NAME

org.freedesktop.resolve1 − The D−Bus interface of systemd−resolved

INTRODUCTION

systemd-resolved.service(8) is a system service that provides hostname resolution and caching using DNS, LLMNR, and mDNS. It also does DNSSEC validation. This page describes the resolve semantics and the D−Bus interface.

This page contains an API reference only. If you are looking for a longer explanation how to use this API, please consult **Writing Network Configuration Managers**[1] and **Writing Resolver Clients**[2].

THE MANAGER OBJECT

The service exposes the following interfaces on the Manager object on the bus:

```
node /org/freedesktop/resolve1 {
  interface org.freedesktop.resolve1.Manager {
    methods:
      ResolveHostname(in  i ifindex,
                      in  s name,
                      in  i family,
                      in  t flags,
                      out a(iiay) addresses,
                      out s canonical,
                      out t flags);
      ResolveAddress(in  i ifindex,
                     in  i family,
                     in  ay address,
                     in  t flags,
                     out a(is) names,
                     out t flags);
      ResolveRecord(in  i ifindex,
                    in  s name,
                    in  q class,
                    in  q type,
                    in  t flags,
                    out a(iqqay) records,
                    out t flags);
      ResolveService(in  i ifindex,
                     in  s name,
                     in  s type,
                     in  s domain,
                     in  i family,
                     in  t flags,
                     out a(qqqsa(iiay)s) srv_data,
                     out aay txt_data,
                     out s canonical_name,
                     out s canonical_type,
                     out s canonical_domain,
                     out t flags);
      GetLink(in  i ifindex,
              out o path);
      SetLinkDNS(in  i ifindex,
                 in  a(iay) addresses);
      SetLinkDNSEx(in  i ifindex,
                   in  a(iayqs) addresses);
      SetLinkDomains(in  i ifindex,
```

```
                in  a(sb) domains);
  SetLinkDefaultRoute(in  i ifindex,
                      in  b enable);
  SetLinkLLMNR(in  i ifindex,
               in  s mode);
  SetLinkMulticastDNS(in  i ifindex,
                      in  s mode);
  SetLinkDNSOverTLS(in  i ifindex,
                    in  s mode);
  SetLinkDNSSEC(in  i ifindex,
                in  s mode);
  SetLinkDNSSECNegativeTrustAnchors(in  i ifindex,
                                    in  as names);
  RevertLink(in  i ifindex);
  RegisterService(in  s name,
                  in  s name_template,
                  in  s type,
                  in  q service_port,
                  in  q service_priority,
                  in  q service_weight,
                  in  aa{say} txt_datas,
                  out o service_path);
  UnregisterService(in  o service_path);
  ResetStatistics();
  FlushCaches();
  ResetServerFeatures();
properties:
  readonly s LLMNRHostname = '...';
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly s LLMNR = '...';
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly s MulticastDNS = '...';
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly s DNSOverTLS = '...';
  readonly a(iiay) DNS = [...];
  readonly a(iiayqs) DNSEx = [...];
  @org.freedesktop.DBus.Property.EmitsChangedSignal("const")
  readonly a(iiay) FallbackDNS = [...];
  @org.freedesktop.DBus.Property.EmitsChangedSignal("const")
  readonly a(iiayqs) FallbackDNSEx = [...];
  readonly (iiay) CurrentDNSServer = ...;
  readonly (iiayqs) CurrentDNSServerEx = ...;
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly a(isb) Domains = [...];
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly (tt) TransactionStatistics = ...;
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly (ttt) CacheStatistics = ...;
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly s DNSSEC = '...';
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly (tttt) DNSSECStatistics = ...;
  @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
  readonly b DNSSECSupported = ...;
```

```
        @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
        readonly as DNSSECNegativeTrustAnchors = ['...', ...];
        @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
        readonly s DNSStubListener = '...';
        @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
        readonly s ResolvConfMode = '...';
      };
      interface org.freedesktop.DBus.Peer { ... };
      interface org.freedesktop.DBus.Introspectable { ... };
      interface org.freedesktop.DBus.Properties { ... };
    };
```

Methods

ResolveHostname() takes a hostname and resolves it to one or more IP addresses. As parameters it takes the Linux network interface index to execute the query on, or 0 if it may be done on any suitable interface. The *name* parameter specifies the hostname to resolve. Note that if required, IDNA conversion is applied to this name unless it is resolved via LLMNR or MulticastDNS. The *family* parameter limits the results to a specific address family. It may be **AF_INET**, **AF_INET6** or **AF_UNSPEC**. If **AF_UNSPEC** is specified (recommended), both kinds are retrieved, subject to local network configuration (i.e. if no local, routable IPv6 address is found, no IPv6 address is retrieved; and similarly for IPv4). A 64−bit *flags* field may be used to alter the behaviour of the resolver operation (see below). The method returns an array of address records. Each address record consists of the interface index the address belongs to, an address family as well as a byte array with the actual IP address data (which either has 4 or 16 elements, depending on the address family). The returned address family will be one of **AF_INET** or **AF_INET6**. For IPv6, the returned address interface index should be used to initialize the .sin6_scope_id field of a struct sockaddr_in6 instance to permit support for resolution to link−local IP addresses. The address array is followed by the canonical name of the host, which may or may not be identical to the resolved hostname. Finally, a 64−bit *flags* field is returned that is defined similarly to the *flags* field that was passed in, but contains information about the resolved data (see below). If the hostname passed in is an IPv4 or IPv6 address formatted as string, it is parsed, and the result is returned. In this case, no network communication is done.

ResolveAddress() executes the reverse operation: it takes an IP address and acquires one or more hostnames for it. As parameters it takes the interface index to execute the query on, or **0** if all suitable interfaces are OK. The *family* parameter indicates the address family of the IP address to resolve. It may be either **AF_INET** or **AF_INET6**. The *address* parameter takes the raw IP address data (as either a 4 or 16 byte array). The *flags* input parameter may be used to alter the resolver operation (see below). The method returns an array of name records, each consisting of an interface index and a hostname. The *flags* output field contains additional information about the resolver operation (see below).

ResolveRecord() takes a DNS resource record (RR) type, class and name, and retrieves the full resource record set (RRset), including the RDATA, for it. As parameter it takes the Linux network interface index to execute the query on, or **0** if it may be done on any suitable interface. The *name* parameter specifies the RR domain name to look up (no IDNA conversion is applied), followed by the 16−bit class and type fields (which may be ANY). Finally, a *flags* field may be passed in to alter behaviour of the look−up (see below). On completion, an array of RR items is returned. Each array entry consists of the network interface index the RR was discovered on, the type and class field of the RR found, and a byte array of the raw RR discovered. The raw RR data starts with the RR's domain name, in the original casing, followed by the RR type, class, TTL and RDATA, in the binary format documented in **RFC 1035**[3]. For RRs that support name compression in the payload (such as MX or PTR), the compression is expanded in the returned data.

Note that currently, the class field has to be specified as IN or ANY. Specifying a different class will return an error indicating that look−ups of this kind are unsupported. Similarly, some special types are not supported either (AXFR, OPT, ...). While systemd−resolved parses and validates resource records of many types, it is crucial that clients using this API understand that the RR data originates from the network and should be thoroughly validated before use.

ResolveService() may be used to resolve a DNS **SRV** service record, as well as the hostnames referenced in it, and possibly an accompanying DNS−SD **TXT** record containing additional service metadata. The primary benefit of using this method over **ResolveRecord()** specifying the **SRV** type is that it will resolve the **SRV** and **TXT** RRs as well as the hostnames referenced in the SRV in a single operation. As parameters it takes a Linux network interface index, a service name, a service type and a service domain. This method may be invoked in three different modes:

1. To resolve a DNS−SD service, specify the service name (e.g. "Lennart's Files"), the service type (e.g. "_webdav._tcp") and the domain to search in (e.g. "local") as the three service parameters. The service name must be in UTF−8 format, and no IDNA conversion is applied to it in this mode (as mandated by the DNS−SD specifications). However, if necessary, IDNA conversion is applied to the domain parameter.

2. To resolve a plain **SRV** record, set the service name parameter to the empty string and set the service type and domain properly. (IDNA conversion is applied to the domain, if necessary.)
3. Alternatively, leave both the service name and type empty and specify the full domain name of the **SRV** record (i.e. prefixed with the service type) in the domain parameter. (No IDNA conversion is applied in this mode.)

The *family* parameter of the **ResolveService()** method encodes the desired family of the addresses to resolve (use **AF_INET**, **AF_INET6**, or **AF_UNSPEC**). If this is enabled (Use the **NO_ADDRESS** flag to turn address resolution off, see below). The *flags* parameter takes a couple of flags that may be used to alter the resolver operation.

On completion, **ResolveService()** returns an array of **SRV** record structures. Each items consisting of the priority, weight and port fields as well as the hostname to contact, as encoded in the **SRV** record. Immediately following is an array of the addresses of this hostname, with each item consisting of the interface index, the address family and the address data in a byte array. This address array is followed by the canonicalized hostname. After this array of **SRV** record structures an array of byte arrays follows that encodes the TXT RR strings, in case DNS−SD look−ups are enabled. The next parameters are the canonical service name, type and domain. This may or may not be identical to the parameters passed in. Finally, a *flags* field is returned that contains information about the resolver operation performed.

The **ResetStatistics()** method resets the various statistics counters that systemd−resolved maintains to zero. (For details, see the statistics properties below.)

The **GetLink()** method takes a network interface index and returns the object path to the org.freedesktop.resolve1.Link object corresponding to it.

The **SetLinkDNS()** method sets the DNS servers to use on a specific interface. This method (and the following ones) may be used by network management software to configure per−interface DNS settings. It takes a network interface index as well as an array of DNS server IP address records. Each array item consists of an address family (either **AF_INET** or **AF_INET6**), followed by a 4−byte or 16−byte array with the raw address data. This method is a one−step shortcut for retrieving the Link object for a network interface using **GetLink()** (see above) and then invoking the **SetDNS()** method (see below) on it.

SetLinkDNSEx() is similar to **SetLinkDNS()**, but allows an IP port (instead of the default 53) and DNS name to be specified for each DNS server. The server name is used for Server Name Indication (SNI), which is useful when DNS−over−TLS is used. C.f. *DNS=* in **resolved.conf**(5).

SetLinkDefaultRoute() specifies whether the link shall be used as the default route for name queries. See the description of name routing in **systemd-resolved.service**(8) for details.

The **SetLinkDomains()** method sets the search and routing domains to use on a specific network interface for DNS look−ups. It takes a network interface index and an array of domains, each with a boolean parameter indicating whether the specified domain shall be used as a search domain (false), or just as a routing domain (true). Search domains are used for qualifying single−label names into FQDN when looking up hostnames, as well as for making routing decisions on which interface to send queries ending in the domain to. Routing domains are only used for routing decisions and not used for single−label name qualification. Pass the search domains in the order they should be used.

The **SetLinkLLMNR()** method enables or disables LLMNR support on a specific network interface. It takes a network interface index as well as a string that may either be empty or one of "yes", "no" or "resolve". If empty, the systemd−wide default LLMNR setting is used. If "yes", LLMNR is used for resolution of single−label names and the local hostname is registered on all local LANs for LLMNR resolution by peers. If "no", LLMNR is turned off fully on this interface. If "resolve", LLMNR is only enabled for resolving names, but the local hostname is not registered for other peers to use.

Similarly, the **SetLinkMulticastDNS()** method enables or disables MulticastDNS support on a specific interface. It takes the same parameters as **SetLinkLLMNR()** described above.

The **SetLinkDNSSEC()** method enables or disables DNSSEC validation on a specific network interface. It takes a network interface index as well as a string that may either be empty or one of "yes", "no", or "allow−downgrade". When empty, the system−wide default DNSSEC setting is used. If "yes", full

DNSSEC validation is done for all look−ups. If the selected DNS server does not support DNSSEC, look−ups will fail if this mode is used. If "no", DNSSEC validation is fully disabled. If "allow−downgrade", DNSSEC validation is enabled, but is turned off automatically if the selected server does not support it (thus opening up behaviour to downgrade attacks). Note that DNSSEC only applies to traditional DNS, not to LLMNR or MulticastDNS.

The **SetLinkDNSSECNegativeTrustAnchors()** method may be used to configure DNSSEC Negative Trust Anchors (NTAs) for a specific network interface. It takes a network interface index and a list of domains as arguments.

The **SetLinkDNSOverTLS()** method enables or disables DNS−over−TLS. C.f. *DNSOverTLS*= in **systemd-resolved.service**(8) for details.

Network management software integrating with systemd−resolved should call **SetLinkDNS()** or **SetLinkDNSEx()**, **SetLinkDefaultRoute()**, **SetLinkDomains()** and others after the interface appeared in the kernel (and thus after a network interface index has been assigned), but before the network interfaces is activated (**IFF_UP** set) so that all settings take effect during the full time the network interface is up. It is safe to alter settings while the interface is up, however. Use **RevertLink()** (described below) to reset all per−interface settings.

The **RevertLink()** method may be used to revert all per−link settings described above to the defaults.

The Flags Parameter

The four methods above accept and return a 64−bit flags value. In most cases passing 0 is sufficient and recommended. However, the following flags are defined to alter the look−up:

```
#define SD_RESOLVED_DNS             (UINT64_C(1) <<  0)
#define SD_RESOLVED_LLMNR_IPV4      (UINT64_C(1) <<  1)
#define SD_RESOLVED_LLMNR_IPV6      (UINT64_C(1) <<  2)
#define SD_RESOLVED_MDNS_IPV4       (UINT64_C(1) <<  3)
#define SD_RESOLVED_MDNS_IPV6       (UINT64_C(1) <<  4)
#define SD_RESOLVED_NO_CNAME        (UINT64_C(1) <<  5)
#define SD_RESOLVED_NO_TXT          (UINT64_C(1) <<  6)
#define SD_RESOLVED_NO_ADDRESS      (UINT64_C(1) <<  7)
#define SD_RESOLVED_NO_SEARCH       (UINT64_C(1) <<  8)
#define SD_RESOLVED_AUTHENTICATED   (UINT64_C(1) <<  9)
#define SD_RESOLVED_NO_VALIDATE     (UINT64_C(1) << 10)
#define SD_RESOLVED_NO_SYNTHESIZE   (UINT64_C(1) << 11)
#define SD_RESOLVED_NO_CACHE        (UINT64_C(1) << 12)
#define SD_RESOLVED_NO_ZONE         (UINT64_C(1) << 13)
#define SD_RESOLVED_NO_TRUST_ANCHOR (UINT64_C(1) << 14)
#define SD_RESOLVED_NO_NETWORK      (UINT64_C(1) << 15)
#define SD_RESOLVED_REQUIRE_PRIMARY (UINT64_C(1) << 16)
#define SD_RESOLVED_CLAMP_TTL       (UINT64_C(1) << 17)
#define SD_RESOLVED_CONFIDENTIAL    (UINT64_C(1) << 18)
#define SD_RESOLVED_SYNTHETIC       (UINT64_C(1) << 19)
#define SD_RESOLVED_FROM_CACHE      (UINT64_C(1) << 20)
#define SD_RESOLVED_FROM_ZONE       (UINT64_C(1) << 21)
#define SD_RESOLVED_FROM_TRUST_ANCHOR (UINT64_C(1) << 22)
#define SD_RESOLVED_FROM_NETWORK    (UINT64_C(1) << 23)
```

On input, the first five flags control the protocols to use for the look−up. They refer to classic unicast DNS, LLMNR via IPv4/UDP and IPv6/UDP respectively, as well as MulticastDNS via IPv4/UDP and IPv6/UDP. If all of these five bits are off on input (which is strongly recommended) the look−up will be done via all suitable protocols for the specific look−up. Note that these flags operate as filter only, but cannot force a look−up to be done via a protocol. Specifically, systemd−resolved will only route

look−ups within the .local TLD to MulticastDNS (plus some reverse look−up address domains), and single−label names to LLMNR (plus some reverse address lookup domains). It will route neither of these to Unicast DNS servers. Also, it will do LLMNR and Multicast DNS only on interfaces suitable for multicast.

On output, these five flags indicate which protocol was used to execute the operation, and hence where the data was found.

The primary use cases for these five flags are follow−up look−ups based on DNS data retrieved earlier. In this case it is often a good idea to limit the follow−up look−up to the protocol that was used to discover the first DNS result.

The NO_CNAME flag controls whether CNAME/DNAME resource records shall be followed during the look−up. This flag is only available at input, none of the functions will return it on output. If a CNAME/DNAME RR is discovered while resolving a hostname, an error is returned instead. By default, when the flag is off, CNAME/DNAME RRs are followed.

The NO_TXT and NO_ADDRESS flags only influence operation of the **ResolveService()** method. They are only defined for input, not output. If NO_TXT is set, the DNS−SD TXT RR look−up is not done in the same operation. If NO_ADDRESS is set, the discovered hostnames are not implicitly translated to their addresses.

The NO_SEARCH flag turns off the search domain logic. It is only defined for input in **ResolveHostname()**. When specified, single−label hostnames are not qualified using defined search domains, if any are configured. Note that **ResolveRecord()** will never qualify single−label domain names using search domains. Also note that multi−label hostnames are never subject to search list expansion.

The AUTHENTICATED bit is defined only in the output flags of the four functions. If set, the returned data has been fully authenticated. Specifically, this bit is set for all DNSSEC−protected data for which a full trust chain may be established to a trusted domain anchor. It is also set for locally synthesized data, such as "localhost" or data from /etc/hosts. Moreover, it is set for all LLMNR or mDNS RRs which originate from the local host. Applications that require authenticated RR data for operation should check this flag before trusting the data. Note that systemd−resolved will never return invalidated data, hence this flag simply allows to discern the cases where data is known to be trusted, or where there is proof that the data is "rightfully" unauthenticated (which includes cases where the underlying protocol or server does not support authenticating data).

NO_VALIDATE can be set to disable validation via DNSSEC even if it would normally be used.

The next four flags allow disabling certain sources during resolution. NO_SYNTHESIZE disables synthetic records, e.g. the local host name, see section SYNTHETIC RECORDS in **systemd-resolved.service**(8) for more information. NO_CACHE disables the use of the cache of previously resolved records. NO_ZONE disables answers using locally registered public LLMNR/mDNS resource records. NO_TRUST_ANCHOR disables answers using locally configured trust anchors. NO_NETWORK requires all answers to be provided without using the network, i.e. either from local sources or the cache.

With REQUIRE_PRIMARY the request must be answered from a "primary" answer, i.e. not from resource records acquired as a side−effect of a previous transaction.

With CLAMP_TTL, if reply is answered from cache, the TTLs will be adjusted by age of cache entry.

The next six bits flags are used in output and provide information about the source of the answer. CONFIDENTIAL means the query was resolved via encrypted channels or never left this system. FROM_SYNTHETIC means the query was (at least partially) synthesized. FROM_CACHE means the query was answered (at least partially) using the cache. FROM_ZONE means the query was answered (at least partially) using LLMNR/mDNS. FROM_TRUST_ANCHOR means the query was answered (at least partially) using local trust anchors. FROM_NETWORK means the query was answered (at least partially) using the network.

Properties

The *LLMNR* and *MulticastDNS* properties report whether LLMNR and MulticastDNS are (globally) enabled. Each may be one of "yes", "no", and "resolve". See **SetLinkLLMNR()** and **SetLinkMulticastDNS()** above.

LLMNRHostname contains the hostname currently exposed on the network via LLMNR. It usually follows the system hostname as may be queried via **gethostname**(3), but may differ if a conflict is detected on the network.

DNS and *DNSEx* contain arrays of all DNS servers currently used by systemd−resolved. *DNS* contains information similar to the DNS server data in /run/systemd/resolve/resolv.conf. Each structure in the array consists of a numeric network interface index, an address family, and a byte array containing the DNS server address (either 4 bytes in length for IPv4 or 16 bytes in lengths for IPv6). *DNSEx* is similar, but additionally contains the IP port and server name (used for Server Name Indication, SNI). Both arrays contain DNS servers configured system−wide, including those possibly read from a foreign /etc/resolv.conf or the *DNS*= setting in /etc/systemd/resolved.conf, as well as per−interface DNS server information either retrieved from **systemd-networkd**(8), or configured by external software via **SetLinkDNS()** or **SetLinkDNSEx()** (see above). The network interface index will be 0 for the system−wide configured services and non−zero for the per−link servers.

FallbackDNS and *FallbackDNSEx* contain arrays of all DNS servers configured as fallback servers, if any, using the same format as *DNS* and *DNSEx* described above. See the description of *FallbackDNS*= in **resolved.conf**(5) for the description of when those servers are used.

CurrentDNSServer and *CurrentDNSServerEx* specify the server that is currently used for query resolution, in the same format as a single entry in the *DNS* and *DNSEx* arrays described above.

Similarly, the *Domains* property contains an array of all search and routing domains currently used by systemd−resolved. Each entry consists of a network interface index (again, 0 encodes system−wide entries), the actual domain name, and whether the entry is used only for routing (true) or for both routing and searching (false).

The *TransactionStatistics* property contains information about the number of transactions systemd−resolved has processed. It contains a pair of unsigned 64−bit counters, the first containing the number of currently ongoing transactions, the second the number of total transactions systemd−resolved is processing or has processed. The latter value may be reset using the **ResetStatistics()** method described above. Note that the number of transactions does not directly map to the number of issued resolver bus method calls. While simple look−ups usually require a single transaction only, more complex look−ups might result in more, for example when CNAMEs or DNSSEC are in use.

The *CacheStatistics* property contains information about the executed cache operations so far. It exposes three 64−bit counters: the first being the total number of current cache entries (both positive and negative), the second the number of cache hits, and the third the number of cache misses. The latter counters may be reset using **ResetStatistics()** (see above).

The *DNSSEC* property specifies current status of DNSSEC validation. It is one of "yes" (validation is enforced), "no" (no validation is done), "allow−downgrade" (validation is done if the current DNS server supports it). See the description of *DNSSEC*= in **resolved.conf**(5).

The *DNSSECStatistics* property contains information about the DNSSEC validations executed so far. It contains four 64−bit counters: the number of secure, insecure, bogus, and indeterminate DNSSEC validations so far. The counters are increased for each validated RRset, and each non−existance proof. The secure counter is increased for each operation that successfully verified a signed reply, the insecure counter is increased for each operation that successfully verified that an unsigned reply is rightfully unsigned. The bogus counter is increased for each operation where the validation did not check out and the data is likely to have been tempered with. Finally the indeterminate counter is increased for each operation which did not complete because the necessary keys could not be acquired or the cryptographic algorithms were unknown.

The *DNSSECSupported* boolean property reports whether DNSSEC is enabled and the selected DNS servers support it. It combines information about system−wide and per−link DNS settings (see below), and

only reports true if DNSSEC is enabled and supported on every interface for which DNS is configured and for the system−wide settings if there are any. Note that systemd−resolved assumes DNSSEC is supported by DNS servers until it verifies that this is not the case. Thus, the reported value may initially be true, until the first transactions are executed.

The *DNSOverTLS* boolean property reports whether DNS−over−TLS is enabled.

The *ResolvConfMode* property exposes how /etc/resolv.conf is managed on the host. Currently, the values "uplink", "stub", "static" (these three correspond to the three different files systemd−resolved.service provides), "foreign" (the file is managed by admin or another service, systemd−resolved.service just consumes it), "missing" (/etc/resolv.conf is missing).

The *DNSStubListener* property reports whether the stub listener on port 53 is enabled. Possible values are "yes" (enabled), "no" (disabled), "udp" (only the UDP listener is enabled), and "tcp" (only the TCP listener is enabled).

LINK OBJECT

```
node /org/freedesktop/resolve1/link/_1 {
 interface org.freedesktop.resolve1.Link {
  methods:
   SetDNS(in  a(iay) addresses);
   SetDNSEx(in  a(iayqs) addresses);
   SetDomains(in  a(sb) domains);
   SetDefaultRoute(in  b enable);
   SetLLMNR(in  s mode);
   SetMulticastDNS(in  s mode);
   SetDNSOverTLS(in  s mode);
   SetDNSSEC(in  s mode);
   SetDNSSECNegativeTrustAnchors(in  as names);
   Revert();
  properties:
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly t ScopesMask = ...;
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly a(iay) DNS = [...];
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly a(iayqs) DNSEx = [...];
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly (iay) CurrentDNSServer = ...;
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly (iayqs) CurrentDNSServerEx = ...;
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly a(sb) Domains = [...];
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly b DefaultRoute = ...;
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly s LLMNR = '...';
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly s MulticastDNS = '...';
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly s DNSOverTLS = '...';
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly s DNSSEC = '...';
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly as DNSSECNegativeTrustAnchors = ['...', ...];
   @org.freedesktop.DBus.Property.EmitsChangedSignal("false")
   readonly b DNSSECSupported = ...;
```

```
  };
  interface org.freedesktop.DBus.Peer { ... };
  interface org.freedesktop.DBus.Introspectable { ... };
  interface org.freedesktop.DBus.Properties { ... };
};
```

For each Linux network interface a "Link" object is created which exposes per−link DNS configuration and state. Use **GetLink()** on the Manager interface to retrieve the object path for a link object given the network interface index (see above).

Methods

The various methods exposed by the Link interface are equivalent to their similarly named counterparts on the Manager interface. e.g. **SetDNS()** on the Link object maps to **SetLinkDNS()** on the Manager object, the main difference being that the later expects an interface index to be specified. Invoking the methods on the Manager interface has the benefit of reducing roundtrips, as it is not necessary to first request the Link object path via **GetLink()** before invoking the methods. The same relationship holds for **SetDNSEx()**, **SetDomains()**, **SetDefaultRoute()**, **SetLLMNR()**, **SetMulticastDNS()**, **SetDNSOverTLS()**, **SetDNSSEC()**, **SetDNSSECNegativeTrustAnchors()**, and **Revert()**. For further details on these methods see the Manager documentation above.

Properties

ScopesMask defines which resolver scopes are currently active on this interface. This 64−bit unsigned integer field is a bit mask consisting of a subset of the bits of the flags parameter describe above. Specifically, it may have the DNS, LLMNR and MDNS bits (the latter in IPv4 and IPv6 flavours) set. Each individual bit is set when the protocol applies to a specific interface and is enabled for it. It is unset otherwise. Specifically, a multicast−capable interface in the "UP" state with an IP address is suitable for LLMNR or MulticastDNS, and any interface that is UP and has an IP address is suitable for DNS. Note the relationship of the bits exposed here with the LLMNR and MulticastDNS properties also exposed on the Link interface. The latter expose what is *configured* to be used on the interface, the former expose what is actually used on the interface, taking into account the abilities of the interface.

DNSSECSupported exposes a boolean field that indicates whether DNSSEC is currently configured and in use on the interface. Note that if DNSSEC is enabled on an interface, it is assumed available until it is detected that the configured server does not actually support it. Thus, this property may initially report that DNSSEC is supported on an interface.

DefaultRoute exposes a boolean field that indicates whether the interface will be used as default route for name queries. See **SetLinkDefaultRoute()** above.

The other properties reflect the state of the various configuration settings for the link which may be set with the various methods calls such as **SetDNS()** or **SetLLMNR()**.

COMMON ERRORS

Many bus methods systemd−resolved exposes (in particular the resolver methods such as **ResolveHostname()** on the Manager interface) may return some of the following errors:

org.freedesktop.resolve1.NoNameServers
No suitable DNS servers were found to resolve a request.

org.freedesktop.resolve1.InvalidReply
A response from the selected DNS server was not understood.

org.freedesktop.resolve1.NoSuchRR
The requested name exists, but there is no resource record of the requested type for it. (This is the DNS NODATA case).

org.freedesktop.resolve1.CNameLoop
The look−up failed because a CNAME or DNAME loop was detected.

org.freedesktop.resolve1.Aborted
The look−up was aborted because the selected protocol became unavailable while the operation was ongoing.

org.freedesktop.resolve1.NoSuchService
A service look−up was successful, but the **SRV** record reported that the service is not available.

org.freedesktop.resolve1.DnssecFailed
The acquired response did not pass DNSSEC validation.

org.freedesktop.resolve1.NoTrustAnchor
No chain of trust could be established for the response to a configured DNSSEC trust anchor.

org.freedesktop.resolve1.ResourceRecordTypeUnsupported
The requested resource record type is not supported on the selected DNS servers. This error is generated for example when an RRSIG record is requested from a DNS server that does not support DNSSEC.

org.freedesktop.resolve1.NoSuchLink
No network interface with the specified network interface index exists.

org.freedesktop.resolve1.LinkBusy
The requested configuration change could not be made because **systemd-networkd**(8), already took possession of the interface and supplied configuration data for it.

org.freedesktop.resolve1.NetworkDown
The requested look−up failed because the system is currently not connected to any suitable network.

org.freedesktop.resolve1.DnsError.NXDOMAIN, **org.freedesktop.resolve1.DnsError.REFUSED**, ...
The look−up failed with a DNS return code reporting a failure. The error names used as suffixes here are defined in by IANA in **DNS RCODEs**[4].

EXAMPLES

Example 1. Introspect org.freedesktop.resolve1.Manager on the bus

```
$ gdbus introspect −−system \
  −−dest org.freedesktop.resolve1 \
```

```
  --object-path /org/freedesktop/resolve1
```

Example 2. Introspect org.freedesktop.resolve1.Link on the bus

```
$ gdbus introspect --system \
  --dest org.freedesktop.resolve1 \
  --object-path /org/freedesktop/resolve1/link/_11
```

VERSIONING

These D-Bus interfaces follow **the usual interface versioning guidelines**[5].

NOTES

1. Writing Network Configuration Managers
   https://wiki.freedesktop.org/www/Software/systemd/writing-network-configuration-managers
2. Writing Resolver Clients
   https://wiki.freedesktop.org/www/Software/systemd/writing-resolver-clients
3. RFC 1035
   https://www.ietf.org/rfc/rfc1035.txt
4. DNS RCODEs
   https://www.iana.org/assignments/dns-parameters/dns-parameters.xhtml#dns-parameters-6
5. the usual interface versioning guidelines
   http://0pointer.de/blog/projects/versioning-dbus.html