# Using Avahi in Guile Scheme Programs

For Guile-Avahi 0.4.1

**Ludovic Courtès**

Edition 0.4.1
8 December 2010

# Table of Contents

# 1 Introduction

Guile-Avahi (`http://www.nongnu.org/guile-avahi/`) provides GNU Guile *bindings* to the Avahi library (`http://avahi.org/`). In other words, it makes it possible to write Scheme programs that use the facilities of Avahi. Avahi itself is a C library that implements the multicast DNS (`http://multicastdns.org/`) (mDNS) and DNS Service Discovery (`http://dns-sd.org/`) (DNS-SD) protocols, sometimes erroneously referred to as "Bonjour". Together, these protocols provide support for fully decentralized host naming and service publication and discovery. They are key components of the so-called Zero-Configuration Networking Stack (`http://www.zeroconf.org/`) (*Zeroconf*).

More precisely, Guile-Avahi provides bindings for the *client* library of Avahi. The client library allows application to use service discovery by transparently connecting them to the Avahi *system-wide daemon* using D-Bus (`http://dbus.freedesktop.org`). This daemon actually implements the DNS-SD protocol and handles service discovery and publication on behalf of applications running on the same host.

Thus, the functionality of Guile-Avahi could be provided to Guile Scheme applications by writing a D-Bus client to the Avahi daemon in Scheme. Alas, no Scheme-friendly D-Bus implementation was available at the time Guile-Avahi was started, hence the approach taken by Guile-Avahi.

This document describes the Scheme API to Avahi offered by Guile-Avahi. The reader is assumed to have basic knowledge of the protocol and library. Please send bug reports and comments to the Guile-Avahi mailing list.

# 2 Conventions

This chapter details the conventions used in Guile-Avahi, as well as specificities of the mapping of the C API to Scheme.

## 2.1 Enumerates and Constants

Lots of enumerates and constants are used in the Avahi C API. For each C enumerate type, a disjoint Scheme type is used—thus, enumerate values and constants are not represented by Scheme symbols nor by integers. This makes it impossible to use an enumerate value of the wrong type on the Scheme side: such errors are automatically detected by type-checking.

The enumerate values are bound to variables exported by the `(avahi)` and other modules within the `avahi` hierarchy. These variables are named according to the following convention:

- All variable names are lower-case; the underscore `_` character used in the C API is replaced by hyphen `-`.
- All variable names are prepended by the name of the enumerate type and the slash `/` character.
- In some cases, the variable name is made more explicit than the one of the C API, e.g., by avoid abbreviations.

Consider for instance this C-side enumerate:

```
typedef enum
{
  AVAHI_CLIENT_S_REGISTERING,
  AVAHI_CLIENT_S_RUNNING,
  AVAHI_CLIENT_S_COLLISION,
  AVAHI_CLIENT_FAILURE,
  AVAHI_CLIENT_CONNECTING
} AvahiClientState;
```

The corresponding Scheme values are bound to the following variables exported by the `(avahi client)` module:

```
client-state/s-registering
client-state/s-running
client-state/s-collision
client-state/failure
client-state/connecting
```

Hopefully, most variable names can be deduced from this convention.

Scheme-side "enumerate" values can be compared using `eq?` (see Section "Equality" in *The GNU Guile Reference Manual*). Consider the following example:

```
(let ((client (make-client ...)))

  ;;
  ;; ...
  ;;
```

```
;; Check the client state.
(if (eq? (client-state client) client-state/failure)
    (format #t "Oh, we failed.")))
```

In addition, all enumerate values can be converted to a human-readable string, in a type-specific way. For instance, `(watch-event->string watch-event/in)` yields `"in"`. Note that these strings may not be sufficient for use in a user interface since they are fairly concise and not internationalized.

## 2.2 Procedure Names

Unlike C functions in Avahi, the corresponding Scheme procedures are named in a way that is close to natural English. Abbreviations are also avoided. For instance, the Scheme procedure corresponding to `avahi_client_get_version` is named `client-server-version`. The `avahi_` prefix is always omitted from variable names since a similar effect can be achieved using Guile's nifty binding renaming facilities, should it be needed (see Section "Using Guile Modules" in *The GNU Guile Reference Manual*).

## 2.3 Explicit Finalization

Except for `client` objects, all objects created by the Avahi client API are local representative of objects implemented by the system-wide Avahi daemon. For instance, `make-service-browser` returns a "service browser" object which is actually a proxy to a daemon-implemented service browser (see Section 4.4 [Service Browsing], page 17). In other words, the Avahi daemon allocates resources (objects) on behalf of its clients.

While the Avahi daemon can reclaim resources allocated on behalf of a client program when that program exits, it cannot automatically determine when such resources become unneeded and reclaimable while the program is running. Thus, clients must *explicitly* tell the daemon when an object is no longer needed.

Consequently, except for `client` objects, objects manipulated by Guile-Avahi programs must be freed using the appropriate `free` procedure. For instance, objects created by `make-service-browser` must eventually be freed by `free-service-browser!`. Additional procedures are available to determine whether a particular object has already been freed; for instance, `freed-service-browser?` returns `#t` when the given service browser has already been freed. Of course, freed objects are no longer usable; procedures that are passed a previously freed object will raise an `error/invalid-object` exception (see Section 2.4 [Error Handling], page 5).

Note that all such client-side proxy objects are *not* subject to garbage collection until they have been explicitly freed. Therefore, it is important to free them when they are no longer needed!

As an exception, `client` objects as returned by `make-client` are subject to garbage collection and need not be explicitly freed. This is because client programs will usually create only one client object whose lifetime is that of the program itself.

## 2.4 Error Handling

Avahi errors are implemented as Scheme exceptions (see Section "Exceptions" in *The GNU Guile Reference Manual*). Each time a Avahi function returns an error, an exception with key `avahi-error` is raised. The additional arguments that are thrown include an error code and the name of the Avahi procedure that raised the exception. The error code is pretty much like an enumerate value: it is one of the `error/` variables exported by the `(avahi)` module (see Section 2.1 [Enumerates and Constants], page 3). Exceptions can be turned into error messages using the `error->string` procedure.

The following examples illustrates how Avahi exceptions can be handled:

```
(let ((poll (make-simple-poll)))

  ;;
  ;; ...
  ;;

  (catch 'avahi-error
    (lambda ()
      (run-simple-poll (simple-poll poll)))
    (lambda (key err function . currently-unused)
      (format (current-error-port)
              "an Avahi error was raised by `~a': ~a~%"
              function (error->string err)))))
```

Again, error values can be compared using `eq?`:

```
;; `avahi-error' handler.
(lambda (key err function . currently-unused)
  (if (eq? err error/no-daemon)
      (format (current-error-port)
              "~a: the Avahi daemon is not running~%"
              function)))
```

Note that the `catch` handler is currently passed only 3 arguments but future versions might provide it with additional arguments. Thus, it must be prepared to handle more than 3 arguments, as in this example.

# 3 Examples

This chapter lists examples that illustrate common use cases.

## 3.1 Publishing a Service

The following example shows the simplest way to publish a service. There are several stages:

- Create an Avahi client using `make-client`. This will actually connect the application to the Avahi daemon that will eventually perform operations on behalf of the application.
- When the client switches to the running state (i.e., `client-state/s-running`), create an *entry group* with `make-entry-group` and add a service (or several services, addresses, etc.) to it.
- Commit the entry group using `commit-entry-group`.
- When all entries in the group have been successfully published, the group's call-back is invoked and passed the `entry-group-state/established` state. The application must keep running so that the service remains published.

Here is the complete example:

```
(use-modules (avahi)
             (avahi client)
             (avahi client publish))

(define (group-callback group state)
  (if (eq? state entry-group-state/established)
      (format #t "service is now published!~%")))

(define client-callback
  (let ((group #f))
    (lambda (client state)
      (if (eq? state client-state/s-running)
          (begin
            ;; The client is now running so we can create an entry
            ;; group and publish a service.
            (set! group (make-entry-group client group-callback))
            (add-entry-group-service! group interface/unspecified
                                      protocol/unspecified '()
                                      "my-avahi-service"
                                      "_some-service-type._tcp"
                                      #f #f ;; any domain and host
                                      1234  ;; the port number

                                      ;; additional `txt' properties
                                      "scheme=yes" "java=no")

            ;; Commit the entry group, i.e., actually publish
            ;; the service.
            (commit-entry-group group))))))
```

```
;; The main loop.
(let* ((poll (make-simple-poll))
       (client (make-client (simple-poll poll)
                            '() ;; no flags
                            client-callback)))
  (and (client? client)

       ;; Run forever.
       (run-simple-poll poll)))
```

Of course, publishing a host address or service subtype works similarly.

## 3.2 Browsing Published Services

Browsing advertised services requires a number of stages. First, an Avahi daemon client must be created, as usual (see Section 3.1 [Publishing a Service], page 7).

```
(use-modules (avahi)
             (avahi client)
             (avahi client lookup))

(define %service-type
  ;; The type of services we are looking for.
  "_workstation._tcp")

(define (service-browser-callback browser interface protocol event
                                  service-name service-type
                                  domain flags)
  (if (eq? event browser-event/new)
      (format #t "found service `~a' of type `~a'~%"
              service-name service-type)))

(define client-callback
  (let ((browser #f))
    (lambda (client state)
      (if (eq? state client-state/s-running)
          ;; Now that the client is up and running, create a service
          ;; browser looking for services of type `%service-type' on
          ;; any network interface and using any protocol.
          (set! browser
                (make-service-browser client
                                      interface/unspecified
                                      protocol/unspecified
                                      %service-type #f '()
                                      service-browser-callback))))))


(let* ((poll (make-simple-poll))
```

```
        (client (make-client (simple-poll poll)
                             '() ;; no flags
                             client-callback)))
    (and (client? client)
         (run-simple-poll poll)))
```

In this example, the service type being looked for is `"_workstation._tcp"`. It is used to advertise the presence of computers on a local area network, rather than an actual service.

## 3.3 Resolving Services

The previous example allowed us to *find* services of a given type, but did not provide us with information such as the IP address of the host providing the service and the port number where the service can be found. To obtain this information, a *service resolver* must be launched, e.g., by augmenting the service browser call-back as follows:

```
(define (service-browser-callback browser interface protocol event
                                  service-name service-type
                                  domain flags)

  (define (service-resolver-callback resolver interface protocol event
                                     service-name service-type domain
                                     host-name address-type address port
                                     txt flags)
    ;; Handle service resolution events.
    (cond ((eq? event resolver-event/found)
           (format #t "resolved service '~a' at '~a:~a'~%"
                   service-name host-name port))
          ((eq? event resolver-event/failure)
           (format #t "failed to resolve service '~a'~%"
                   service-name))))

  (if (eq? event browser-event/new)
      (begin
        (format #t "found service '~a' of type '~a'~%"
                service-name service-type)

        ;; Launch a service resolver for the service we just found.
        (make-service-resolver (service-browser-client browser)
                               interface protocol
                               service-name service-type domain
                               protocol/unspecified '()
                               service-resolver-callback))))
```

Now you know all the important things you need to know to benefit from Avahi!

# 4 API Reference

This chapter documents Guile-Avahi Scheme procedures. Note that further details can be found in the Avahi C API reference (`http://avahi.org/`).

## 4.1 Core Interface

This section lists the Scheme procedures exported by the `(avahi)` module. These procedures are mainly related to *polls*, the building block of event loops in Avahi programs. Polls come in three flavors:

- The *simple poll* provides simple, single-threaded event dispatching. It essentially hangs on `select()`, processes D-Bus I/O events, and invokes the relevant client call-backs when appropriate.
- The *threaded poll* processes events similarly but in a separate thread of execution.
- Finally, the *guile poll* allows you to create customized event loops. This is useful, for instance, in single-threaded programs that process events coming not only from Avahi but also from other sources (e.g., GTK+ events, networking events, etc.).

Creating and manipulating polls is achieved using the procedures below.

`unlock-threaded-poll` *threaded-poll* [Scheme Procedure]
: Unlock the event look object associated with *threaded-poll*.

`lock-threaded-poll` *threaded-poll* [Scheme Procedure]
: Lock the event loop associated with *threaded-poll*. Use this if you want to access the event loop object (e.g., creating a new event source) from anything but the event loop helper thread, i.e. not from callbacks.

`quit-threaded-poll` *threaded-poll* [Scheme Procedure]
: Quit the event loop associated with *threaded-poll* responsible for running the event loop. It must be called from outside said thread (i.e., not from callbacks).

`stop-threaded-poll` *threaded-poll* [Scheme Procedure]
: Stop the helper thread associated with *threaded-poll* responsible for running the event loop. It must be called from outside said thread (i.e., not from callbacks).

`start-threaded-poll` *threaded-poll* [Scheme Procedure]
: Start the helper thread associated with *threaded-poll*, which is responsible for running the event loop. Callbacks are called from the helper thread. Thus, synchronization may be required among threads.

`threaded-poll` *threaded-poll* [Scheme Procedure]
: Return the `poll` object associated with *threaded-poll*.

`make-threaded-poll` [Scheme Procedure]
: Return a `threaded-poll` object. A threaded poll is essentially an event loop that processes events from the Avahi daemon in its own thread.

`run-simple-poll` *simple-poll* [Scheme Procedure]
: Run the event loop of *simple-poll* until either an error occurs or a quit request is scheduled. In the former case, an error is raised; in the latter, `#f` is returned.

`iterate-simple-poll` *simple-poll* [*sleep-time*] [Scheme Procedure]
Handle events registered by *simple-poll*. If *sleep-time* is not specified, the function blocks until an I/O event occurs. If *sleep-time* is specified, it is the maximum number of milliseconds of blocking. Return `#f` is a quit request has been scheduled, `#t` otherwise.

`simple-poll` *simple-poll* [Scheme Procedure]
Return the `poll` object associated with *simple-poll*.

`make-simple-poll` [Scheme Procedure]
Return a `simple-poll` object. This is the easiest way to handle I/O of Avahi `client` objects and similar.

`guile-poll` *guile-poll* [Scheme Procedure]
Return the `poll` object associated with *guile-poll*.

`make-guile-poll` *new-watch update-watch! free-watch new-timeout update-timeout! free-timeout* [Scheme Procedure]
Return a `guile-poll` object that can then be used to handle I/O events for Avahi objects such as clients. All arguments should be procedures:

- *new-watch* and *new-timeout* are invoked when the poll-using code requires a new file descriptor to be watched after, or a new timeout to be honored, respectively. *new-watch* is passed a `watch` object and a list of `watch-event` values; *new-timeout* is passed a *timeout* object and a number of seconds and nanoseconds representing the absolute date when the timeout expires, or `#f` if the newly created timeout is disabled.
- *update-watch!* and *update-timeout!* are called to modify a previously created watch or timeout. *update-watch!* is passed the `watch` object and a new list of events; *update-timeout!* is passed a new expiration time or `#f`.
- Finally, *free-watch* and *free-timeout* are called when the poll is asked to to no longer look handle them. For instance, when *free-watch* is called, the event loop code may remove the associated file descriptor from the list of descriptors passed to `select`.

The Guile-Avahi source code distribution comes with a detailed example.

`timeout?` *obj* [Scheme Procedure]
Return true if *obj* is of type `timeout`.

`watch?` *obj* [Scheme Procedure]
Return true if *obj* is of type `watch`.

`threaded-poll?` *obj* [Scheme Procedure]
Return true if *obj* is of type `threaded-poll`.

`guile-poll?` *obj* [Scheme Procedure]
Return true if *obj* is of type `guile-poll`.

`simple-poll?` *obj* [Scheme Procedure]
Return true if *obj* is of type `simple-poll`.

`poll?` *obj* [Scheme Procedure]
Return true if *obj* is of type `poll`.

`interface->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `interface` value.

`protocol->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `protocol` value.

`watch-event->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `watch-event` value.

`error->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `error` value.

The low-level API for watches, timeouts, and "guile polls", all of which serve as the basic for the creation of customized event loops (using `make-guile-poll`) is described below. In practice, you should only need it in applications where the Avahi event loop needs to be integrated in some other event loop; in other cases, the "simple poll" or "threaded poll" should be enough.

`set-timeout-user-data!` *timeout data* [Scheme Procedure]
Associated *data* (an arbitrary Scheme object) with *timeout*.

`timeout-user-data` *timeout* [Scheme Procedure]
Return the user-specified data associated with *timeout*.

`timeout-value` *timeout* [Scheme Procedure]
Return the expiration time for *timeout* as two values: the number of seconds and nanoseconds. If *timeout* is disabled, both values are `#f`.

`set-watch-user-data!` *watch data* [Scheme Procedure]
Associated *data* (an arbitrary Scheme object) with *watch*.

`watch-user-data` *watch* [Scheme Procedure]
Return the user-specified data associated with *watch*.

`watch-events` *watch* [Scheme Procedure]
Return the events of interest (a list of `watch-event/` values) for *watch*.

`watch-fd` *watch* [Scheme Procedure]
Return the file descriptor associated with *watch*.

`invoke-timeout` *timeout* [Scheme Procedure]
Invoke the call-back associated with *timeout*. This notifies the interested code that the timeout associated with *timeout* has been reached. The return value is unspecified. An `error/invalid-object` error is raised if *timeout* is disabled or is no longer valid.

`invoke-watch` *watch events* [Scheme Procedure]
Invoke the call-back associated with *watch*. This notifies the interested code that the events listed in *events* (a list of `watch-event/` values) occurred on the file descriptor associated with *watch*. The return value is unspecified. An `error/invalid-object` error is raised if *watch* is no longer valid.

## 4.2 Client Interface

This section lists the Scheme procedures exported by the `(avahi client)` module.

`client-state` *client* [Scheme Procedure]
Return the state (a `client-state/` value) of *client*.

`client-host-fqdn` *client* [Scheme Procedure]
Return the fully qualified domain name (FQDN) of the server *client* is connected to.

`client-host-name` *client* [Scheme Procedure]
Return the host name of the server *client* is connected to.

`client-server-version` *client* [Scheme Procedure]
Return the version (a string) of the server the client is connected to.

`make-client` *poll flags callback* [Scheme Procedure]
Return a new Avahi client. The client will use *poll* (a `poll` object as returned by, e.g., `(simple-poll (make-simple-poll))`) for I/O management. In addition, when the client state changes, `callback` (a two-argument procedure) will be invoked and passed the client object and a client-state value. *flags* must be a list of client flags (i.e., `client-flag/` values).

`client?` *obj* [Scheme Procedure]
Return true if *obj* is of type `client`.

`client-flag->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `client-flag` value.

`client-state->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `client-state` value.

The *flags* argument expected by `make-client` is a list containing zero or more values among the following:

`client-flag/ignore-user-config` [Scheme Variable]
Don't read user configuration.

`client-flag/no-fail` [Scheme Variable]
Don't fail if the daemon is not available when `make-client` is called; instead enter `client-state/connecting` state and wait for the daemon to appear.

## 4.3 Service Publication

The service publication API is provided by the `(avahi client publish)`. To publish services, one must first create a client for the Avahi daemon (see Section 4.2 [Client Interface], page 14).

`alternative-service-name` *service-name* [Scheme Procedure]
Find an alternative name to *service-name*. If called with an original service name, `" #2"` is appended. Afterwards the number is incremented on each call (i.e., `"foo"` becomes `"foo #2"`, which becomes `"foo #3"`, and so on).

`alternative-host-name` *hostname* [Scheme Procedure]
Find an alternative name to *hostname*. If called with an original host name, `"2"` is appended. Afterwards the number is incremented on each call (i.e., `"foo"` becomes `"foo2"`, which becomes `"foo3"`, and so on).

`add-entry-group-address!` *group interface protocol publish-flags fqdn address-protocol address* [Scheme Procedure]
Add to *group* a mapping from fully-qualified domain name *fqdn* to address *address*. Depending on *address-protocol* (a `protocol/` value), *address* should be a 32-bit or 128-bit integer (for IPv4 and IPv6, respectively) in host byte order (see Section "Network Address Conversion" in *The GNU Guile Reference Manual*).

`update-entry-group-service!` *group interface protocol publish-flags service-name service-type domain* [*txt...*] [Scheme Procedure]
Update the service named *service-name* in *group*.

`add-entry-group-service-subtype!` *group interface protocol publish-flags service-name service-type domain subtype* [Scheme Procedure]
Add *subtype* as a sub-type of a service already present in *group*. You may add as many subtypes for a service as you wish.

`add-entry-group-service!` *group interface protocol publish-flags service-name service-type domain host port* [*txt...*] [Scheme Procedure]
Add a service of type *service-type* (e.g., `"_http._tcp"`) named *service-name* to *group*. *port* should be an integer telling which port this service is listening on; *host* can be a string indicating which host it is running on, or `#f` to let the daemon decide by itself (recommended). Likewise, *domain* can be `#f` (recommended) or a string indicating the domain where this service is to be registered. Additionaly *txt* arguments should be string denoting additional `txt` properties (e.g., `"color-printer=yes"`). Finally, *interface* and *protocol* denote, respectively, the network interface and protocol used to publish the service. *interface* may be `interface/unspecified`, in which case the daemon will choose the most appropriate interface, or it can be a string (e.g., `"eth0"`), or an integer OS-provided integer index; similarly, *protocol* may be `protocol/unspecified`, in which case the daemon will choose a protocol, or it can be any other `protocol/` value.

`entry-group-client` *group* [Scheme Procedure]
Return the client used by *group*.

`entry-group-empty?` *group* [Scheme Procedure]
Return `#t` if *group* is empty, `#f` otherwise.

`entry-group-state` *group* [Scheme Procedure]
Return the state of *group*, i.e., an `entry-group-state/` value.

`reset-entry-group!` *group* [Scheme Procedure]
Reset *group*.

`commit-entry-group` *group* [Scheme Procedure]
Commit entry group *group*, i.e., register its entries on the network. It is an error to commit an empty group.

`make-entry-group` *client callback* [Scheme Procedure]
Return a new entry group using *client* and *callback* as the state-change notification procedure. *callback* should be a two-argument procedure. It will be passed the group object and the group entry's state (i.e., a `group-entry-state/` value).

`publish-flag->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `publish-flag` value.

`entry-group-state->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `entry-group-state` value.

`entry-group?` *obj* [Scheme Procedure]
Return true if *obj* is of type `entry-group`.

`freed-entry-group?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an object of type `entry-group` that has already been explicitly freed.

`free-entry-group!` *obj* [Scheme Procedure]
Explicitly free *obj*, an object of type `entry-group`.

The *publish-flags* argument expected by `add-entry-group-service!` and similar procedures is a list containing zero or more values among the following:

`publish-flag/unique` [Scheme Variable]
For raw records: The RRset is intended to be unique.

`publish-flag/no-probe` [Scheme Variable]
For raw records: Though the RRset is intended to be unique no probes shall be sent.

`publish-flag/no-announce` [Scheme Variable]
For raw records: Do not announce this RR to other hosts.

`publish-flag/allow-multiple` [Scheme Variable]
For raw records: Allow multiple local records of this type, even if they are intended to be unique.

`publish-flag/no-reverse` [Scheme Variable]
For address records: don't create a reverse (PTR) entry.

`publish-flag/no-cookie` [Scheme Variable]
For service records: do not implicitly add the local service cookie to TXT data.

`publish-flag/update` [Scheme Variable]
Update existing records instead of adding new ones.

`publish-flag/use-wide-area` [Scheme Variable]
Register the record using wide area DNS (i.e., unicast DNS update).

`publish-flag/use-multicast` [Scheme Variable]
Register the record using multicast DNS.

## 4.4 Service Browsing

The service discovery API is provided by the `(avahi client lookup)` module. Service discovery typically consists of two phases: *browsing* where one can find, e.g., available services, and *resolution* where one can, e.g., get detailed information about a discovered service such as its IP address.

All browsers and resolvers support the following *lookup flags*:

`lookup-flag/use-wide-area` [Scheme Variable]
Force lookup via wide-area DNS.

`lookup-flag/use-multicast` [Scheme Variable]
Force lookup via multicast DNS.

`lookup-flag/no-txt` [Scheme Variable]
When doing service resolving, don't lookup TXT record.

`lookup-flag/no-address` [Scheme Variable]
When doing service resolving, don't lookup A/AAAA record.

Procedures to create browsers and resolvers are described below.

`make-address-resolver` *client interface protocol address-type address lookup-flags callback* [Scheme Procedure]
Return a new address resolver using the specified *client*, *interface*, etc., that will resolve the host name corresponding to *address* of type *address-type* (either `protocol/inet` for an IPv4 address or `protocol/inet6` for an IPv6 address). As usual, *address* should be the raw IP address in host byte order (see Section "Network Address Conversion" in *The GNU Guile Reference Manual*). Upon resolution, *callback* is invoked and passed:

- the address resolver object;
- an interface name or number (depending on the OS);
- the protocol (i.e., one of the `protocol/` values);
- a resolver event type (i.e., one of the `resolver-event/` values);
- the host IP address type (i.e., *address-type*);
- the host IP address (i.e., *address*);
- the corresponding host name;
- lookup result flags (i.e., a list of `lookup-result-flag/` values).

An exception may be raised on failure.

`make-host-name-resolver` *client interface protocol host-name a-protocol lookup-flags callback* [Scheme Procedure]
Return a new host-name resolver using the specified *client*, *interface*, etc., that will resolve *host-name*, i.e., find the corresponding IP address. Upon resolution, *callback* is invoked and passed:

- the host-name resolver object;
- an interface name or number (depending on the OS);

- the protocol (i.e., one of the `protocol/` values);
- a resolver event type (i.e., one of the `resolver-event/` values);
- the host name;
- the host IP address type (i.e., `protocol/inet` for an IPv4 address and `protocol/inet6` for an IPv6 address);
- the host IP address in host byte order (see Section "Network Address Conversion" in *The GNU Guile Reference Manual*);
- lookup result flags (i.e., a list of `lookup-result-flag/` values).

An exception may be raised on failure.

`make-service-resolver` *client interface protocol service-name type domain a-protocol lookup-flags callback* [Scheme Procedure]

Return a new service resolver using the specified *client*, *interface*, etc., that will resolve the host name, IP address, port and `txt` properties of the service of type *type* named *service-name*. Upon resolution, *callback* is invoked and passed:

- the service type resolver object;
- an interface name or number (depending on the OS);
- the protocol (i.e., one of the `protocol/` values);
- a resolver event type (i.e., one of the `resolver-event/` values);
- the service name;
- the service type (e.g., `"_http._tcp"`);
- the domain;
- the host name (name of the host the service is running on);
- the host IP address type (i.e., `protocol/inet` for an IPv4 address and `protocol/inet6` for an IPv6 address);
- the host IP address in host byte order (see Section "Network Address Conversion" in *The GNU Guile Reference Manual*);
- a list of `txt` properties (strings);
- lookup result flags (i.e., a list of `lookup-result-flag/` values).

An exception may be raised on failure.

`make-service-browser` *client interface protocol type domain lookup-flags callback* [Scheme Procedure]

Return a new service browser using the specified *client*, *interface*, etc. Upon browsing events (discovery, removal, etc.) *callback* will be called and passed:

- the service browser object;
- an interface name or number (depending on the OS);
- the protocol (i.e., one of the `protocol/` values);
- a browser event type (i.e., one of the `browser-event/` values);
- the service name;
- the service type (e.g., `"_http._tcp"`);

- the domain;
- lookup result flags (i.e., a list of `lookup-result-flag/` values).

`make-service-type-browser` *client interface protocol domain lookup-flags callback* [Scheme Procedure]

Return a new service type browser using the specified *client*, *interface*, etc. Upon browsing events (discovery, removal, etc.) *callback* will be called and passed:

- the service type browser object;
- an interface name or number (depending on the OS);
- the protocol (i.e., one of the `protocol/` values);
- a browser event type (i.e., one of the `browser-event/` values);
- a service type (e.g., `"_http._tcp"`);
- the domain;
- lookup result flags (i.e., a list of `lookup-result-flag/` values).

`make-domain-browser` *client interface protocol domain domain-browser-type lookup-flags callback* [Scheme Procedure]

Return a new domain browser of type *domain-browser-type* (a `domain-browser-type/` value) for *domain* that uses *client*. Upon browsing events (discovery, removal, etc.) *callback* will be called and passed:

- the domain browser object;
- an interface name or number (depending on the OS);
- the protocol (i.e., one of the `protocol/` values);
- a browser event type (i.e., one of the `browser-event/` values);
- the domain;
- lookup result flags (i.e., a list of `lookup-result-flag/` values).

`address-resolver-client` *address-resolver* [Scheme Procedure]

Return the client associated with *address-resolver*.

`host-name-resolver-client` *host-name-resolver* [Scheme Procedure]

Return the client associated with *host-name-resolver*.

`service-resolver-client` *service-resolver* [Scheme Procedure]

Return the client associated with *service-resolver*.

`service-browser-client` *service-browser* [Scheme Procedure]

Return the client associated with *service-browser*.

`service-type-browser-client` *service-type-browser* [Scheme Procedure]

Return the client associated with *service-type-browser*.

`domain-browser-client` *domain-browser* [Scheme Procedure]

Return the client associated with *domain-browser*.

`lookup-result-flag->string` *enumval* [Scheme Procedure]

Return a string describing *enumval*, a `lookup-result-flag` value.

`lookup-flag->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `lookup-flag` value.

`resolver-event->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `resolver-event` value.

`browser-event->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `browser-event` value.

`domain-browser-type->string` *enumval* [Scheme Procedure]
Return a string describing *enumval*, a `domain-browser-type` value.

`address-resolver?` *obj* [Scheme Procedure]
Return true if *obj* is of type `address-resolver`.

`freed-address-resolver?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an object of type `address-resolver` that has already been explicitly freed.

`free-address-resolver!` *obj* [Scheme Procedure]
Explicitly free *obj*, an object of type `address-resolver`.

`host-name-resolver?` *obj* [Scheme Procedure]
Return true if *obj* is of type `host-name-resolver`.

`freed-host-name-resolver?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an object of type `host-name-resolver` that has already been explicitly freed.

`free-host-name-resolver!` *obj* [Scheme Procedure]
Explicitly free *obj*, an object of type `host-name-resolver`.

`service-resolver?` *obj* [Scheme Procedure]
Return true if *obj* is of type `service-resolver`.

`freed-service-resolver?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an object of type `service-resolver` that has already been explicitly freed.

`free-service-resolver!` *obj* [Scheme Procedure]
Explicitly free *obj*, an object of type `service-resolver`.

`service-type-browser?` *obj* [Scheme Procedure]
Return true if *obj* is of type `service-type-browser`.

`freed-service-type-browser?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an object of type `service-type-browser` that has already been explicitly freed.

`free-service-type-browser!` *obj* [Scheme Procedure]
Explicitly free *obj*, an object of type `service-type-browser`.

`service-browser?` *obj* [Scheme Procedure]
Return true if *obj* is of type `service-browser`.

`freed-service-browser?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an object of type `service-browser` that has already been explicitly freed.

`free-service-browser!` *obj* [Scheme Procedure]
Explicitly free *obj*, an object of type `service-browser`.

`domain-browser?` *obj* [Scheme Procedure]
Return true if *obj* is of type `domain-browser`.

`freed-domain-browser?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an object of type `domain-browser` that has already been explicitly freed.

`free-domain-browser!` *obj* [Scheme Procedure]
Explicitly free *obj*, an object of type `domain-browser`.

Browser and resolver call-backs are usually passed a browser event or resolver event value, respectively, among the following:

`browser-event/new` [Scheme Variable]
The object is new on the network.

`browser-event/remove` [Scheme Variable]
The object has been removed from the network.

`browser-event/cache-exhausted` [Scheme Variable]
One-time event, to notify the user that all entries from the caches have been sent.

`browser-event/all-for-now` [Scheme Variable]
One-time event, to notify the user that more records will probably not show up in the near future, i.e., all cache entries have been read and all static servers been queried.

`browser-event/failure` [Scheme Variable]
Browsing failed.

`resolver-event/found` [Scheme Variable]
RR found, resolving successful.

`resolver-event/failure` [Scheme Variable]
Resolving failed.

In addition, browser and resolver call-backs are passed a list *lookup result flags* which is a list of values among the following:

`lookup-result-flag/cached` [Scheme Variable]
This response originates from the cache.

`lookup-result-flag/wide-area` [Scheme Variable]
This response originates from wide area DNS.

`lookup-result-flag/multicast` [Scheme Variable]
This response originates from multicast DNS.

`lookup-result-flag/local` [Scheme Variable]
This record/service resides on and was announced by the local host. Only available in service and record browsers and only on `browser-event/new` events.

`lookup-result-flag/our-own` [Scheme Variable]
This service belongs to the same local client as the browser object. Only available for service browsers and only on `browser-event/new` events.

This is useful for applications that both publish and browse services to distinguish between services published by the application itself and services published from other applications.

`lookup-result-flag/static` [Scheme Variable]
The returned data has been defined statically by some configuration option.

# Concept Index

# Procedure Index

## R



## S



## T



## U



## W

# Variable Index