



[QUICKSTART](#) | [API](#) | [FAQ](#) | [Download](#) | [Source](#)

CloudI API Documentation

version 1.2.1b

last updated on March 30 2013

CloudI API - Making a Service

- [1.0 - Introduction](#)
- [1.1 - \(initialization\)](#)
- [1.2 - subscribe](#)
- [1.3 - unsubscribe](#)
- [1.4 - get_pid \(Erlang-only\)](#)
- [1.5 - send_sync](#)
- [1.6 - send_async](#)
- [1.7 - send_async_active \(Erlang-only\)](#)
- [1.8 - mcast_async](#)
- [1.9 - recv_async](#)
- [1.10 - return](#)
- [1.11 - forward](#)

CloudI Service API - Controlling CloudI

- [2.0 - Introduction](#)
- [2.1 - acl_add](#)
- [2.2 - acl_remove](#)
- [2.3 - services_add](#)
- [2.4 - services_remove](#)
- [2.5 - services_restart](#)
- [2.6 - services](#)
- [2.7 - nodes_add](#)
- [2.8 - nodes_remove](#)
- [2.9 - nodes_alive](#)

[2.10 - nodes_dead](#)
[2.11 - nodes](#)
[2.12 - loglevel_set](#)
[2.13 - log_redirect](#)
[2.14 - code_path_add](#)
[2.15 - code_path_remove](#)
[2.16 - code_path](#)

CloudI API - Making a Service

1.0 - Introduction

The CloudI API provides a simple messaging API which allows CloudI services to send requests. So, the CloudI API contains messaging primitives that can be used to emulate other messaging APIs, but normally the CloudI API is used directly. The CloudI API supports both publish/subscribe and point-to-point communication in an intuitive way. It is not necessary to understand the Erlang programming language, to use the CloudI API since a full CloudI API implementation is provided for every supported programming language (C/C++, Java, Python, and Ruby, currently).

The [subscribe](#) function subscribes to a service name pattern which can contain "*" wildcard characters, to accept any matching service requests. "*" within a service name pattern matches 1 or more characters, but "***" is forbidden. The [send_sync](#) function and the [send_async](#) function provide point-to-point communication based on the service name provided. When multiple services [subscribe](#) with the same service name pattern the destination is picked based on the sending service's "destination refresh method", which can be any of the following:

| Destination Refresh Method | Meaning |
|--|---|
| lazy_closest (or) immediate_closest | A service running on the local node will be selected, unless the destination only exists on a remote node |
| lazy_furthest (or) immediate_furthest | A service running on a remote node will be selected, unless the destination only exists on the local node |

| | |
|--------------------------------------|---|
| lazy_random (or) immediate_random | A service is selected randomly, based on the destination |
| lazy_local (or) immediate_local | Only a service on the local node is selected, based on the destination |
| lazy_remote (or) immediate_remote | Only a service on a remote node is selected, based on the destination |
| none | The service should never send a request and it is an error when the service attempts to send (the service may still receive requests) |

The "lazy" prefix and the "immediate" prefix on the destination refresh method determines whether stale data is used within the service's data or if a single Erlang lookup process is used to get the most current destination result, respectively ("lazy" is for when long-lived services are the destination but consumes more service memory, and "immediate" is for when short-lived services are the destination but creates contention for the Erlang lookup process).

If the same service subscribes with the same service name pattern more than once, each subscription is used in round-robin order (first subscription is called first, so order is preserved), when the service receives a request for the specific service name pattern.

The [mcast_async](#) function provides publish functionality by sending a request asynchronously to all services that have [subscribed](#) to the same service name pattern. To receive an asynchronous request [recv_async](#) is used with the "TransId" (i.e., Transaction Id, a v1 UUID) or a null UUID to receive the oldest service request.

The [return](#) function is used to respond to a service request and terminate the current request handler (i.e., the service request is finished, at that point). A service can [return](#) a null response if the sending service should not receive a response, which can be used for typical response-less publish functionality. The [forward](#) function provides a new destination for the same service request, delaying the request's completion, but still terminating the current request handler.

[Top](#)

1.1 - (initialization)

The service configuration will control the CloudI API initialization, which is done automatically, but does influence the source code. The service

configuration defines the number of Operating System (OS) processes to create and the number of threads for an external (non-Erlang) service. For an internal (Erlang) service, the configuration defines the number of Erlang processes to create. A number specified as an integer in the configuration is the exact number of processes or threads. However, if the number is specified as a floating point number, it is used as a CPU count (i.e., Erlang scheduler count) multiplier where >1.0 implies ceiling and <1.0 implies round. The external service APIs provide the `thread_count` function so that the total number of threads can be used for thread creation, with each thread holding an instance of the CloudI API (to avoid lock contention):

| Programming Language | Function Call |
|----------------------|---|
| C | <code>int cloudi_initialize_thread_count(unsigned int * const thread</code> |
| C++ | <code>unsigned int CloudI::API::thread_count();</code> |
| Java | <code>int org.cloudi.API.thread_count();</code> |
| Python | <code>cloudi_c.API.thread_count() cloudi.API.thread_count()</code> |
| Ruby | <code>CloudI::API.thread_count()</code> |

The service configuration also allows Access Control Lists (ACLs) to define explicit service name patterns for allowing or denying service destinations when the service sends a service request. The ACLs along with the destination refresh method determine how service requests are sent while other service options can tweak default settings:

| Service Configuration Parameter w/Default | Details |
|---|--|
| <code>priority_default=0</code> | $-128 \text{ (high)} \leq \text{priority_default} \leq 127 \text{ (low)}$ |
| <code>queue_limit=undefined</code> | A limit on the total number of incoming service requests that are queued while the service is busy (limits memory consumption) |
| <code>dest_refresh_start=500</code> | Delay after startup before requesting the initial service group membership (when using a lazy destination refresh method), |

specified in milliseconds.

`dest_refresh_delay=300000`

Maximum possible time for a service death to remove service group membership (when using a lazy destination refresh method), specified in milliseconds.

External (non-Erlang) services are provided both the command line and the environmental variables specified within the service configuration. External service configuration uses the full path to the executable while internal services use the module name (and the OTP application name) within the code search paths.

Please see [the CloudI Service API \(services_add\)](#) for more details about service configuration.

Specific Language Integration Notes:

The Erlang CloudI API functions shown below accept the most function parameters in `cloudi_service` but functions with less parameters do exist and they utilize default values for timeouts and request priority. Please see the [cloudi_service module](#) to see all the available functions.

Both the C and the C++ CloudI API rely on the same underlying code, with the C++ API object as a wrapper around the C API pointer, so there should be no large performance difference. STL is avoided, to avoid the libstdc++ memory pool and internal memory pools are used. The C++ CloudI API functions below use the `STRING` type to represent either `char const * const` (or) `std::string const &`, since both are supported with overloaded functions.

The Java CloudI API avoids any C or C++ integration. It only uses reflection to utilize the low-level file descriptor object and store object function pointers.

The python CloudI API is provided as both the "cloudi" module and the "cloudi_c" module. The "cloudi_c" module uses the C++ CloudI API for more efficiency, while the "cloudi" module only uses Python source code.

[Top](#)

1.2 - subscribe

**Programming
Language**

Function Call

| | |
|--------|--|
| Erlang | <pre>cloudi_service:subscribe(Dispatcher :: pid(), Pattern :: string, ok.</pre> |
| C | <pre>int cloudi_subscribe(cloudi_instance_t * p, char const * const pattern, cloudi_callback_t f);</pre> |
| C++ | <pre>template <typename T> int CloudI::API::subscribe(String pattern, T & object, void (T::*f) (CloudI::API const &, int const, String, String, void const * const, uint32_t const, void const * const, uint32_t const, uint32_t, int8_t, char const * const, char const * const, uint32_t const)) const; int CloudI::API::subscribe(String pattern, void (*f) (API const &, int const, String, String, void const * const, uint32_t const, void const * const, uint32_t const, uint32_t, int8_t, char const * const, char const * const, uint32_t const)) const;</pre> |
| Java | <pre>void org.cloudi.API.subscribe(final String pattern, final Object instance, final String methodName);</pre> |
| Python | <pre>cloudi_c.API.subscribe(pattern, Function) cloudi.API.subscribe(pattern, Function)</pre> |
| Ruby | <pre>CloudI::API.subscribe(pattern, function)</pre> |

Subscribes with a service name pattern which provides a destination for other services to send to. The subscribing service will receive a service request, if a different service sends a service request with a service name that matches the service name pattern. The service name pattern is a string that may contain a "*" wildcard character to match 1 or more characters, while "***" is forbidden. The service names and service name patterns are expected to be in a filepath format (e.g., "/root/directory/file.extension") by some provided CloudI services, though nothing enforces this convention. Good design dictates that service names operate within a given scope. Both the service names and the service name patterns should represent an appropriate scope, which the service manages (i.e., the same concept as a [Uniform Resource Identifier \(URI\)](#)).

When a service subscribes to a service name pattern, the supplied pattern string is appended to the service name prefix from the service's configuration, to provide the full service name pattern. The prefix provided within the service's configuration declares the scope of all service operations, as they are seen from other running services.

[Top](#)

1.3 - unsubscribe

Programming Language

Function Call

| | |
|--------|---|
| Erlang | <code>cloudi_service:unsubscribe(Dispatcher :: pid(), Pattern :: string) -> ok.</code> |
| C | <code>int cloudi_unsubscribe(cloudi_instance_t * p, char const * const pattern);</code> |
| C++ | <code>int CloudI::API::unsubscribe(STRING pattern) const;</code> |
| Java | <code>void org.cloudi.API.unsubscribe(final String pattern);</code> |
| Python | <code>cloudi_c.API.unsubscribe(pattern) cloudi.API.unsubscribe(pattern)</code> |
| Ruby | <code>CloudI::API.unsubscribe(pattern)</code> |

Unsubscribe will remove the service's subscription for the specific service name pattern. If a service has subscribed with the same service name

pattern multiple times, the unsubscribe will only remove one subscription instance. The subscription instance which is removed is whatever subscription would have been called next, for a matching service request.

[Top](#)

1.4 - get_pid (Erlang-only)

| Programming Language | Function Call |
|----------------------|---|
| Erlang | <pre>cloudi_service:get_pid(Dispatcher :: pid(), Name :: string(), Timeout :: non_neg_integer()) -> {'ok', PatternPid :: {string(), pid()}} {'error', Reason :: atom()}.</pre> |

Internal (Erlang-only) services can request an Erlang process based on the service name provided, before calling either the `send_sync` function or the `send_async` function. The `get_pid` function should rarely be necessary, but it can allow other logic to be used for determining which service should receive a request (e.g., based on apparent processing power, like within the hexpi test). The Erlang process (tuple) returned could become invalid if the service destination terminated, so the Erlang process monitoring becomes the burden of the `get_pid` function user. Due to the intimate nature of this function, it only exists within the Erlang CloudI API (to implement it in other languages would do more harm than good).

[Top](#)

1.5 - send_sync

| Programming Language | Function Call |
|----------------------|--|
| Erlang | <pre>cloudi_service:send_sync(Dispatcher :: pid(), Name :: string(), RequestInfo :: any(), Request :: any(), Timeout :: non_neg_integer() 'undefined', Priority :: integer() 'undefined') {'ok', ResponseInfo :: any(), Response :: any()} {'ok', Response :: any()} {'error', Reason :: atom()}.</pre> <pre>cloudi_service:send_sync(Dispatcher :: pid(),</pre> |

| | |
|--------|--|
| | <pre> Name :: string(), RequestInfo :: any(), Request :: any(), Timeout :: non_neg_integer() 'undefined', Priority :: integer() 'undefined', PatternPid :: {string(), pid()}) -> {'ok', ResponseInfo :: any(), Response :: any()} {'ok', Response :: any()} {'error', Reason :: atom()}. </pre> |
| C | <pre> int cloudi_send_sync_(cloudi_instance_t * p, char const * const name, void const * const request_info, uint32_t const request_info_size, void const * const request, uint32_t const request_size, uint32_t timeout, int8_t const priority); </pre> |
| C++ | <pre> int CloudI::API::send_sync(STRING name, void const * const request_info, uint32_t const request_info_size, void const * const request, uint32_t const request_size, uint32_t timeout, int8_t const priority) const; </pre> |
| Java | <pre> Response org.cloudi.API.send_sync(String name, byte[] request_info, byte[] request, Integer timeout, Byte priority); </pre> |
| Python | <pre> cloudi_c.API.send_sync(name, request, timeout=None, request_info=None, priority=None) cloudi.API.send_sync(name, request, timeout=None, request_info=None, priority=None) </pre> |
| Ruby | <pre> CloudI::API.send_sync(name, request, timeout=nil, request_info=nil, priority=nil) </pre> |

Send a synchronous request to a service name with a specific timeout and a specific priority. If a timeout is not provided, the default synchronous timeout from the service configuration is used. If a priority is not provided, the default priority from the service configuration options is used (normally the default priority is 0).

Function Return Values:

Programming Language

Return Value

ResponseInfo is only returned if it does not equal <<>>. Response is only returned if it does not equal <<>>.

Erlang

```
{'ok', ResponseInfo :: any(), Response :: any()}
{'ok', Response :: any()}
{'error', Reason :: atom()}
```

Separate functions are provided to get the function result after a successful send_sync function call (an integer 0 return value).

C

```
cloudi_get_response(p)
cloudi_get_response_size(p)
cloudi_get_response_info(p)
cloudi_get_response_info_size(p)
cloudi_get_trans_id_count(p)
cloudi_get_trans_id(p, i)
```

Separate functions are provided to get the function result after a successful send_sync function call (an integer 0 return value).

C++

```
char const * CloudI::API::get_response() const;
uint32_t CloudI::API::get_response_size() const;
char const * CloudI::API::get_response_info() const;
uint32_t CloudI::API::get_response_info_size() const;
uint32_t CloudI::API::get_trans_id_count() const;
char const * CloudI::API::get_trans_id(unsigned int const i =
```

A class encapsulates the function result.

Java

```
| org.cloudi.API.Response
```

A tuple provides the function result.

Python

```
| (response_info, response, trans_id)
```

An array provides the function result.

Ruby

```
| [response_info, response, trans_id]
```

The send_sync response data is provided in ways typical to each programming language, as shown above. The non-Erlang send_sync functions provide the TransId of the request because the calling service may need to use

the v1 UUID to manipulate and/or store the response.

[Top](#)

1.6 - send_async

Programming Language

Function Call

Erlang

```
cloudi_service:send_async(Dispatcher :: pid(),
                          Name :: string(),
                          RequestInfo :: any(),
                          Request :: any(),
                          Timeout :: non_neg_integer() | 'undefined',
                          Priority :: integer() | 'undefined') ->
  {'ok', TransId :: <<_:128>>} |
  {'error', Reason :: atom()}.

cloudi_service:send_async(Dispatcher :: pid(),
                          Name :: string(),
                          RequestInfo :: any(),
                          Request :: any(),
                          Timeout :: non_neg_integer() | 'undefined',
                          Priority :: integer() | 'undefined',
                          PatternPid :: {string(), pid()}) ->
  {'ok', TransId :: <<_:128>>} |
  {'error', Reason :: atom()}.

cloudi_service:send_async_passive(Dispatcher :: pid(),
                                  Name :: string(),
                                  RequestInfo :: any(),
                                  Request :: any(),
                                  Timeout :: non_neg_integer() | 'undefined',
                                  Priority :: integer() | 'undefined') ->
  {'ok', TransId :: <<_:128>>} |
  {'error', Reason :: atom()}.

cloudi_service:send_async_passive(Dispatcher :: pid(),
                                  Name :: string(),
                                  RequestInfo :: any(),
                                  Request :: any(),
                                  Timeout :: non_neg_integer() | 'undefined',
                                  Priority :: integer() | 'undefined',
                                  PatternPid :: {string(), pid()}) ->
  {'ok', TransId :: <<_:128>>} |
  {'error', Reason :: atom()}.
```

C

```
int cloudi_send_async_(cloudi_instance_t * p,
                      char const * const name,
                      void const * const request_info,
                      uint32_t const request_info_size,
```

| | |
|--------|---|
| | <pre> void const * const request, uint32_t const request_size, uint32_t timeout, int8_t const priority); </pre> |
| C++ | <pre> int CloudI::API::send_async(String name, void const * const request_info, uint32_t const request_info_size, void const * const request, uint32_t const request_size, uint32_t timeout, int8_t const priority) const; </pre> |
| Java | <pre> TransId org.cloudi.API.send_async(String name, byte[] request_info, byte[] request, Integer timeout, Byte priority); </pre> |
| Python | <pre> cloudi_c.API.send_async(name, request, timeout=None, request_info=None, priority=None) cloudi.API.send_async(name, request, timeout=None, request_info=None, priority=None) </pre> |
| Ruby | <pre> CloudI::API.send_async(name, request, timeout=nil, request_info=nil, priority=nil) </pre> |

Send an asynchronous request to a service name with a specific timeout and a specific priority. If a timeout is not provided, the default asynchronous timeout from the service configuration is used. If a priority is not provided, the default priority from the service configuration options is used (normally the default priority is 0).

An asynchronous send will block until a live service matches the service name destination or the timeout expires. Once the asynchronous request is sent the TransId which identifies the request is returned.

Function Return Values:

| Programming Language | Return Value |
|----------------------|---|
| Erlang | <pre> {'ok', TransId :: <<_:128>>} {'error', Reason :: atom()} </pre> |

Separate functions are provided to get the function result after a successful `send_async` function call (an integer 0 return value).

C

```
| cloudi_get_trans_id_count(p)
| cloudi_get_trans_id(p, i)
```

Separate functions are provided to get the function result after a successful `send_async` function call (an integer 0 return value).

C++

```
| uint32_t CloudI::API::get_trans_id_count() const;
| char const * CloudI::API::get_trans_id(unsigned int const i =
```

A class encapsulates the function result.

Java

```
| org.cloudi.API.TransId
```

The `trans_id` is a string of 16 bytes.

Python

```
| trans_id
```

The `trans_id` is a string of 16 bytes.

Ruby

```
| trans_id
```

The `send_async` result is provided in ways typical to each programming language, as shown above. A `TransId` is a v1 UUID.

[Top](#)

1.7 - `send_async_active` (Erlang-only)

Programming Language

Function Call

Erlang

```
cloudi_service:send_async_active(Dispatcher :: pid(),
                                Name :: string(),
                                RequestInfo :: any(),
                                Request :: any(),
                                Timeout :: non_neg_integer()
                                Priority :: integer() | 'undefined'),
{'ok', TransId :: <<_:128>>} |
{'error', atom()}.
cloudi_service:send_async_active(Dispatcher :: pid(),
                                Name :: string(),
```

| | |
|--|---|
| | <pre> RequestInfo :: any(), Request :: any(), Timeout :: non_neg_integer() Priority :: integer() 'und PatternPid :: {string(), pid </pre> |
| <pre> {'ok', TransId :: <<_:128>>} {'error', atom()}. </pre> | |

The `send_async_active` function provides the same functionality as the `send_async` function within an Erlang process, but the response is automatically sent to the Erlang process, after completion. Using `send_async_active` is the preferred way to send an asynchronous service request in Erlang because it utilizes Erlang's concurrency without requiring a blocking operation (a passive send, using Erlang vernacular, since it would otherwise require a call of the function `recv_async` to receive the request).

Incoming Process Message:

Programming Language

Messages

| | |
|--------|--|
| Erlang | <pre> {'return_async_active', Name :: string(), Pattern :: string(), ResponseInfo :: any(), Response :: any(), Timeout :: non_neg_integer(), TransId :: <<_:128>>} {'timeout_async_active', TransId :: <<_:128>>} </pre> |
|--------|--|

The `send_async_active` message is sent to the Erlang process as an Erlang message, so it arrives in the `cloudi_service_handle_info` function of the Erlang service module (i.e., the module that implements the [cloudi_service behavior](#)).

[Top](#)

1.8 - mcast_async

Programming Language

Function Call

| | |
|--------|--|
| Erlang | <pre> cloudi_service:mcast_async(Dispatcher :: pid(), Name :: string(), RequestInfo :: any(), Request :: any(), Timeout :: non_neg_integer() 'unc </pre> |
|--------|--|

| | |
|--------|---|
| | <pre> Priority :: integer() 'undefined' {'ok', TransIdList :: list(<<_:128>>)} {'error', Reason :: atom()}. cloudi_service:mcast_async(Dispatcher :: pid(), Name :: string(), RequestInfo :: any(), Request :: any(), Timeout :: non_neg_integer() 'unc Priority :: integer() 'undefined' PatternPid :: {string(), pid()}) -> {'ok', TransIdList :: list(<<_:128>>)} {'error', Reason :: atom()}. </pre> |
| C | <pre> int cloudi_mcast_async_(cloudi_instance_t * p, char const * const name, void const * const request_info, uint32_t const request_info_size, void const * const request, uint32_t const request_size, uint32_t timeout, int8_t const priority); </pre> |
| C++ | <pre> int CloudI::API::mcast_async(String name, void const * const request_info, uint32_t const request_info_size, void const * const request, uint32_t const request_size, uint32_t timeout, int8_t const priority) const; </pre> |
| Java | <pre> List<TransId> org.cloudi.API.mcast_async(String name, byte[] r byte[] request, Integer Byte priority); </pre> |
| Python | <pre> cloudi_c.API.mcast_async(name, request, timeout=None, request_info=None, prio cloudi.API.mcast_async(name, request, timeout=None, request_info=None, prior </pre> |
| Ruby | <pre> CloudI::API.mcast_async(name, request, timeout=nil, request_info=nil, priorit </pre> |

Multicast asynchronously, which is the same as publish, except that it is possible to respond to the service request. The function `mcast_async` will send the service request asynchronously to all services that have subscribed to a service name pattern that matches the service name destination. The

`mcast_async` function will block until at least a single request has been sent or the timeout has expired. The result of the function call is a list of TransIds (one TransId per service request). If a publish request is required, the destination service should have a null response (an empty binary of size 0), so that the service request response is ignored.

Function Return Values:

Programming Language

Return Value

| | |
|--------|---|
| Erlang | <pre>{'ok', TransIdList :: list(<_:128>)} {'error', Reason :: atom()}</pre> |
|--------|---|

Separate functions are provided to get the function result after a successful `send_async` function call (an integer 0 return value).

| | |
|---|---|
| C | <pre>cloudi_get_trans_id_count(p) cloudi_get_trans_id(p, i)</pre> |
|---|---|

Separate functions are provided to get the function result after a successful `send_async` function call (an integer 0 return value).

| | |
|-----|--|
| C++ | <pre>uint32_t CloudI::API::get_trans_id_count() const; char const * CloudI::API::get_trans_id(unsigned int const i =</pre> |
|-----|--|

A class encapsulates the function result.

| | |
|------|---|
| Java | <pre>List<org.cloudi.API.TransId></pre> |
|------|---|

The `trans_id` is a string of 16 bytes.

| | |
|--------|-----------------------|
| Python | <pre>[trans_id]</pre> |
|--------|-----------------------|

The `trans_id` is a string of 16 bytes.

| | |
|------|-----------------------|
| Ruby | <pre>[trans_id]</pre> |
|------|-----------------------|

The `mcast_async` result is provided in ways typical to each programming language, as shown above. A TransId is a v1 UUID.

[Top](#)

Programming Language

Function Call

| | |
|--------|---|
| Erlang | <pre>cloudi_service:recv_async(Dispatcher :: pid(), Timeout :: non_neg_integer(), TransId :: <<_:128>>, Consume :: boolean()) -> {'ok', ResponseInfo :: any(), Response :: any(), TransId :: any()} {'error', Reason :: atom()}.</pre> |
| C | <pre>int cloudi_recv_async(cloudi_instance_t * p, uint32_t timeout, char const * const trans_id, int consume);</pre> |
| C++ | <pre>int CloudI::API::recv_async(uint32_t timeout, STRING trans_id, bool consume) const;</pre> |
| Java | <pre>Response org.cloudi.API.recv_async(Integer timeout, byte[] trans_id, boolean consume);</pre> |
| Python | <pre>cloudi_c.API.recv_async(timeout=None, trans_id=None, consume=True) cloudi.API.recv_async(timeout=None, trans_id=None, consume=True)</pre> |
| Ruby | <pre>CloudI::API.recv_async(timeout=nil, trans_id=nil, consume=true)</pre> |

Receive an asynchronous service request's response. If a TransId is not provided, a null UUID is used to request the oldest response that has not timed out. By default, the `recv_async` function will consume the service request so it is not accessible with the same function call in the future. The TransId of the service request is always returned for any external use or tracking of the request or response.

Function Return Values:

Programming Language

Return Value

Response is only returned if it does not equal <<>>. ResponseInfo is always returned if Response is returned.

| | |
|--------|---|
| Erlang | <pre>{'ok', ResponseInfo :: any(), Response :: any(), TransId :: <_:_:128>} {'error', Reason :: atom()}</pre> |
|--------|---|

Separate functions are provided to get the function result after a successful `recv_async` function call (an integer 0 return value).

C

```
cloudi_get_response(p)
cloudi_get_response_size(p)
cloudi_get_response_info(p)
cloudi_get_response_info_size(p)
cloudi_get_trans_id_count(p)
cloudi_get_trans_id(p, i)
```

Separate functions are provided to get the function result after a successful `recv_async` function call (an integer 0 return value).

C++

```
char const * CloudI::API::get_response() const;
uint32_t CloudI::API::get_response_size() const;
char const * CloudI::API::get_response_info() const;
uint32_t CloudI::API::get_response_info_size() const;
uint32_t CloudI::API::get_trans_id_count() const;
char const * CloudI::API::get_trans_id(unsigned int const i =
```

A class encapsulates the function result.

Java

```
| org.cloudi.API.Response
```

A tuple provides the function result.

Python

```
| (response_info, response, trans_id)
```

An array provides the function result.

Ruby

```
| [response_info, response, trans_id]
```

[Top](#)

1.10 - return

Programming Language

Function Call

Erlang

```
cloudi_service:return(Dispatcher :: pid(),
                      Type :: 'send_async' | 'send_sync',
                      Name :: string(),
                      Pattern :: string(),
                      ResponseInfo :: any(),
                      Response :: any(),
                      Timeout :: non_neg_integer(),
```

| | |
|--------|--|
| | <pre> TransId :: <<_:128>>, Pid :: pid()) -> none(). </pre> |
| C | <pre> int cloudi_return(cloudi_instance_t * p, int const command, char const * const name, char const * const pattern, void const * const response_info, uint32_t const response_info_size, void const * const response, uint32_t const response_size, uint32_t timeout, char const * const trans_id, char const * const pid, uint32_t const pid_size); </pre> |
| C++ | <pre> int CloudI::API::return_(int const command, STRING name, STRING pattern, void const * const response_info, uint32_t const response_info_size, void const * const response, uint32_t const response_size, uint32_t timeout, char const * const trans_id, char const * const pid, uint32_t const pid_size) const; </pre> |
| Java | <pre> void org.cloudi.API.return_(Integer command, String name, String pattern, byte[] response_info, byte[] response, Integer timeout, byte[] transId, OtpErlangPid pid); </pre> |
| Python | <pre> cloudi_c.API.return_(command, name, pattern, response_info, response, timeout, trans_id, pid) cloudi.API.return_(command, name, pattern, response_info, response, timeout, trans_id, pid) </pre> |
| Ruby | <pre> CloudI::API.return_(command, name, pattern, response_info, response, timeout, trans_id, pid) </pre> |

Return a response to a service request. The return function will throw a caught exception so that the request handler execution is aborted after returning the service request response. The simplest and preferred way to

return a response within an Erlang service is to utilize the `cloudi_service_handle_request` function return values used by the [cloudi_service behavior](#). You can also utilize the request handler return value for the response in the programming languages Java, Python, and Ruby. However, within the external services it is more explicit (i.e., easier to understand the source code) when the source code uses the return functions.

If the service is configured with the `request_timeout_adjustment` option set to true (the default is false), the request handler execution time will automatically decrement the request timeout, after the request has been handled. If the service is configured with the `response_timeout_adjustment` option set to true (the default is false), the response timeout is automatically decremented based on the sender-side's timing (more accurate).

[Top](#)

1.11 - forward

Programming Language

Function Call

Erlang

```
cloudi_service:forward(Dispatcher :: pid(),
                      Type :: 'send_async' | 'send_sync',
                      Name :: string(),
                      RequestInfo :: any(),
                      Request :: any(),
                      Timeout :: non_neg_integer(),
                      Priority :: integer(),
                      TransId :: <<_:128>>,
                      Pid :: pid()) ->
    none().
```

C

```
int cloudi_forward(cloudi_instance_t * p,
                  int const command,
                  char const * const name,
                  void const * const request_info,
                  uint32_t const request_info_size,
                  void const * const request,
                  uint32_t const request_size,
                  uint32_t timeout,
                  int8_t const priority,
                  char const * const trans_id,
                  char const * const pid,
                  uint32_t const pid_size);
```

| | |
|--------|--|
| C++ | <pre>int CloudI::API::forward_(int const command, STRING name, void const * const request_info, uint32_t const request_info_size, void const * const request, uint32_t const request_size, uint32_t timeout, int8_t const priority, char const * const trans_id, char const * const pid, uint32_t const pid_size) const;</pre> |
| Java | <pre>Response org.cloudi.API.forward_(Integer command, String name, byte[] request_info, byte[] request, Integer timeout, Byte priority, byte[] transId, OtpErlangPid pid);</pre> |
| Python | <pre>cloudi_c.API.forward_(command, name, request_info, request, timeout, priority, trans_id, pid) cloudi.API.forward_(command, name, request_info, request, timeout, priority, trans_id, pid)</pre> |
| Ruby | <pre>CloudI::API.forward_(command, name, request_info, request, timeout, priority, trans_id, pid)</pre> |

Forward the service request to a different destination, possibly with different parameters (e.g., a completely different request). The forward function will throw a caught exception so that the request handler execution is aborted after forwarding the service request. The simplest and preferred way to forward a request within an Erlang service is to utilize the `cloudi_service_handle_request` function return values used by the [cloudi_service behavior](#). All external services must use a forward function when forwarding a request.

If the service is configured with the `request_timeout_adjustment` option set to true (the default is false), the request handler execution time will automatically decrement the request timeout, after the request has been handled. If the service is configured with the `response_timeout_adjustment` option set to true (the default is false), the response timeout is automatically decremented based on the sender-side's timing (more accurate).

[Top](#)

CloudI Service API - Controlling CloudI

2.0 - Introduction

When CloudI is first started, the configuration file at [/usr/local/etc/cloudi/cloudi.conf](#) is used to determine what Access Control Lists (ACLs) should be used for services, what services should be started, what nodes should be connected, and what logging should occur. All the configuration functionality for CloudI can be done dynamically, after startup, with the CloudI Service API. A typical way to use the Service API is with either erlang terms or JSON-RPC over HTTP.

| Protocol | Example |
|----------|---|
| Erlang | <code>curl http://localhost:6467/cloudi/api/erlang/services</code> |
| JSON-RPC | <code>curl -X POST -d '{"method": "services", "params": [], "id": 1}' http://localhost:6467/cloudi/api/json_rpc/</code> |

The data returned in both examples is Erlang terms within a string. All of the examples below use the Erlang protocol.

[Top](#)

2.1 - acl_add

```
curl -X POST -d ' [{sensitive, ["/accouting/", "/finance/"]} ] '
http://localhost:6467/cloudi/api/erlang/acl_add
```

Add more ACL entries to be later used when starting services. An ACL entry is an Erlang atom() -> list(atom() | string()) relationship which provides a logical grouping of service name patterns (e.g., {api, ["/cloudi/api/"]}). When providing a service name pattern for an ACL entry, a non-pattern will be assumed to be a prefix (i.e., "/cloudi/api/" == "/cloudi/api/*").

[Top](#)

2.2 - acl_remove

```
curl -X POST -d '[sensitive]' http://localhost:6467/cloudi
/api/erlang/acl_remove
```

Remove ACL entries that are no longer needed. Running services will retain their configuration, so this impacts services that are started in the

future.

[Top](#)

2.3 - services_add

```
curl -X POST -d '[{external, "/tests/flood/", "tests/flood
/service/flood", "1 tcp 16384", [{"LD_LIBRARY_PATH", "api/c
/lib/"}, {"DYLD_LIBRARY_PATH", "api/c/lib/"}], none, tcp,
16384, 5000, 5000, 5000, [api], undefined, 1, 1, 5, 300,
[]}, {internal, "/tests/flood/", cloudi_service_flood,
[{flood, "/tests/flood/c", <<"DATA">>, 1000}],
lazy_closest, 5000, 5000, 5000, [api], undefined, 2, 5,
300, []}]' http://localhost:6467/cloudi/api/erlang
/services_add
```

Start services. Provide service configuration using the same syntax found in the configuration file (i.e., [/usr/local/etc/cloudi/cloudi.conf](#)). Internal services will need to be located in a code path that the running Erlang VM is aware of (see [code_path_add](#)). The syntax of the configuration entries is shown below:

```
{internal,
 (ServiceNamePrefix),
 (ErlangModuleName),
 (ModuleInitializationList),
 (DestinationRefreshMethod),
 (InitializationTimeout in milliseconds),
 (DefaultAsynchronousTimeout in milliseconds),
 (DefaultSynchronousTimeout in milliseconds),
 (DestinationDenyACL),
 (DestinationAllowACL),
 (ProcessCount),
 (MaxR),
 (MaxT in seconds),
 (ServiceOptionsPropList)}

{external,
 (ServiceNamePrefix),
 (ExecutableFilePath),
 (ExecutableCommandLineArguments),
 (ExecutableEnvironmentalVariables),
 (DestinationRefreshMethod),
 (Protocol, use tcp),
 (ProtocolBufferSize, use 16384 on Linux),
 (InitializationTimeout in milliseconds),
 (DefaultAsynchronousTimeout in milliseconds),
 (DefaultSynchronousTimeout in milliseconds),
 (DestinationDenyACL),
```

```
(DestinationAllowACL),
(ProcessCount),
(ThreadCount),
(MaxR),
(MaxT in seconds),
(ServiceOptionsPropList)}
```

The ACL lists contain either atoms that reference the current ACL configuration or strings. If an ACL string is not a pattern, it is assumed to be a prefix (i.e., "*" is appended to make it a pattern). The ProcessCount and ThreadCount can be specified as integers for an exact count or as a floating point number to provide a CPU multiplier ($X < 1.0$ is round, $X > 1.0$ is floor). MaxR is the maximum restarts allowed within MaxT seconds (same parameters used by Erlang supervisors). The ServiceOptionsPropList provides the configurable defaults:

| Option | Default | Details |
|-----------------------------|-----------|---|
| priority_default | 0 | -128(high) \leq priority \leq 127(low) |
| queue_limit | undefined | A limit on the total number of incoming service requests that are queued while the service is busy (limits memory consumption) |
| dest_refresh_start | 500 | Delay after startup (in milliseconds) before requesting the initial service group membership (when using a lazy destination refresh method) |
| dest_refresh_delay | 300000 | Maximum possible time (in milliseconds) for a service death to remove service group membership (when using a lazy destination refresh method) |
| request_timeout_adjustment | false | Should the service request handler execution time decrement the request timeout, after the request has been handled. |
| response_timeout_adjustment | false | Should the service's incoming response timeout be automatically decremented |

| | | |
|---------------------|----------|---|
| | | based on the sender-side's timing (more accurate). |
| duo_mode | false | (internal services only) Use two Erlang processes instead of one Erlang process, so that more incoming service throughput can be handled with low latency. |
| request_pid_uses | 1 | (internal services only) How many service requests to handle before utilizing a new Erlang process for a new incoming service request. |
| request_pid_options | [] | (internal services only) erlang:spawn_opt/2 options to control memory usage of the service request handling Erlang process (fullsweep_after, min_heap_size, min_bin_vheap_size). |
| info_pid_uses | infinity | (internal services only) How many info messages to handle before utilizing a new Erlang process for a new incoming info message. This Erlang process is the second process that is utilized when duo_mode is true (duo_mode requires that this is set to infinity). |
| info_pid_options | [] | (internal services only) erlang:spawn_opt/2 options to control memory usage of the info message handling Erlang process (fullsweep_after, min_heap_size, min_bin_vheap_size). |

Please see the configuration file [/usr/local/etc/cloudi/cloudi.conf](#) for more specific examples.

[Top](#)

2.4 - services_remove

```
curl -X POST -d
' [<<110,129,240,166,122,31,17,226,212,14,165,221,0,0,0,88>>,
  <<110,129,240,236,122,31,17,226,212,14,165,221,0,0,0,88>>] '
http://localhost:6467/cloudi/api/erlang/services_remove
```

Provide the Service UUIDs for the services that should be stopped. The Service UUID is shown in the output of [services](#). When the service is stopped, its running instance is removed from CloudI, but does not impact any other running instances (even if they are the same service module or binary).

[Top](#)

2.5 - services_restart

```
curl -X POST -d
' [<<106,103,84,112,122,31,17,226,212,14,165,221,0,0,0,88>>] '
http://localhost:6467/cloudi/api/erlang/services_restart
```

Provide the Service UUIDs for the services that should be restarted. The Service UUID is shown in the output of [services](#). When the service is restarted, the old instance is stopped and a new instance is started. During the restart delay, it is possible to lose queued service requests and received asynchronous responses. Keeping the state separate between the service instances is important to prevent failures within the new instance.

[Top](#)

2.6 - services

```
curl http://localhost:6467/cloudi/api/erlang/services
```

List the service configuration parameters with each service's UUID.

[Top](#)

2.7 - nodes_add

```
curl -X POST -d "['cloud001@cluster1']" http://localhost:6467
/cloudi/api/erlang/nodes_add
```

Explicitly add a CloudI node name, so that services between all other CloudI nodes and the added nodes can send each other service requests.

[Top](#)

2.8 - nodes_remove

```
| curl -X POST -d "['cloud001@cluster1']" http://localhost:6467  
    /cloudi/api/erlang/nodes_remove
```

Explicitly remove a CloudI node name.

[Top](#)

2.9 - nodes_alive

```
| curl http://localhost:6467/cloudi/api/erlang/nodes_alive
```

List all the CloudI nodes known to be connected.

[Top](#)

2.10 - nodes_dead

```
| curl http://localhost:6467/cloudi/api/erlang/nodes_dead
```

List all the CloudI nodes that are disconnected but expected to reconnect.

[Top](#)

2.11 - nodes

```
| curl http://localhost:6467/cloudi/api/erlang/nodes
```

List both the connected and disconnected CloudI nodes.

[Top](#)

2.12 - loglevel_set

```
| curl -X POST -d 'warn' http://localhost:6467/cloudi/api/erlang  
    /loglevel_set
```

Modify the loglevel. CloudI uses asynchronous logging with flow control (backpressure handling) to prevent misbehaving services from causing instability.

[Top](#)

2.13 - log_redirect

```
| curl -X POST -d 'cloudi@host' http://localhost:6467/cloudi  
    /api/erlang/log_redirect
```

Redirect all local log output to a remote CloudI node. Use 'undefined' as the node name to log locally.

[Top](#)

2.14 - code_path_add

```
| curl -X POST -d '"/home/user/code/services"'  
    http://localhost:6467/cloudi/api/erlang/code_path_add
```

Add a directory to the CloudI Erlang VM code server's search paths. The path is always appended to the list of search paths (you should not need to rely on search path order because of unique naming).

[Top](#)

2.15 - code_path_remove

```
| curl -X POST -d '"/home/user/code/services"'  
    http://localhost:6467/cloudi/api/erlang/code_path_remove
```

Remove a directory from the CloudI Erlang VM code server's search paths. This doesn't impact any running services, only services that will be started in the future.

[Top](#)

2.16 - code_path

```
| curl http://localhost:6467/cloudi/api/erlang/code_path
```

List all the CloudI Erlang VM code server search paths (in the same order the directories are searched).

[Top](#)

