replies are printed. Default is false.

net.slp.traceDrop

A boolean controlling printing details when a SLP message is dropped for any reason. Default is false.

net.slp.traceReg

A boolean controlling dumps of all registered services upon registration and deregistration. If true, the contents of the DA or SA server are dumped after a registration or deregistration occurs. Default is false.

2.1.4. Serialized Proxy Registrations

These properties control the reading and writing of serialized registrations.

net.slp.serializedRegURL

A string containing a URL pointing to a document containing serialized registrations that should be processed when the DA or SA server starts up. Default is none.

2.1.5. Network Configuration Properties

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

net.slp.isBroadcastOnly

A boolean indicating if broadcast should be used instead of multicast. Like the net.slp.useScopes and net.slp.DAAddresses properties, this property is "read-only", so attempts to change it after the configuration file has been read are ignored. Default is false.

net.slp.passiveDADetection

A boolean indicating whether passive DA detection should be used. Default is true.

net.slp.multicastTTL

A positive integer less than or equal to 255, giving the multicast TTL. Default is 255.

net.slp.DAActiveDiscoveryInterval

A 16 bit positive integer giving the number of seconds between DA active discovery queries. Default is 900 seconds (15 minutes). This property corresponds to the protocol specification parameter CONFIG_DA_FIND [7]. If the property is set to zero, active discovery is turned off. This is useful when the DAs available are explicitly restricted to those obtained from DHCP or the net.slp.DAAddresses property.

net.slp.multicastMaximumWait

A 32 bit integer giving the maximum amount of time to perform multicast, in milliseconds. Default is 15000 ms (15 sec.). This property corresponds to the CONFIG_MC_MAX parameter in the protocol specification [7].

net.slp.multicastTimeouts

A value-list of 32 bit integers used as timeouts, in milliseconds, to implement the multicast convergence algorithm. Each value specifies the time to wait before sending the next request, or until nothing new has been learned from two successive requests. Default is: 3000,3000,3000,3000,3000. In a fast network the

aggressive values of 1000,1250,1500,2000,4000 allow better performance. This property corresponds to the CONFIG_MC_RETRY parameter in the protocol specification [7]. Note that the net.slp.DADiscoveryTimeouts property must be used for active DA discovery.

net.slp.DADiscoveryTimeouts

A value-list of 32 bit integers used as timeouts, in milliseconds, to implement the multicast convergence algorithm during 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. This property corresponds to the protocol specification parameter CONFIG_RETRY [7]. Default is: 2000,2000,2000,2000,3000,4000.

net.slp.datagramTimeouts

A value-list of 32 bit integers used as timeouts, in milliseconds, to implement unicast datagram transmission to DAs. 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 is the protocol specification property CONFIG_RETRY_MAX [7].

net.slp.randomWaitBound

A 32 bit integer giving the maximum value for all random wait parameters, in milliseconds. Default is 1000 (1 sec.). This value corresponds to the protocol specification parameters CONFIG_START_WAIT, CONFIG_REG_PASSIVE, and CONFIG_REG_ACTIVE [7].

net.slp.MTU

A 16 bit integer giving the network packet MTU, in bytes. This is the maximum size of any datagram to send, but the implementation might receive a larger datagram. The maximum size includes IP, and UDP or TCP headers. Default is 1400.

net.slp.interfaces

Value-list of strings giving the IP addresses of network interfaces on which the DA or SA should listen on port 427 for multicast, unicast UDP, and TCP messages. Default is empty, i.e. use the default network interface. The grammar for this property is:

```
addr-list     =  hostnumber / hostnumber "," addr-list
hostnumber    =  1*3DIGIT 3("." 1*3DIGIT)
```

An example is:

195.42.42.42,195.42.142.1,195.42.120.1

The example machine has three interfaces on which the DA should listen.

Note that since this property only takes IP addresses, it will need to be changed if the network is renumbered.

2.1.6. SA Configuration

This section contains configuration properties for the SA. These properties are typically set programmatically by the SA, since they are specific to each SA.

net.slp.SAAttributes

A comma-separated list of parenthesized attribute/value list pairs that the SA must advertise in SAAdverts. The property must be in the SLP attribute list wire format, including escapes for reserved characters. [7]

2.1.7. UA Configuration

This section contains configuration properties for the UA. These properties can be set either programmatically by the UA or in the configuration file.

net.slp.locale

A RFC 1766 Language Tag [6] for the language locale. Setting this property causes the property value to become the default locale for SLP messages. Default is "en". This property is also used for SA and DA configuration.

net.slp.maxResults

A 32 bit integer giving the maximum number of results to accumulate and return for a synchronous request before the timeout, or the maximum number of results to return through a callback if the request results are reported asynchronously.

Positive integers and -1 are legal values. If -1, indicates that all results should be returned. Default value is -1.

DAs and SAs always return all results that match the request. This configuration value applies only to UAs, that filter incoming results and only return as many values as net.slp.maxResults indicates.

net.slp.typeHint

A value-list of service type names. In the absence of any DAs, UAs perform SA discovery for finding scopes. These SA discovery requests may contain a request for service types as an attribute.

The API implementation will use the service type names supplied by this property to discover only those SAs (and their scopes) which support the desired service type or types. For example, if net.slp.typeHint is set to "service:imap,service:pop3" then SA discovery requests will include the search filter:

(|(service-type=service:imap)(service-type=service:pop3))

The API library can also use unicast to contact the discovered SAs for subsequent requests for these service types, to optimize network access.

2.1.8. Security

The property in this section allows security for all agents to be set on or off. When the property is true, then the agent must include security information on all SLP messages transacted by that agent. Since security policy must be set network wide to be effective, a single property controls security for all agents. Key management and management of SLP SPI strings [7] are implementation and policy dependent.

net.slp.securityEnabled

A boolean indicating whether the agent should enable security for URLs, attribute lists, DAAdverts, and SAAdverts. Each agent is responsible for interpreting the property appropriately. Default is false.

2.2. Multihomed Machines

On multihomed machines, the bandwidth and latency characteristics on different network interfaces may differ considerably, to the point where different configuration properties are necessary to achieve optimal performance. 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 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 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 radius, 255.

The net.slp.interfaces property must only be set if there is no routing between the interfaces. If the property is set, the DA (if any) and SAs should advertise with the IP address or host name appropriate to the interface on the interfaces in the list. If packets are routed between the interfaces, then the DA and SAs should only advertise on the default interface. The property should also be set if broadcast is used rather than multicast on the subnets connected to the interfaces. 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 when multicast packets are routed 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.

2.3. Serialized Registration File

The serialized registration file contains a group of registrations that a DA or SA server (if one exists) registers 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 required to be UTF-8.

The syntax of the serialized registration file, in ABNF format [5], 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 [slist] [attr-list] newline
   url-props     =  surl "," lang "," ltime [ "," type ] newline
   surl          =  ;The registration's URL. See
                    ; [8] for syntax.
   lang          =  1*8ALPHA [ "-" 1*8ALPHA ]
                    ;RFC 1766 Language Tag see [6].
   ltime         =  1*5DIGIT
                    ; A positive 16-bit integer
                    ; giving the lifetime
                    ; of the registration.
   type          =  ; The service type name, see [7]
                    ; and [8] for syntax.
   slist         =  "scopes" "=" scope-list newline
   scope-list    =  scope-name / scope-name "," scope-list
   scope         =  ; See grammar of [7] for
                    ; scope-name 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 [7] for syntax.
   attr-val-list =  attr-val / attr-val "," attr-val-list
   attr-val      =  ;Attribute value, see [7] 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 scope names, attribute tags, and attribute values requires escapes for special characters as specified in [7]. 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 present for a service: URL, a warning is signaled and the type name is ignored. If the maximum lifetime is specified (65535 sec.), the registration is taken to be permanent, and is continually refreshed by the DA or SA server until it exits. Scopes can be included in a registration by including an attribute definition with tag "scopes" followed by a comma separated list of scope names immediately after the url-props production. If the optional scope list is present, the registrations are made in the indicated scopes; otherwise, they are registered in the scopes with which the DA or SA server was configured through the net.slp.useScopes property.

If the scope list contains scopes that are not in the net.slp.useScopes property (provided that property is set) or are not specified by DHCP, the API library should reject the registration and issue a warning message.

2.4. Processing Serialized Registration and Configuration Files

Implementations are encouraged to make processing of configuration and serialized files as transparent as possible to clients of the API. At the latest, errors must be caught when the relevant configuration item is used. At the earliest, errors may be caught when the relevant file is loaded into the executing agent. Errors should be reported by logging to the appropriate platform logging file, error output, or log device, and the default value substituted. Serialized registration file entries should be caught and reported when the file is loaded.

Configuration file loading must be complete prior to the initiation of the first networking connection. Serialized registration must be complete before the DA accepts the first network request.

3. Binding Independent Implementation Considerations

This section discusses a number of implementation considerations independent of language binding, with language specific notes where applicable.

3.1. Multithreading

Implementations of both the C and Java APIs are required to make API calls thread-safe. Access to data structures shared between threads must be co-ordinated to avoid corruption or invalid access. One way to achieve this goal is to allow only one thread at a time in the implementing library. Performance in such an implementation suffers, however. Therefore, where possible, implementations are encouraged to allow multiple threads within the SLP API library.

3.2. Asynchronous and Incremental

The APIs are designed to encourage implementations supporting asynchronous and incremental client interaction. The goal is to allow large numbers of returned service URLs, service types, and attributes without requiring the allocation of huge chunks of memory. The particular design features to support this goal differ in the two language bindings.

3.3. Type Checking for Service Types

Service templates [8] allow SLP registrations to be type checked for correctness. Implementations of the API are free to make use of service type information for type checking, but are not required to do so. If a type error occurs, the registration should terminate with TYPE_ERROR.

3.4. Refreshing Registrations

SLP advertisements carry an explicit lifetime with them. 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. Implementations of the SA API must provide this automatic refreshing capability. Note that a client which uses 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 will time out in the DA eventually.

3.5. Configuration File Processing

DAs, SAs and UAs processing the configuration file, and DAs and SA servers processing the serialized registration file are required to log any errors using whatever underlying error mechanism is appropriate for the platform. Examples include writing error messages to the standard output, writing to a system logging device, or displaying the errors to a logging window. After the error is reported, the offending property must be set to the default and program execution continued. An agent MUST NOT fail if a file format error occurs.

3.6. Attribute Types

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 [8] in order to convert from the string representation on the wire to an object typed appropriately.

3.7. Removal of Duplicates

The UA implementation SHOULD always collate results to remove duplicates during synchronous operations and for the Java API. During asynchronous operation in C, the UA implementation SHOULD forgo duplicate elimination to reduce memory requirements in the library. This allows the API library to simply take the returned attribute value list strings, URL strings, or service type list strings and call the callback function with it, without any additional processing. Naturally, the burden of duplicate elimination is thrown onto the client in this case.

3.8. Character Set Encoding

Character string parameters in the Java API are all represented in Unicode internally because that is the Java-supported character set. Characters buffer parameters in the C API are represented in UTF-8 to maintain maximum compatibility on platforms that only support US-ASCII and not UTF-8. API functions are still required to handle the full range of 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. Inside URLs, characters that are not allowed by URL syntax [2] 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 C or Java API may exceed 64K bytes in length.

3.9. Error Semantics

All errors encountered processing SLP messages should be logged. For synchronous calls, an error is only reported on a call if no successful replies were received from any SLP framework entity. If an error occurred among one of several successful replies, then the error should be logged and the successful replies returned. For asynchronous calls, an error occurring during correspondence with a

particular remote SLP agent is reported through the first callback (in the C API) or enumeration method invocation (in the Java API) after the error occurs, which would normally report the results of the correspondence. This allows the callback or client code to determine whether the operation should be terminated or continue. In some cases, the error returned from the SLP framework may be fatal (SLP_PARSE_ERROR, etc.). In these cases, the API library terminates the operation.

Both the Java and C APIs contain language specific error code mechanisms for returning error information. The names of the error codes are consistent between the two implementations, however.

The following error codes are returned from a remote agent (DA or SA server):

LANGUAGE_NOT_SUPPORTED

No DA or SA has service advertisement or attribute information in the language requested, but at least one DA or SA indicated, via the LANGUAGE_NOT_SUPPORTED error code, that it might have information for that service in another language.

PARSE_ERROR

The SLP message was rejected by a remote SLP agent. The API returns this error only when no information was retrieved, and at least one SA or DA indicated a protocol error. The data supplied through the API may be malformed or a may have been damaged in transit.

INVALID_REGISTRATION

The API may return this error if an attempt to register a service was rejected by all DAs because of a malformed URL or attributes. SLP does not return the error if at least one DA accepted the registration.

AUTHENTICATION_ABSENT

If the SLP framework supports authentication, this error arises when the UA or SA failed to send an authenticator for requests or registrations in a protected scope.

INVALID_UPDATE

An update for a non-existing registration was issued, or the update includes a service type or scope different than that in the initial registration, etc.

The following errors result from interactions with remote agents or can occur locally:

AUTHENTICATION_FAILED

If the SLP framework supports authentication, this error arises when a authentication on an SLP message failed.

SCOPE_NOT_SUPPORTED

The API returns this error if the SA has been configured with net.slp.useScopes value-list of scopes and the SA request did not specify one or more of these allowable scopes, and no others. It may be returned by a DA or SA if the scope included in a request is not supported by the DA or SA.

REFRESH_REJECTED

The SA attempted to refresh a registration more frequently than the minimum refresh interval. The SA should call the appropriate API function to obtain the minimum refresh interval to use.

The following errors are generated through a program interacting with the API implementation. They do not involve a remote SLP agent.

NOT_IMPLEMENTED

If an unimplemented feature is used, this error is returned.

NETWORK_INIT_FAILED

If the network cannot initialize properly, this error is returned.

NETWORK_TIMED_OUT

When no reply can be obtained in the time specified by the configured timeout interval for a unicast request, this error is returned.

NETWORK_ERROR

The failure of networking during normal operations causes this error to be returned.

BUFFER_OVERFLOW

An outgoing request overflowed the maximum network MTU size. The request should be reduced in size or broken into pieces and tried again.

MEMORY_ALLOC_FAILED

If the API fails to allocate memory, the operation is aborted and returns this.

PARAMETER_BAD

If a parameter passed into an interface is bad, this error is returned.

INTERNAL_SYSTEM_ERROR

A basic failure of the API causes this error to be returned. This occurs when a system call or library fails. The operation could not recover.

HANDLE_IN_USE

In the C API, callback functions are not permitted to recursively call into the API on the same SLPHandle, either directly or indirectly. If an attempt is made to do so, this error is returned from the called API function.

TYPE_ERROR

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

Some error codes are handled differently in the Java API. These differences are discussed in Section 5.

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.

3.10. Modular Implementations

Subset implementations that do not support the full range of functionality are required to nevertheless 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 should be returned. The Java API has some additional conventions for handling subsets. Applications that are expected to run on a wide variety of platforms should be prepared for subset API implementations by checking returned error codes.

3.11. Handling Special Service Types

The service types service:directory-agent and 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 services of type service:service-agent and with its URL only to a request for services of type service:directory-agent. The UA would therefore not obtain a complete picture of the available DAs and SAs.

3.12. Scope Discovery and Handling

Both APIs contain an operation to obtain a list of currently known scope names. This scope information comes from a variety of places: DHCP, the net.slp.useScopes property, unicast to DAs configured via DHCP or the net.slp.DAAddresses property, and active and passive discovery.

The API is required to be implemented in a way that re-enforces the administrative and user scoping models described in [7]. SA clients only support the administrative scoping model. SAs must know a priori what DAs they need to register with since there is typically no human intervention in scope selection for SAs. UAs must support both administrative and user scoping because an application may require human intervention in scope selection.

API implementations are required to support administrative scoping in the following way. Scopes configured by DHCP and scopes of DAs configured by DHCP have first priority (in that order) and must be returned if they are available. The net.slp.useScopes property has second priority, and scopes discovered through the net.slp.useScopes property must be returned if this property is set and there are no scopes available from DHCP. If scopes are not available from either of these sources and the net.slp.DAAddresses property is set, then the scopes available from the configured DAs must be returned. Note that if both DAs and scopes are configured, the scopes of the configured DAs must match the configured scope list; otherwise and error is signaled and agent execution is terminated. If no configured scope information is available, then an SA client has default scope, "DEFAULT", and a UA client employs user scoping.

User scoping is supported in the following way. Scopes discovered from active DA discovery, and from passive DA discovery all must be returned. If no information is available from active and passive DA discovery, then the API library may perform SA discovery, using the service types in the net.slp.typeHint property to limit the search to SAs supporting particular service types. If no net.slp.typeHint property is set, the UA may perform SA discovery without any service type query. In the absence of any of the above sources of information, the API must return the default scope, "DEFAULT". Note that the API must always return some scope information.

SLP requires that SAs must perform their operations in all scopes currently known to them. [7]. The API enforces this constraint by not requiring the API client to supply any scopes as parameters to API operations. The API library must obtain all currently known scopes and use them in SA operations. UA API clients should use a scope obtained through one of the API operations for finding scopes. Any other scope name may result in a SCOPE_NOT_SUPPORTED error from a remote agent. The UA API library can optionally check the scope and return the error without contacting a remote agent.

4. C Language Binding

The C language binding presents a minimal overhead implementation that maps directly into the protocol. There is one C language function per protocol request, with the exception of the SLPDereg() and SLPDelAttrs() functions, which map into different uses of the SLP deregister request. Parameters are for the most part character buffers. Memory management is kept simple by having the client allocate most memory and requiring that client callback functions copy incoming parameters into memory allocated by the client code. Any memory returned directly from the API functions is deallocated using the SLPFree() function.

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 UTF-8 but they may still be passed as a C string (a null terminated sequence of bytes.) Escaped characters must be encoded by the API client as UTF-8. In the common case of US-ASCII, the usual one byte per character C strings work. API functions assist in escaping and unescaping strings.

Unless otherwise noted, parameters to API functions and callbacks are non-NULL. Some parameters may have other restrictions. If any parameter fails to satisfy the restrictions on its value, the operation returns a PARAMETER_BAD error.

4.1. Constant Types

4.1.1. URL Lifetimes

4.1.1.1. Synopsis

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

4.1.1.2. Description

The SLPURLLifetime enum type contains URL lifetime values, in seconds, that are frequently used. 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.

4.1.2. Error Codes

4.1.2.1. 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_AUTHENTICATION_ABSENT     = -6,
  SLP_AUTHENTICATION_FAILED     = -7,
```

```
     SLP_INVALID_UPDATE               = -13,
     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 ;
```

4.1.2.2. Description

The SLPError enum contains error codes that are returned from API functions.

The SLP_OK code indicates that the no error occurred during the operation.

The SLP_LAST_CALL code is passed to callback functions 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. On the last call of a callback during both a synchronous and asynchronous call, the error code parameter has value SLP_LAST_CALL, and the other parameters are all NULL. If no results are returned by an API operation, then only one call is made, with the error parameter set to SLP_LAST_CALL.

4.1.3. SLPBoolean

4.1.3.1. Synopsis

```
   typedef enum {
      SLP_FALSE = 0,
      SLP_TRUE = 1

   } SLPBoolean;
```

4.1.3.2. Description

The SLPBoolean enum is used as a boolean flag.

4.2. Struct Types

4.2.1. SLPSrvURL

4.2.1.1. Synopsis

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

4.2.1.2. Description

The SLPSrvURL structure is filled in by the SLPParseSrvURL() function with information parsed from a character buffer containing a service URL. 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. [7]

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.

4.2.2. SLPHandle

4.2.2.1. Synopsis

```
typedef void* SLPHandle;
```

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.

4.3. Callbacks

A function pointer to a callback function specific to a particular API operation is included in the parameter list when the API function is invoked. The callback function is called with the results of the operation in both the synchronous and asynchronous cases. The memory included in the callback parameters is owned by the API library, and the client code in the callback must copy out the contents if it wants to maintain the information longer than the duration of the current callback call.

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 that starts the operation by invoking the API function to pass information down to the callback without using global variables. 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).

4.3.1. SLPRegReport

4.3.1.1. Synopsis

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

4.3.1.2. Description

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

4.3.1.3. Parameters

hSLP

The SLPHandle used to initiate the operation.

errCode

An error code indicating if an error occurred during the operation.

pvCookie

Memory passed down from the client code that called the original API function, starting the operation. May be NULL.

4.3.2. SLPSrvTypeCallback

4.3.2.1. Synopsis

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

4.3.2.2. Description

The SLPSrvTypeCallback type is the type of the callback function parameter to SLPFindSrvTypes() function. If the hSLP handle parameter was opened asynchronously, the results returned through the callback MAY be uncollated. If the hSLP handle parameter was opened synchronously, then the returned results must be collated and duplicates eliminated.

4.3.2.3. Parameters

hSLP

The SLPHandle used to initiate the operation.

pcSrvTypes

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

errCode

An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than SLP_OK, then the API library may choose to terminate the outstanding operation.

pvCookie

Memory passed down from the client code that called the original API function, starting the operation. May be NULL.

4.3.2.4. Returns

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

4.3.3. SLPSrvURLCallback

4.3.3.1. Synopsis

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

4.3.3.2. Description

The SLPSrvURLCallback type is the type of the callback function parameter to SLPFindSrvs() function. If the hSLP handle parameter was opened asynchronously, the results returned through the callback MAY be uncollated. If the hSLP handle parameter was opened synchronously, then the returned results must be collated and duplicates eliminated.

4.3.3.3. 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

An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than SLP_OK, then the API library may choose to terminate the outstanding operation.

pvCookie

Memory passed down from the client code that called the original API function, starting the operation. May be NULL.

4.3.3.4. Returns

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

4.3.4. SLPAttrCallback

4.3.4.1. Synopsis

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

4.3.4.2. Description

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

The behavior of the callback differs depending on whether the attribute request was by URL or by service type. If the SLPFindAttrs() operation was originally called with a URL, the callback is called once regardless of whether the handle was opened asynchronously or synchronously. The pcAttrList parameter contains the requested attributes as a comma separated list (or is empty if no attributes matched the original tag list).

If the SLPFindAttrs() operation was originally called with a service type, the value of pcAttrList and calling behavior depend on whether the handle was opened asynchronously or synchronously. If the handle was opened asynchronously, the callback is called every time the API library has results from a remote agent. The pcAttrList parameter MAY be uncollated between calls. It contains a comma separated list with the results from the agent that immediately returned results. If the handle was opened synchronously, the results must be collated from all returning agents and the callback is called once, with the pcAttrList parameter set to the collated result.

4.3.4.3. 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; i.e. "(attr-id=attr-value-list)" [7].

errCode

An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than SLP_OK, then the API library may choose to terminate the outstanding operation.

pvCookie

Memory passed down from the client code that called the original API function, starting the operation. May be NULL.

4.3.4.4. Returns

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

4.4. Opening and Closing an SLPHandle

4.4.1. SLPOpen

4.4.1.1. Synopsis

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

4.4.1.2. Description

Returns a SLPHandle handle in the phSLP parameter for the language locale passed in as the pcLang parameter. The client indicates if operations on the handle are to be synchronous or asynchronous through the isAsync parameter. 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. Upon failure, the phSLP parameter is NULL.

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. The SLPClose() API function terminates any outstanding calls on the handle. If an implementation is unable to support a asynchronous(resp. synchronous) operation, due to memory constraints or lack of threading support, the SLP_NOT_IMPLEMENTED flag may be returned when the isAsync flag is

SLP_TRUE (resp. SLP_FALSE).

4.4.1.3. Parameters

pcLang

A pointer to an array of characters containing the RFC 1766 Language Tag [6] for the natural language locale of requests and registrations issued on the handle.

isAsync

An 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.

4.4.2. SLPClose

4.4.2.1. Synopsis

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

4.4.2.2. 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 so their callback functions will not be called any further.

4.4.2.3. Parameters

SLPHandle

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

4.5. Protocol API

4.5.1. SLPReg

4.5.1.1. Synopsis

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

4.5.1.2. 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. If the fresh flag is SLP_TRUE, then the registration is new (the SLP protocol FRESH flag is set) and the registration replaces any existing registrations. The pcSrvType parameter is a service type name and can be included for service URLs that are not in the service: scheme. If the URL is in the service: scheme, the pcSrvType parameter is ignored. If the fresh flag is SLP_FALSE, then an existing registration is updated. Rules for new and updated registrations, and the format for pcAttrs and pcScopeList can be found in [7]. 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.

4.5.1.3. Parameters

hSLP

The language specific SLPHandle on which to register the advertisement.

pcSrvURL

The URL to register. May not be the empty string.

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 pURL is a service: URL, then this parameter is ignored.

pcAttrs

A comma separated list of attribute assignment expressions for the attributes of the advertisement. Use empty string, "" for no attributes.

fresh

An SLPBoolean that is SLP_TRUE if the registration is new or SLP_FALSE if a reregistration.

callback

A callback to report the operation completion status.

pvCookie

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

4.5.1.4. Returns

If an error occurs in starting the operation, one of the SLPError codes is returned.

4.5.2. SLPDereg

4.5.2.1. Synopsis

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

4.5.2.2. 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.

4.5.2.3. Parameters

hSLP

The language specific SLPHandle to use for deregistering.

pcURL

The URL to deregister. May not be the empty string.

callback

A callback to report the operation completion status.

pvCookie

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

4.5.2.4. Returns

If an error occurs in starting the operation, one of the SLPError codes is returned.

4.5.3. SLPDelAttrs

4.5.3.1. Synopsis

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

4.5.3.2. Description

Delete the selected attributes in the locale of the SLPHandle. The API library is required to perform the operation in all scopes obtained through configuration.

4.5.3.3. Parameters

hSLP

The language specific SLPHandle to use for deleting attributes.

pcURL

The URL of the advertisement from which the attributes should be deleted. May not be the empty string.

pcAttrs

A comma separated list of attribute ids for the attributes to deregister. See Section 9.8 in [7] for a description of the list format. May not be the empty string.

callback

A callback to report the operation completion status.

pvCookie

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

4.5.3.4. Returns

If an error occurs in starting the operation, one of the SLPError codes is returned.

4.5.4. SLPFindSrvTypes

4.5.4.1. Synopsis

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

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 the empty string, i.e. "", 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. [7] See [8] for more information on the syntax of service type names.

4.5.4.2. Parameters

hSLP

The SLPHandle on which to search for types.

pcNamingAuthority

The naming authority to search. Use "*" for all naming authorities and the empty string, "", for the default naming authority.

pcScopeList

A pointer to a char containing comma separated list of scope names to search for service types. May not be the empty 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.

4.5.4.3. Returns

If an error occurs in starting the operation, one of the SLPError codes is returned.

4.5.5. SLPFindSrvs

4.5.5.1. Synopsis

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

4.5.5.2. Description

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

4.5.5.3. Parameters

hSLP

The language specific SLPHandle on which to search for services.

pcServiceType

The Service Type String, including authority string if any, for the request, such as can be discovered using SLPSrvTypes(). This could be, for example "service:printer:lpr" or "service:nfs". May not be the empty string.

pcScopeList

A pointer to a char containing comma separated list of scope names. May not be the empty string, "".

pcSearchFilter

A query formulated of attribute pattern matching expressions in the form of a LDAPv3 Search Filter, see [4]. If this filter is empty, i.e. "", all services of the requested type in the

specified scopes are returned.

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.

4.5.5.4. Returns

If an error occurs in starting the operation, one of the SLPError codes is returned.

4.5.6. SLPFindAttrs

4.5.6.1. Synopsis

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

4.5.6.2. Description

This function returns service attributes matching the attribute ids for the indicated service URL or service type. If pcURLOrServiceType is a service URL, the attribute information returned is for that particular advertisement in the language locale of the SLPHandle.

If pcURLOrServiceType is a service type name (including naming authority if any), then the attributes for all advertisements of that service type are returned regardless of the language of registration. Results are returned through the callback.

The result is filtered with an SLP attribute request filter string parameter, the syntax of which is described in [7]. If the filter string is the empty string, i.e. "", all attributes are returned.

4.5.6.3. Parameters

hSLP

The language specific SLPHandle on which to search for attributes.

pcURLOrServiceType

The service URL or service type. See [7] for URL and service type syntax. May not be the empty string.

pcScopeList

A pointer to a char containing a comma separated list of scope names. May not be the empty string, "".

pcAttrIds

The filter string indicating which attribute values to return. Use empty string, "", to indicate all values. Wildcards matching all attribute ids having a particular prefix or suffix are also possible. See [7] 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.

4.5.6.4. Returns

If an error occurs in starting the operation, one of the SLPError codes is returned.

4.6. Miscellaneous Functions

4.6.1. SLPGetRefreshInterval

4.6.1.1. Synopsis

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

4.6.1.2. 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, assures that no refresh registration will be rejected. If no DA advertises a min-refresh-interval attribute, a value of 0 is returned.

4.6.1.3. 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.

4.6.2. SLPFindScopes

4.6.2.1. Synopsis

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

4.6.2.2. Description

Sets ppcScopeList parameter to a pointer to a comma separated list including all available scope values. The list of scopes comes from a variety of sources: the configuration file's net.slp.useScopes property, unicast to DAs on the net.slp.DAAddresses property, DHCP, or through the DA discovery process. 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".

4.6.2.3. Parameters

hSLP

The SLPHandle on which to search for scopes.

ppcScopeList

A pointer to char pointer into which the buffer pointer is placed upon return. The buffer is null terminated. The memory should be freed by calling SLPFree().

4.6.2.4. Returns

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

4.6.3. SLPParseSrvURL

4.6.3.1. Synopsis

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

4.6.3.2. Description

Parses the URL passed in as the argument into a service URL structure and returns it in the ppSrvURL pointer. If a parse error occurs, returns SLP_PARSE_ERROR. The input buffer pcSrvURL is 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 SLPFreeURL(). 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.

4.6.3.3. Parameters

pcSrvURL

A pointer to a character buffer containing the null terminated URL string to parse. It is destructively modified to produce the output structure.

ppSrvURL

A pointer to a pointer for the SLPSrvURL structure to receive the parsed URL. The memory should be freed by a call to SLPFree() when no longer needed.

4.6.3.4. Returns

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

4.6.4. SLPEscape

4.6.4.1. Synopsis

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

4.6.4.2. 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.

4.6.4.3. Parameters

pcInbuf

Pointer to he input buffer to process for escape characters.

ppcOutBuf

Pointer to a pointer for the output 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.

4.6.4.4. 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.

4.6.5. SLPUnescape

4.6.5.1. Synopsis

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

4.6.5.2. 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 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.

4.6.5.3. Parameters

pcInbuf

Pointer to he input buffer to process for escape characters.

ppcOutBuf

Pointer to a pointer for the output 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.

4.6.5.4. 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.

4.6.6. SLPFree

4.6.6.1. Synopsis

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

4.6.6.2. Description

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

4.6.6.3. Parameters

pvMem

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

4.6.7. SLPGetProperty

4.6.7.1. Synopsis

```
const char* SLPGetProperty(const char* pcName);
```

4.6.7.2. Description

Returns the value of the corresponding SLP property name. The returned string is owned by the library and MUST NOT be freed.

4.6.7.3. Parameters

pcName

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

4.6.7.4. Returns

If no error, returns a pointer to a character buffer containing the property value. If the property was not set, returns the default value. If an error occurs, returns NULL. The returned string MUST NOT be freed.

4.6.8. SLPSetProperty

4.6.8.1. Synopsis

```
void SLPSetProperty(const char *pcName,
                    const char *pcValue);
```

4.6.8.2. Description

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

4.6.8.3. Parameters

pcName

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

pcValue

Null terminated string with the property value, in UTF-8 character encoding.

4.7. Implementation Notes

4.7.1. Refreshing Registrations

Clients indicate that they want URLs to be automatically refreshed by setting the usLifetime parameter in the SLPReg() function call to SLP_LIFETIME_MAXIMUM. This will cause the API implementation to refresh the URL before it times out. Although using SLP_LIFETIME_MAXIMUM to designate automatic reregistration means that a transient URL can't be registered for the maximum lifetime, little hardship is likely to occur, since service URL lifetimes are measured in seconds and the client can simply use a lifetime of SLP_LIFETIME_MAXIMUM - 1 if a transient URL near the maximum lifetime is desired. API implementations MUST provide this facility.

4.7.2. Syntax for String Parameters

Query strings, attribute registration lists, attribute deregistration lists, scope lists, and attribute selection lists follow the syntax described in [7] 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.

4.7.3. Client Side Syntax Checking

Client side API implementations may do syntax checking of scope names, naming authority names, and service type names, but are not required to do so. 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.

4.7.4. System Properties

The system 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.

4.7.5. Memory Management

The only API functions returning memory specifically requiring deallocation on the part of the client are SLPParseSrvURL(), SLPFindScopes(), SLPEscape(), and SLPUnescape(). This memory should be freed using SLPFree() when no longer needed. Character strings returned via the SLPGetProperty() function should NOT be freed, they are owned by the SLP library.

Memory passed to callbacks belongs to the library and MUST NOT be retained 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.

4.7.6. Asynchronous and Incremental Return Semantics

If a handle parameter to an API function was opened asynchronously, API function calls on the handle check the other parameters, open the appropriate operation and return immediately. In an error occurs in the process of starting the operation, an error code is returned. If the handle parameter was opened synchronously, the API function call blocks until all results are available, and returns only after the results are reported through the callback function. The return code indicates whether any errors occurred both starting and during the operation.

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 has the option of terminating 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. On the other hand, it means that 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.

On the last call to a callback, whether asynchronous or synchronous, the status code passed to the callback has value SLP_LAST_CALL. There are four reasons why the call can terminate:

DA reply received

A reply from a DA has been received and therefore nothing more is expected.

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 giving up on that (as an optimization).

Maximum results

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

4.8. 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! */

}
```

5. Java Language Binding

5.1. Introduction

The Java API is designed to model the various SLP entities in classes and objects. APIs are provided for SA, UA, and service type template access capabilities. The ServiceLocationManager class contains methods that return instances of objects implementing SA and UA capability. Each of these is modeled in an interface. The Locator interface provides UA capability and the Advertiser interface provides SA capability. The TemplateRegistry abstract class contains methods that return objects for template introspection and attribute type checking. The ServiceURL, ServiceType, and ServiceLocationAttribute classes model the basic SLP concepts. A concrete subclass instance of TemplateRegistry is returned by a class method.

All SLP classes and interfaces are located within a single package. The package name should begin with the name of the implementation and conclude with the suffix "slp". Thus, the name for a hypothetical implementation from the University of Michigan would look like:

edu.umich.slp

This follows the Java convention of prepending the top level DNS domain name for the organization implementing the package onto the organization's name and using that as the package prefix.

5.2. Exceptions and Errors

Most parameters to API methods are required to be non-null. The API description indicates if a null parameter is acceptable, or if other restrictions constrain a parameter. When parameters are checked for validity (such as not being null) or their syntax is checked, an error results in the RuntimeException subclass IllegalArgumentException being thrown. Clients of the API are reminded that IllegalArgumentException, derived from RuntimeException, is unchecked by the compiler. Clients should thus be careful to include try/catch blocks for it if the relevant parameters could be erroneous.

Standard Java practice is to encode every exceptional condition as a separate subclass of Exception. Because of the relatively high cost in code size of Exception subclasses, the API contains only a single Exception subclass with different conditions being determined by an integer error code property. A subset, appropriate to Java, of the error codes described in Section 3 are available as constants on the ServiceLocationException class. The subset excludes error codes such

as MEMORY_ALLOC_FAILED.

5.2.1. Class ServiceLocationException

5.2.1.1. Synopsis

```
public class ServiceLocationException
extends Exception
```

5.2.1.2. Description

The ServiceLocationException class is thrown by all methods when exceptional conditions occur in the SLP framework. The error code property determines the exact nature of the condition, and an optional message may provide more information.

5.2.1.3. Fields

```
public static final short LANGUAGE_NOT_SUPPORTED = 1
public static final short PARSE_ERROR = 2
public static final short INVALID_REGISTRATION = 3
public static final short SCOPE_NOT_SUPPORTED = 4
public static final short AUTHENTICATION_ABSENT = 6
public static final short AUTHENTICATION_FAILED = 7
public static final short INVALID_UPDATE = 13
public static final short REFRESH_REJECTED = 15
public static final short NOT_IMPLEMENTED = 16
public static final short NETWORK_INIT_FAILED 17
public static final short NETWORK_TIMED_OUT = 18
public static final short NETWORK_ERROR = 19
public static final short INTERNAL_SYSTEM_ERROR = 20
public static final short TYPE_ERROR = 21
public static final short BUFFER_OVERFLOW = 22
```

5.2.1.4. Instance Methods

```
public short getErrorCode()
```

Return the error code. The error code takes on one of the static field values.

5.3. Basic Data Structures

5.3.1. Interface ServiceLocationEnumeration

```
public interface ServiceLocationEnumeration
 extends Enumeration
```

5.3.1.1. Description

The ServiceLocationEnumeration class is the return type for all Locator SLP operations. The Java API library may implement this class to block until results are available from the SLP operation, so that the client can achieve asynchronous operation by retrieving results from the enumeration in a separate thread. Clients use the superclass nextElement() method if they are unconcerned with SLP exceptions.

5.3.1.2. Instance Methods

```
public abstract Object next() throws ServiceLocationException
```

Return the next value or block until it becomes available.

Throws:

ServiceLocationException

Thrown if the SLP operation encounters an error.

NoSuchElementException

If there are no more elements to return.

5.3.2. Class ServiceLocationAttribute

5.3.2.1. Synopsis

```
public class ServiceLocationAttribute
  extends Object implements Serializable
```

5.3.2.2. Description

The ServiceLocationAttribute class models SLP attributes. Instances of this class are returned by Locator.findAttributes() and are communicated along with register/deregister requests.

5.3.2.3. Constructors

```
public ServiceLocationAttribute(String id,Vector values)
```

Construct a service location attribute. Errors in the id or values vector result in an IllegalArgumentException.

Parameters:

id

The attribute name. The String can consist of any Unicode character.

values

A Vector of one or more attribute values. Vector contents must be uniform in type and one of Integer, String, Boolean, or byte[]. If the attribute is a keyword attribute, then the parameter should be null. String values can consist of any Unicode character.

5.3.2.4. Class Methods

```
public static String escapeId(String id)
```

Returns an escaped version of the id parameter, suitable for inclusion in a query. Any reserved characters as specified in [7] are escaped using UTF-8 encoding. If any characters in the tag are illegal, throws IllegalArgumentException.

Parameters:

id

The attribute id to escape. ServiceLocationException is thrown if any characters are illegal for an attribute tag.

public static String escapeValue(Object value)

Returns a String containing the escaped value parameter as a string, suitable for inclusion in a query. If the parameter is a string, any reserved characters as specified in [7] are escaped using UTF-8 encoding. If the parameter is a byte array, then the escaped string begins with the nonUTF-8 sequence `\ff` and the rest of the string consists of the escaped bytes, which is the encoding for opaques. If the value parameter is a Boolean or Integer, then the returned string contains the object converted into a string. If the value is any type other than String, Integer, Boolean or byte[], an IllegalArgumentException is thrown.

Parameters:

value

The attribute value to be converted into a string and escaped.

5.3.2.5. Instance Methods

public Vector getValues()

Returns a cloned vector of attribute values, or null if the attribute is a keyword attribute. If the attribute is single-valued, then the vector contains only one object.

public String getId()

Returns the attribute's name.

public boolean equals(Object o)

Overrides Object.equals(). Two attributes are equal if their identifiers are equal and their value vectors contain the same number of equal values as determined by the Object equals() method. Values having byte[] type are equal if the contents of all byte arrays in both attribute vectors match. Note that the SLP string matching algorithm [7] MUST NOT be used for comparing attribute identifiers or string values.

public String toString()

Overrides Object.toString(). The string returned contains a formatted representation of the attribute, giving the attribute's id, values, and the Java type of the values. The returned string is suitable for debugging purposes, but is not in SLP wire format.

public int hashCode()

Overrides Object.hashCode(). Hashes on the attribute's identifier.

5.3.3. Class ServiceType

5.3.3.1. Synopsis

public class ServiceType extends Object implements Serializable

5.3.3.2. Description

The ServiceType object models the SLP service type. It parses a string based service type specifier into its various components, and contains property accessors to return the components. URL schemes, protocol service types, and abstract service types are all handled.

5.3.3.3. Constructors

public ServiceType(String type)

Construct a service type object from the service type specifier. Throws IllegalArgumentException if the type name is syntactically incorrect.

Parameters:

type

The service type name as a String. If the service type is from a service: URL, the "service:" prefix must be intact.

5.3.3.4. Methods

public boolean isServiceURL()

Returns true if the type name contains the "service:" prefix.

public boolean isAbstractType()

Returns true if the type name is for an abstract type.

public boolean isNADefault()

Returns true if the naming authority is the default, i.e. is the empty string.

public String getConcreteTypeName()

Returns the concrete type name in an abstract type, or the empty string if the service type is not abstract. For example, if the type name is "service:printing:ipp", the method returns "ipp". If the type name is "service:ftp", the method returns "".

public String getPrincipleTypeName()

Returns the abstract type name for an abstract type, the protocol name in a protocol type, or the URL scheme for a generic URL. For example, in the abstract type name "service:printing:ipp", the method returns "printing". In the protocol type name "service:ftp", the method returns "ftp".

public String getAbstractTypeName()

If the type is an abstract type, returns the fully formatted abstract type name including the "service:" and naming authority but without the concrete type name or intervening colon. If not an abstract type, returns the empty string. For example, in the abstract type name "service:printing:ipp", the method returns "service:printing".

public String getNamingAuthority()

Return the naming authority name, or the empty string if the naming authority is the default.

public boolean equals(Object obj)

Overrides Object.equals(). The two objects are equal if they are both ServiceType objects and the components of both are equal.

public String toString()

Returns the fully formatted type name, including the "service:" if the type was originally from a service: URL.

public int hashCode()

Overrides Object.hashCode(). Hashes on the string value of the "service" prefix, naming authority, if any, abstract and concrete type names for abstract types, protocol type name for protocol types, and URL scheme for generic URLs.

5.3.4. Class ServiceURL

5.3.4.1. Synopsis

public class ServiceURL extends Object implements Serializable

5.3.4.2. Description

The ServiceURL object models the advertised SLP service URL. It can be either a service: URL or a regular URL. These objects are returned from service lookup requests, and describe the registered services. This class should be a subclass of java.net.URL but can't since that class is final.

5.3.4.3. Class Variables

public static final int NO_PORT = 0

Indicates that no port information is required or was returned for this URL.

public static final int LIFETIME_NONE = 0

Indicates that the URL has a zero lifetime. This value is never returned from the API, but can be used to create a ServiceURL object to deregister, delete attributes, or find attributes.

public static final int LIFETIME_DEFAULT = 10800

The default URL lifetime (3 hours) in seconds.

public static final int LIFETIME_MAXIMUM = 65535

The maximum URL lifetime (about 18 hours) in seconds.

public static final int LIFETIME_PERMANENT = -1

Indicates that the API implementation should continuously re-register the URL until the application exits.

5.3.4.4. Constructors

public ServiceURL(String URL,int lifetime)

Construct a service URL object having the specified lifetime.

Parameters:

URL

The URL as a string. Must be either a service: URL or a valid generic URL according to RFC 2396 [2].

lifetime

The service advertisement lifetime in seconds. This value may be between LIFETIME_NONE and LIFETIME_MAXIMUM.

5.3.4.5. Methods

```
public ServiceType getServiceType()
```

Returns the service type object representing the service type name of the URL.

```
public final void setServiceType(ServiceType type)
throws ServiceLocationException
```

Set the service type name to the object. Ignored if the URL is a service: URL.

Parameters:

type

The service type object.

```
public String getTransport()
```

Get the network layer transport identifier. If the transport is IP, an empty string, "", is returned.

```
public String getHost()
```

Returns the host identifier. For IP, this will be the machine name or IP address.

public int getPort()

Returns the port number, if any. For non-IP transports, always returns NO_PORT.

public String getURLPath()

Returns the URL path description, if any.

public int getLifetime()

Returns the service advertisement lifetime. This will be a positive int between LIFETIME_NONE and LIFETIME_MAXIMUM.

public boolean equals(Object obj)

Compares the object to the ServiceURL and returns true if the two are the same. Two ServiceURL objects are equal if their current service types match and they have the same host, port, transport, and URL path.

public String toString()

Returns a formatted string with the URL. Overrides Object.toString(). The returned URL has the original service type or URL scheme, not the current service type.

public int hashCode()

Overrides Object.hashCode(). Hashes on the current service type, transport, host, port, and URL part.

5.4. SLP Access Interfaces

5.4.1. Interface Advertiser

5.4.1.1. Synopsis

```
public interface Advertiser
```

5.4.1.2. Description

The Advertiser is the SA interface, allowing clients to register new service instances with SLP, to change the attributes of existing services, and to deregister service instances. New registrations and modifications of attributes are made in the language locale with which the Advertiser was created, deregistrations of service instances are made for all locales.

5.4.1.3. Instance Methods

```
public abstract Locale getLocale()
```

Return the language locale with which this object was created.

```
public abstract void register(ServiceURL URL,
                              Vector attributes)
throws ServiceLocationException
```

Register a new service with SLP having the given attributes.

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

Parameters:

URL

The URL for the service.

attributes

A vector of ServiceLocationAttribute objects describing the service.

```
public abstract void deregister(ServiceURL URL)
throws ServiceLocationException
```

Deregister a service from the SLP framework. This has the effect of deregistering the service from every language locale. The API library is required to perform the operation in all scopes obtained through configuration.

Parameters:

URL

The URL for the service.

```
public abstract void
addAttributes(ServiceURL URL,
              Vector attributes)
throws ServiceLocationException
```

Update the registration by adding the given attributes. The API library is required to perform the operation in all scopes obtained through configuration.

Parameters:

URL

The URL for the service.

attributes

A Vector of ServiceLocationAttribute objects to add to the existing registration. Use an empty vector to update the URL alone. May not be null.

```
public abstract void
deleteAttributes(ServiceURL URL,
                 Vector attributeIds)
throws ServiceLocationException
```

Delete the attributes from a URL for the locale with which the Advertiser was created. The API library is required to perform the operation in all scopes obtained through configuration.

Parameters:

URL

The URL for the service.

attributeIds

A vector of Strings indicating the ids of the attributes to remove. The strings may be attribute ids or they may be wildcard patterns to match ids. See [7] for the syntax of wildcard patterns. The strings may include SLP reserved characters, they will be escaped by the API before transmission. May not be the empty vector or null.

5.4.2. Interface Locator

5.4.2.1. Synopsis

```
public interface Locator
```

5.4.2.2. Description

The Locator is the UA interface, allowing clients to query the SLP framework about existing service types, services instances, and about the attributes of an existing service instance or service type. Queries for services and attributes are made in the locale with which the Locator was created, queries for service types are independent of locale.

5.4.2.3. Instance Methods

```
public abstract Locale getLocale()
```

Return the language locale with which this object was created.

```
public abstract ServiceLocationEnumeration
findServiceTypes(String namingAuthority,
                 Vector scopes)
throws ServiceLocationException
```

Returns an enumeration of ServiceType objects giving known service types for the given scopes and given naming authority. If no service types are found, an empty enumeration is returned.

Parameters:

namingAuthority

The naming authority. Use "" for the default naming authority and "*" for all naming authorities.

scopes

A Vector of scope names. The vector should be selected from the results of a findScopes() API invocation. Use "DEFAULT" for the default scope.

```
public abstract ServiceLocationEnumeration
findServices(ServiceType type,
             Vector scopes,
             String searchFilter)
throws ServiceLocationException
```

Returns a vector of ServiceURL objects for services matching the query, and having a matching type in the given scopes. If no services are found, an empty enumeration is returned.

Parameters:

type

The SLP service type of the service.

scopes

A Vector of scope names. The vector should be selected from the results of a findScopes() API invocation. Use "DEFAULT" for the default scope.

searchFilter

An LDAPv3 [4] string encoded query. If the filter is empty, i.e. "", all services of the requested type in the specified scopes are returned. SLP reserved characters must be escaped in the query. Use ServiceLocationAttribute.escapeId() and ServiceLocationAttribute.escapeValue() to construct the query.

```
public abstract ServiceLocationEnumeration
findAttributes(ServiceURL URL,
               Vector scopes,
               Vector attributeIds)
throws ServiceLocationException
```

For the URL and scope, return a Vector of ServiceLocationAttribute objects whose ids match the String patterns in the attributeIds Vector. The request is made in the language locale of the Locator. If no attributes match, an empty enumeration is returned.

Parameters:

URL

The URL for which the attributes are desired.

scopes

A Vector of scope names. The vector should be selected from the results of a findScopes() API invocation. Use "DEFAULT" for the default scope.

attributeIds

A Vector of String patterns identifying the desired attributes. An empty vector means return all attributes. As described in [7], the patterns may include wildcards to match substrings. The strings may include SLP reserved characters, they will be escaped by the API before transmission.

```
public abstract ServiceLocationEnumeration
findAttributes(ServiceType type,
               Vector scopes,
               Vector attributeIds)
throws ServiceLocationException
```

For the type and scope, return a Vector of all ServiceLocationAttribute objects whose ids match the String patterns in the attributeIds Vector regardless of the Locator's locale. The request is made independent of language locale. If no attributes are found, an empty vector is returned.

Parameters:

serviceType

The service type.

scopes

A Vector of scope names. The vector should be selected from the results of a findScopes() API invocation. Use "DEFAULT" for the default scope.

attributeIds

A Vector of String patterns identifying the desired attributes. An empty vector means return all attributes. As described in [7], the patterns may include wildcards to match all prefixes or suffixes. The patterns may include SLP reserved characters, they will be escaped by the API before transmission.

5.5. The Service Location Manager

5.5.1. Class ServiceLocationManager

5.5.1.1. Synopsis

```
public class ServiceLocationManager
extends Object
```

5.5.1.2. Description

The ServiceLocationManager manages access to the service location framework. Clients obtain the Locator and Advertiser objects for UA and SA, and a Vector of known scope names from the ServiceLocationManager.

5.5.1.3. Class Methods

```
public static int getRefreshInterval()
throws ServiceLocationException
```

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, assures that no refresh registration will be rejected. If no DA advertises a min-refresh-interval attribute, a value of 0 is returned.

```
public static Vector findScopes()
throws ServiceLocationException
```

Returns an Vector of strings with all available scope names. The list of scopes comes from a variety of sources, see Section 2.1 for the scope discovery algorithm. There is always at least one string in the Vector, the default scope, "DEFAULT".

```
public static Locator
getLocator(Locale locale)
throws ServiceLocationException
```

Return a Locator object for the given language Locale. If the implementation does not support UA functionality, returns null.

Parameters:

locale

The language locale of the Locator. The default SLP locale is used if null.

```
public static Advertiser
getAdvertiser(Locale locale)
throws ServiceLocationException
```

Return an Advertiser object for the given language locale. If the implementation does not support SA functionality, returns null.

Parameters:

locale

The language locale of the Advertiser. The default SLP locale is used if null.

5.6. Service Template Introspection

5.6.1. Abstract Class TemplateRegistry

5.6.1.1. Synopsis

```
public abstract class TemplateRegistry
```

5.6.1.2. Description

Subclasses of the TemplateRegistry abstract class provide access to service location templates [8]. Classes implementing TemplateRegistry perform a variety of functions. They manage the registration and access of service type template documents. They create attribute verifiers from service templates, for verification of attributes and introspection on template documents. Note that clients of the Advertiser are not required to verify attributes before registering (though they may get a TYPE_ERROR if the implementation supports type checking and there is a mismatch with the template).

5.6.1.3. Class Methods

```
public static TemplateRegistry getTemplateRegistry();
```

Returns the distinguished TemplateRegistry object for performing operations on and with service templates. Returns null if the implementation doesn't support TemplateRegistry functionality.

5.6.1.4. Instance Methods

```
public abstract void
registerServiceTemplate(ServiceType type,
                        String documentURL,
                        Locale locale,
                        String version)
throws ServiceLocationException
```

Register the service template with the template registry.

Parameters:

type

The service type.

documentURL

A string containing the URL of the template document. May not be the empty string.

locale

A Locale object containing the language locale of the template.

version

The version number identifier of template document.

```
public abstract void

deregisterServiceTemplate(ServiceType type,
                          Locale locale,
                          String version)
throws ServiceLocationException
```

Deregister the template for the service type.

Parameters:

type

The service type.

locale

A Locale object containing the language locale of the template.

version

A String containing the version number. Use null to indicate the latest version.

```
public abstract
String findTemplateURL(ServiceType type,
                       Locale locale,
                       String version)
throws ServiceLocationException
```

Returns the URL for the template document.

Parameters:

type

The service type.

locale

A Locale object containing the language locale of the template.

version

A String containing the version number. Use null to indicate the latest version.

```
public abstract
ServiceLocationAttributeVerifier
attributeVerifier(String documentURL)
throws ServiceLocationException
```

Reads the template document URL and returns an attribute verifier for the service type. The attribute verifier can be used for verifying that registration attributes match the template, and for introspection on the template definition.

Parameters:

documentURL

A String containing the template document's URL. May not be the empty string.

5.6.2. Interface ServiceLocationAttributeVerifier

5.6.2.1. Synopsis

```
public interface ServiceLocationAttributeVerifier
```

5.6.2.2. Description

The ServiceLocationAttributeVerifier provides access to service templates. Classes implementing this interface parse SLP template definitions, provide information on attribute definitions for service types, and verify whether a ServiceLocationAttribute object matches a template for a particular service type. Clients obtain ServiceLocationAttributeVerifier objects for specific SLP service types through the TemplateRegistry.

5.6.2.3. Instance Methods

```
public abstract ServiceType getServiceType()
```

Returns the SLP service type for which this is the verifier.

```
public abstract Locale getLocale()
```

Return the language locale of the template.

```
public abstract String getVersion()
```

Return the template version number identifier.

```
public abstract String getURLSyntax()
```

Return the URL syntax expression for the service: URL.

```
public abstract String getDescription()
```

Return the descriptive help text for the template.

```
public abstract ServiceLocationAttributeDescriptor
getAttributeDescriptor(String attrId)
```

Return the ServiceLocationAttributeDescriptor for the attribute having the named id. If no such attribute exists in this template, return null. This method is primarily for GUI tools to display attribute information. Programmatic verification of attributes should use the verifyAttribute() method.

```
public abstract Enumeration
getAttributeDescriptors()
```

Returns an Enumeration allowing introspection on the attribute definition in the service template. The Enumeration returns ServiceLocationAttributeDescriptor objects for the attributes. This method is primarily for GUI tools to display attribute information. Programmatic verification of attributes should use the verifyAttribute() method.

```
public abstract void
verifyAttribute(
  ServiceLocationAttribute attribute)
throws ServiceLocationException
```

Verify that the attribute matches the template definition. If the attribute doesn't match, ServiceLocationException is thrown with the error code as ServiceLocationException.PARSE_ERROR.

Parameters:

attribute

The ServiceLocationAttribute object to be verified.

```
public abstract void
verifyRegistration(
  Vector attributeVector)
throws ServiceLocationException
```

Verify that the Vector of ServiceLocationAttribute objects matches the template for this service type. The vector must contain all the required attributes, and all attributes must match their template definitions. If the attributes don't match, ServiceLocationException is thrown with the error code as ServiceLocationException.PARSE_ERROR

Parameters:

attributeVector

A Vector of ServiceLocationAttribute objects for the registration.

5.6.3. Interface ServiceLocationAttributeDescriptor

5.6.3.1. Synopsis

public interface
ServiceLocationAttributeDescriptor

5.6.3.2. Description

The ServiceLocationAttributeDescriptor interface provides introspection on a template attribute definition. Classes implementing the ServiceLocationAttributeDescriptor interface return information on a particular service location attribute definition from the service template. This information is primarily for GUI tools. Programmatic attribute verification should be done through the ServiceLocationAttributeVerifier.

5.6.3.3. Instance Methods

public abstract String getId()

Return a String containing the attribute's id.

public abstract String getValueType()

Return a String containing the fully package-qualified Java type of the attribute. SLP types are translated into Java types as follows:

STRING

"java.lang.String"

INTEGER

"java.lang.Integer"

BOOLEAN

"java.lang.Boolean"

OPAQUE

"[B" (i.e. array of byte, byte[])

KEYWORD

empty string, ""

public abstract String getDescription()

Return a String containing the attribute's help text.

public abstract Enumeration
getAllowedValues()

Return an Enumeration of allowed values for the attribute type. For keyword attributes returns null. For no allowed values (i.e. unrestricted) returns an empty Enumeration.

public abstract Enumeration
getDefaultValues()

Return an Enumeration of default values for the attribute type. For keyword attributes returns null. For no allowed values (i.e. unrestricted) returns an empty Enumeration.

public abstract boolean
getRequiresExplicitMatch()

Returns true if the "X"" flag is set, indicating that the attribute should be included in an any Locator.findServices() request search filter.

public abstract boolean getIsMultivalued()

Returns true if the "M" flag is set.

public abstract boolean getIsOptional()

Returns true if the "O"" flag is set.

public abstract boolean getIsLiteral()

Returns true if the "L" flag is set.

public abstract boolean getIsKeyword()

Returns true if the attribute is a keyword attribute.

5.7. Implementation Notes

5.7.1. Refreshing Registrations

A special lifetime constant, ServiceURL.LIFETIME_PERMANENT, is used by clients to indicate that the URL should be automatically refreshed until the application exits. The API implementation should interpret this flag as indicating that the URL lifetime is ServiceURL.LIFETIME_MAXIMUM, and MUST arrange for automatic refresh to occur.

5.7.2. Parsing Alternate Transports in ServiceURL

The ServiceURL class is designed to handle multiple transports. The standard API performs no additional processing on transports other than IP except to separate out the host identifier and the URL path. However, implementations are free to subclass ServiceURL and support additional methods that provide more detailed parsing of alternate transport information. For IP transport, the port number, if any, is

returned from the getPort() method. For non-IP transports, the getPort() method returns NO_PORT.

5.7.3. String Attribute Values

In general, translation between Java types for attribute values and the SLP on-the-wire string is straightforward. However, there are two corner cases. If the Java attribute value type is String and the value of the string has an on-the-wire representation that is inferred by SLP as an integer, the registered attribute value may not be what the API client intended. A similar problem could result if the Java attribute value is the string "true" or "false", in which case the on-the-wire representation is inferred to boolean. To handle these corner cases, the Java API prepends a space onto the string. So, for example, if the string attribute value is "123", the Java API transforms the value to "123 ", which will have an on-the-wire representation that is inferred by SLP to be string. Since appended and prepended spaces have no effect on query handling, this procedure should cause no problem with queries. API clients need to be aware, however, that the transformation is occurring.

5.7.4. Client Side Syntax Checking

The syntax of scope names, service type names, naming authority names, and URLs is described in [7] and [8]. The various methods and classes taking String parameters for these entities SHOULD type check the parameters for syntax errors on the client side, and throw an IllegalArgumentException if an error occurs. In addition, character escaping SHOULD be implemented before network transmission for escapable characters in attribute ids and String values. This reduces the number of error messages transmitted. The ServiceLocationAttribute class provides methods for clients to obtain escaped attribute id and value strings to facilitate query construction.

5.7.5. Language Locale Handling

The Locator and Advertiser interfaces are created with a Locale parameter. The language locale with which these objects are created is used in all SLP requests issued through the object. If the Locale parameter is null, the default SLP locale is used. The default SLP locale is determined by, first, checking the net.slp.locale System property. If that is unset, then the default SLP locale [7] is used, namely "en". Note that the default SLP locale may not be the same as the default Java locale.

5.7.6. Setting SLP System Properties

SLP system properties that are originally set in the configuration file can be overridden programmatically in API clients by simply invoking the System.getProperties() operation to get a copy of the system properties, modifying or adding the SLP property in question, then using System.setProperties() to set the properties to the modified Property object. Program execution continues without interruption by substituting the default for the erroneous parameter. Errors are checked when the property is used and are logged.

The SLP configuration file cannot be read with the java.util.Properties file reader because there are some syntactic differences. The SLP configuration file syntax defines a different escape convention for non-ASCII characters than the Java syntax. However, after the file has been read, the properties are stored and retrieved from java.util.Properties objects.

Properties are global for a process, affecting all threads and all Locator and Advertiser objects obtained through the ServiceLocationManager. With the exception of the net.slp.locale, net.slp.typeHint, and net.slp.maxResults properties, clients 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.

5.7.7. Multithreading

Thread-safe operation is relatively easy to achieve in Java. By simply making each method in the classes implementing the Locator and Advertiser interfaces synchronized, and by synchronizing access to any shared data structures within the class, the Locator and Advertiser interfaces are made safe. Alternatively, finer grained synchronization is also possible within the classes implementing Advertiser and Locator.

5.7.8. Modular Implementations

While, at first glance, the API may look rather heavyweight, the design has been carefully arranged so that modular implementations that provide only SA, only UA, or only service template access capability, or any combination of the three, are possible.

Because the objects returned from the ServiceLocationManager.getLocator() and ServiceLocationManager.getAdvertiser() operations are interfaces, and because the objects returned through those interfaces are in the set

of base data structures, an implementation is free to omit either UA or SA capability by simply returning null from the instance creation operation if the classes implementing the missing function cannot be dynamically linked. API clients are encouraged to check for such a contingency, and to signal an exception if it occurs. Similarly, the TemplateRegistry concrete subclass can simply be omitted from an implementation that only supports UA and/or SA clients, and the TemplateRegistry.getRegistry() method can return null. In this way, the API implementation can be tailored for the particular memory requirements at hand.

In addition, if an implementation only supports the minimal subset of SLP [7], the unsupported Locator and Advertiser interface operations can throw an exception with ServiceLocationException.NOT_IMPLEMENTED as the error code. This supports better source portability between low and high memory platforms.

5.7.9. Asynchronous and Incremental Return Semantics

The Java API contains no specific support for asynchronous operation. Incremental return is not needed for the Advertiser because service registrations can be broken up into pieces when large. Asynchronous return is also not needed because clients can always issue the Advertiser operation in a separate thread if the calling thread can't block.

The Locator can be implemented either synchronously or asynchronously. Since the return type for Locator calls is ServiceLocationEnumeration, a Java API implementation that supports asynchronous semantics can implement ServiceLocationEnumeration to dole results out as they come in, blocking when no results are available. If the client code needs to support other processing while the results are trickling in, the call into the enumeration to retrieve the results can be done in a separate thread.

Unlike the C case, collation semantics for return of attributes when an attribute request by service type is made require that the API collate returned values so that only one attribute having a collation of all returned values appear to the API client. In practice, this may limit the amount of asynchronous processing possible with the findAttributes() method. This requirement is imposed because memory management is much easier in Java and so implementing collation as part of the API should not be as difficult as in C, and it saves the client from having to do the collation.

5.8. Example

In this example, a printer server advertises its availability to clients. Additionally, the server advertises a service template for use by client software in validating service requests:

```
//Get the Advertiser and TemplateRegistry.

Advertiser adv = null;
TemplateRegistry tr = null

try {

  adv = ServiceLocationManager.getAdvertiser("en");

  tr = TemplateRegistry.getTemplateRegistry();

} catch( ServiceLocationException ex ) { } //Deal with error.

if( adv == null ) {

  //Serious error as printer can't be registered
  //  if the implementation doesn't support SA
  //  functionality.

}

//Get the printer's attributes, from a file or
//  otherwise. We assume that the attributes
//  conform to the template, otherwise, we
//  could register the template here and verify
//  them.

Vector attributes = getPrinterAttributes();

//Create the service: URL for the printer.

ServiceURL printerURL =
  new ServiceURL(
    "service:printer:lpr://printshop/color2",
    ServiceURL.LIFETIME_MAXIMUM);

try {

  //Register the printer.

  adv.register(printerURL, attributes);
```

```
    //If the template registry is available,
    //  register the printer's template.

    if( tr != null ) {
      tr.registerServiceTemplate(
        new ServiceType("service:printer:lpr"),
        "http://shop.arv/printer/printer-lpr.slp",
        new Locale("en",""),
        "1.0");

   }

  } catch( ServiceLocationException ex ) { } //Deal with error.
```

Suppose a client is looking for color printer. The following code is used to issue a request for printer advertisements:

```
  Locator loc = null;
  TemplateRegistry tr = null;

  try {

    loc = ServiceLocationManager.getLocator("en");

  } catch( ServiceLocationException ex ) { } //Deal with error.

  if( loc == null ) {

    //Serious error as client can't be located
    //  if the implementation doesn't support
    //  UA functionality.

  }

  //We want a color printer that does CMYK
  //  and prints at least 600 dpi.

  String query = "(&(marker-type=CMYK)(resolution=600))";

  //Get scopes.

  Vector scopes = ServiceLocationManager.findScopes();

  Enumeration services;

  try {
```

```
    services =
      loc.findServices(new ServiceType("service:printer"),scopes,query);

  } catch { } //Deal with error.

  if (services.hasMoreElements() ) {

    //Printers can now be used.
    ServiceURL surl = (ServiceURL) services.next();

    Socket sock = new Socket(surl.getHost, surl.getPort());

    // Use the Socket...

  }
```

6. Internationalization Considerations

6.1. service URL

The service URL itself must be encoded using the rules set forth in [2]. The character set encoding is limited to specific ranges within the UTF-8 character set [3].

The attribute information associated with the service URL must be expressed in UTF-8. See [8] for attribute internationalization guidelines.

6.2. Character Set Encoding

Configuration and serialized registration files are encoded in the UTF-8 character set [3]. This is fully compatible with US-ASCII character values. C platforms that do not support UTF-8 are required to check the top bit of input bytes to determine whether the incoming character is multibyte. If it is, the character should be dealt with accordingly. This should require no additional implementation effort, since the SLP wire protocol requires that strings are encoded as UTF-8. C platforms without UTF-8 support need to supply their own support, if only in the form of multibyte string handling.

At the API level, the character encoding is specified to be Unicode for Java and UTF-8 for C. Unicode is the default in Java. For C, the standard US-ASCII 8 bits per character, null terminated C strings are a subset of the UTF-8 character set, and so work in the API. Because the C API is very simple, the API library needs to do a minimum of processing on UTF-8 strings. The strings primarily just need to be reflected into the outgoing SLP messages, and reflected out of the

API from incoming SLP messages.

6.3. Language Tagging

All SLP requests and registrations are tagged to indicate in which language the strings included are encoded. This allows multiple languages to be supported. It also presents the possibility that error conditions result when a request is made in a language that is not supported. In this case, an error is only returned when there is data available, but not obtainable in the language requested.

The dialect portion of the Language Tag is used on 'best effort' basis for matching strings by SLP. Dialects that match are preferred over those which don't. Dialects that do not match will not prevent string matching or comparisons from occurring.

7. Security Considerations

Security is handled within the API library and is not exposed to API clients except in the form of exceptions. The net.slp.securityEnabled, property determines whether an SA client's messages are signed, but a UA client should be prepared for an authentication exception at any time, because it may contact a DA with authenticated advertisements.

An adversary could delete valid service advertisements, provide false service information and deny UAs knowledge of existing services unless the mechanisms in SLP for authenticating SLP messages are used. These mechanisms allow DAAdverts, SAAdverts, Service URLs and Service Attributes to be verified using digital cryptography. For this reason, all SLP agents should be configured to use SLP SPIs. See [7] for a description of how this mechanism works.

8. Acknowledgements

The authors would like to thank Don Provan for his pioneering work during the initial stages of API definition.

9. References

[1] Bradner, S., "Key Words for Use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

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

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

[4] Howes, T., "The String Representation of LDAP Search Filters", RFC 2254 December 1997.

[5] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[6] Alvestrand, H., "Tags for the Identification of Languages", RFC 1766, March 1995.

[7] Guttman, E., Perkins, C., Veizades, J. and M. Day, "Service Location Protocol, Version 2", RFC 2608, June 1999.

[8] Guttman, E., Perkins, C. and J. Kempf, "Service Templates and Service: Schemes", RFC 2609, June 1999.

10. Authors' Addresses

Questions about this memo can be directed to:

James Kempf
Sun Microsystems
901 San Antonio Rd.
Palo Alto, CA, 94303
USA

Phone: +1 650 786 5890
Fax: +1 650 786 6445
EMail: james.kempf@sun.com

Erik Guttman
Sun Microsystems
Bahnstr. 2
74915 Waibstadt
Germany

Phone: +49 7263 911 701
EMail: erik.guttman@sun.com

11. Full Copyright Statement

Acknowledgement

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