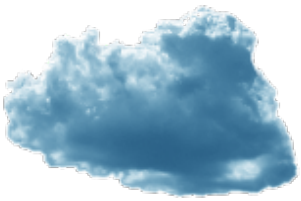
CloudI



Frequently Asked Questions

version 1.2.0b
last updated on February 18 2013

1 - Introduction to CloudI

2 - Learning about CloudI

3 - CloudI Installation Guide

4 - General Questions

5 - Migrating to CloudI

6 - Services

7 - Databases

1 - Introduction to CloudI

1.1 - Why is it named "CloudI"?

A **Cloud** is more dynamic than a 3 dimensional **Grid** and is more ubiquitous than the legend of **Beowulf**, so it is easy to understand why computing Clouds are the next generation distributed systems. The relevant connotations the word Cloud contains are: dynamic, supervision, intermingle, and points (i.e., point clouds). Any computing Cloud should offer dynamic configuration, should supervise processes in a fault-tolerant way, offer easy integration and should support an arbitrarily large number of processes (respectively). This project offers Cloud functionality facilitated by the Erlang programming language and its implementation of the Actor Model.

CloudI has an "I" suffix for several connotations: cloudy, one, singularity, interface, and independence. CloudI is referred to as "A Cloud as an Interface" because a light-weight interface facilitates Cloud functionality. The interface supports multiple programming languages and is called the CloudI API. CloudI supports private cloud development and deployment, so only one Cloud is necessary for Cloud functionality with implicit security. CloudI is also able to facilitate online services and offers extreme connection scalability.

Top

1.2 - How is CloudI pronounced?

As "cloud-e" /klaʊdi/ (think: Cloud Erlang).

Top

1.3 - How does CloudI compare to other "Clouds"?

Currently, "Clouds" generally fall into two categories:

- Infrastructure as a Service (IaaS) - Hypervisor "Clouds"
- Platform as a Service (PaaS) - Integration "Clouds"

Hypervisor "Clouds"

Hypervisor "Clouds" are the most popular type of Cloud because of the amount of revenue they can generate as a service. Popular examples include: Amazon Web Services (AWS), OpenStack, CloudStack, Eucalyptus, OpenNebula, and Nimbus. The Hypervisor has existed since 1965 when software was used on the IBM 360/65 to emulate an IBM 7080 with computation time split between the separate modes. Modern Hypervisors

provide Operating System virtualization to provide better security and reliability. There is meant to be minimal software development effort when utilizing a virtualized Operating System, so it is an obvious choice for source code that is not actively developed (legacy software) and lacks reliability/scalability. Part of the reason Hypervisors have not been popular in the past is because virtualization increases the hardware requirements for the same amount of processing. Hardware has advanced enough that many software applications are unable to fully utilize the hardware capacity that has become commonplace. For software that is often idle, Hypervisors can provide cost savings on both hardware and power without software modifications.

Integration "Clouds"

Integration "Clouds" provide software developers with a platform for simpler integration development. Popular examples include: AppScale, CloudFoundry, OpenShift, and Heroku. Generally, Integration "Clouds" provide software packages for common scripting language deployment scenarios (typically Python and Ruby web frameworks). Integration "Clouds" (PaaS) normally do not provide fault-tolerance or reliability, so they are typically deployed with a Hypervisor.

CloudI is an Integration Cloud that focuses on flexible integration that is efficient, scalable, and fault-tolerant. CloudI does not force a user to use particular software libraries but instead provides light-weight interfaces for integration. Scalability and fault-tolerance are both provided by CloudI's usage of the Erlang programming language. This means that no Hypervisor is necessary to make CloudI's processes reliable, so there can be a performance benefit when using CloudI. Scalability is a natural gain with CloudI's Erlang concurrency which reduces the amount of power and hardware necessary to facilitate external connections, making CloudI a greener solution!

Top

1.4 - What is CloudI?

Short Answer

An application server that efficiently integrates with many languages, many databases, and many messaging buses in a way that is both scalable and fault-tolerant.

Shorter Answer

A rock-solid transaction processing system for flexible software development.

Shortest Answer

A Cloud at the lowest level.

Long Answer

CloudI is an implementation of Cloud functionality that can be developed and deployed publicly or privately. CloudI provides a simple server back-end that can be used for infrastructure development of data processing systems, event processing systems, web services, and combinations thereof. CloudI is a system that enforces RESTful development practices and provides a Service Oriented Architecture (SOA). CloudI services communicate with messaging that can be controlled by simple Access Control List (ACL) entries (to provide service communication isolation).

CloudI was architected to easily integrate with other services, software, and frameworks. The CloudI API provides a light-weight interface for creating services in C++/C, Erlang, Java, Python, and Ruby. By using CloudI, external software can become more scalable and fault-tolerant by utilizing CloudI's load balancing of CloudI requests. CloudI messaging enforces realtime constraints using timeouts, so that request failures can be handled locally within the service where they are most relevant. ACL entries explicitly allow or deny communication between services and are a simple method of isolating critical services from potentially volatile services. All CloudI API usage in languages other than Erlang receive the isolation of Operating System processes and are called external services. External services can utilize the CloudI API with any threading library to achieve greater scalability and reduce internal latency. The Erlang CloudI API is used to create internal services which utilize light-weight Erlang processes. Examples of using the CloudI API are provided as integration tests or internal services.

The Service API provides dynamic configuration which is accessible from any allowed CloudI service (i.e., allowed based on the ACL entries). The Service API is accessible remotely by using Erlang terms or JSON-RPC over HTTP. Examples of using the Service API are provided as separate integration tests.

Top

1.5 - On what Operating Systems does CloudI run?

CloudI runs on UNIX-based operating systems like Linux (Ubuntu, etc.) and BSDs (FreeBSD, OpenBSD, NetBSD, OSX, etc.). CloudI development has primarily taken place on Ubuntu and other Operating Systems may not be completely tested yet. Windows may work by using Cygwin for dependencies.

Erlang must be able to run on the system for CloudI to function properly. So, checking Erlang support would be a good place to start if you are experimenting with a different Operating System. The information here will be updated as more Operating Systems are tested.

Top

1.6 - Is Commercial support available for CloudI?

- Integration Development
- Operations Maintenance

Contact Michael Truog if you are interested in commercial CloudI support.

Top

1.7 - Is CloudI really free?

CloudI is completely free. CloudI uses a BSD license which permits reuse for personal or commercial purposes. Small amounts of source code is included that is under the Erlang Public License (e.g., part of the Java CloudI API and cpg.erl) like Erlang itself. All external source code dependencies are also under a BSD license. Some conditional external source code dependencies (not included by default) are under other licenses (e.g., ZeroMQ is under the LGPL license). For a more detailed look at the licenses of external dependencies, please check the src/external/README.

Top

1.8 - Who develops CloudI?

Michael Truog

Top

1.9 - Can I use CloudI as a Private Cloud?

Yes! CloudI provides everything for running a Cloud in isolation (i.e., without a connection to the Internet). For more details, please refer to "1.4 - What is CloudI?".

Top

1.10 - Can I use CloudI as an Online Service?

Yes! CloudI accepts incoming HTTP traffic and can be easily extended to handle other incoming protocols. For more details, please refer to "1.4 - What is CloudI?".

Top

1.11 - What CAP theorem guarantees does CloudI provide?

CloudI is an AP-type distributed system (guarantees of Availability and Partition tolerance). A Consistency guarantee (the guarantee not provided by CloudI) can be provided by either a CloudI service interface to a database driver or a CloudI service interface to a messaging bus (i.e., to a persistent message queue). In both cases, a request can be sent to the CloudI service with the CloudI API (if the response is stored, the request succeeded).

Top

1.12 - Does CloudI support REST?

Yes! CloudI is a system that enforces RESTful development practices. For more details please refer to "1.4 - What is CloudI?".

Top

1.13 - Is CloudI a Platform as a Service (PaaS)?

Yes! CloudI can be used as a Platform as a Service (PaaS) and is the first fault-tolerant PaaS open source project. CloudI is not limited to the development of web services and has a broader focus. CloudI also does not enforce particular development libraries on the programmer, so it is a much more flexible framework. For more details please refer to "1.4 - What is CloudI?".

Top

1.14 - Why doesn't CloudI integrate with ProductX?

There are many possibilities for CloudI integration. If you know of a public product that you think should be integrated or if you need commercial support for a private product, contact Michael Truog.

Top

2 - Learning about CloudI

2.1 - Web Pages

Main Web Site:	http://cloudi.org
Source Code:	https://github.com/okeuday/CloudI
Releases:	http://sourceforge.net/projects/cloudi/files/

Top

2.2 - Mailing List

Email Address:	cloudi-questions@googlegroups.com
Subscribe:	http://groups.google.com/group/cloudi-questions/subscribe
Archive:	http://groups.google.com/group/cloudi-questions

Top

2.3 - Internet Relay Chat (IRC)

IRC Server:	irc.freenode.net
Chat Room:	#cloudi (#erlang can offer additional help, if necessary)

Top

2.4 - RSS Feeds

Development:	https://github.com/feeds/okeuday/commits/CloudI/develop
Releases:	http://sourceforge.net/api/file/index/project-id/281423/mtime/desc/limit/20/rss

Top

2.5 - Twitter

Development:	@cloudi_org

Top

2.6 - Presentations

Version 1.0.0	2012 Open Source Bridge Unconference
Version 0.1.6	2011 ErLounge Meetup Vancouver BC (slides)
Version 0.1.5	2011 ErLounge Meetup SF Bay Area (slides)
Version 0.0.9	2010 Erlang Factory SF Bay Area (slides) (demo text)
Version 0.0.8	2009 Erlang User Conference (video) (slides)

Top

2.7 - Reporting Bugs

Bug Reports:	https://github.com/okeuday/CloudI/issues/new
Mailing List:	cloudi-questions@googlegroups.com

If you are unsure whether you have found a bug, please send an email to the mailing list or utilize the IRC chat room. Otherwise, you can easily enter a bug report for the problem by using the online form.

Top

3 - CloudI Installation Guide

3.1 - Overview

Installation of CloudI from source (within the archive's "src" directory) uses the typical open source command sequence of:

```
1. ./configure
2. make
3. sudo make install
```

All the supported languages are currently required for the configuration, so that the generated configuration uses valid paths and the integration tests can be run. So, that means that the configuration will expect a C compiler, a C++ compiler, Java Development Kit (JDK), Python (≥ 2.7), Ruby (≥ 1.9), and Erlang (≥ R14B02). CloudI is tested on each release of Erlang (currently R16A). Dependencies as they are packaged for different operating systems are listed below:

Operating System	Packages
Ubuntu 12.04 (apt-get install <package(s)>)	• erlang • g++ • libboost-thread-dev • libboost-dev • default-jdk • python • python-dev • ruby1.9.1 • libgmp3-dev • uuid-dev
OSX w/macports (port install <package(s)>)	• erlang • libstdcxx • boost • python27 • ruby19 • gmp

Top

3.2 - Installation Options

Common CloudI installation configuration options ("./configure" command line arguments) are:

--prefix="/path/to/install/" Specify an Installation Path (default="/usr

	/local/")
--with-zeromq	Include ZeroMQ support
--with-zeromq-version=[2|3]	ZeroMQ major version (2 is the default)

For more installation configuration option details, please execute "./configure --help" (otherwise, you can refer to src/INSTALL for basic configuration information).

Top

3.3 - OS X Installation

To install CloudI dependencies on OSX you either need macports or homebrew. All configuration and build steps are the same as Linux.

Top

3.4 - Running CloudI

To start CloudI, execute:

```
sudo cloudi start
```

To stop the running CloudI node, execute:

```
sudo cloudi stop
```

When CloudI is running, CloudI logging output will be appended to PREFIX/var/logs/cloudi/cloudi.log.

Top

3.5 - Configuration

The CloudI configuration provides all the initial parameters for startup. It is also possible to do the same configuration with the CloudI Service API (so, the configuration can also be done dynamically as described in "4.2 - How do I control CloudI dynamically?").

The configuration is organized into sections for the ACLs, Services, Nodes, and Logging. The ACLs provide a name which can be referenced by a Service to either explicitly allow or deny communication between services (based on service name prefixes, see "4.6 - How do I use Access Control Lists (ACLs)?" for more information).

The Services configuration specifies both the services that are ran and the order in which the services should be started. The "internal" Services are Erlang modules that use the cloudi_service behavior. The "external" Services

are all non-Erlang languages that use the CloudI API. There is more information about service integration in "6 - Services" and "4.1 - How do I integrate external software with CloudI?".

The Nodes configuration lists all CloudI nodes that should be connected. This allows the CloudI node connections to reconnect after network failures.

The Logging configuration specifies the logging level and whether the logging output should be directed to a different CloudI node (which is present in the Nodes section). If the logging is redirected to a different CloudI node, it is possible to lose logging data when a network outage occurs. However, if the node has failed, the logging output will be stored locally until the node reconnects (i.e., the logging output is redirected to the CloudI node automatically, when it is connected).

Below is a summary of the layout of the CloudI configuration file. The ()s have been used to specify the configuration parameters that are supplied. You can find this file in src/cloudi.conf.in within the source code repository (in its state before it gets modified by the local operating system configuration parameters) or PREFIX/etc/cloudi/cloudi.conf after the installation.

```
{acl, [
    {(AliasName), [(ServiceNamePrefix) or (AliasName), ...]}
    ...
]}.
{services, [
    {internal,
     (ServiceNamePrefix),
     (ErlangModuleName),
     (ModuleInitializationList),
     (DestinationRefreshMethod),
     (InitializationTimeout),
     (DefaultAsynchronousTimeout),
     (DefaultSynchronousTimeout),
     (DestinationDenyACL),
     (DestinationAllowACL),
     (ProcessCount),
     (MaxR),
     (MaxT),
     (ServiceOptionsPropList)},
    {external,
     (ServiceNamePrefix),
     (ExecutableFilePath),
     (ExecutableCommandLineArguments),
     (ExecutableEnvironmentalVariables),
     (DestinationRefreshMethod),
     (Protocol),
     (ProtocolBufferSize),
     (InitializationTimeout),
     (DefaultAsynchronousTimeout),
     (DefaultSynchronousTimeout),
     (DestinationDenyACL),
     (DestinationAllowACL),
     (ProcessCount),
     (ThreadCount),
```

```
      (MaxR),
      (MaxT),
      (ServiceOptionsPropList)},
     ...
]}.
{nodes, [
    'cloudi@hostname1',
    ...
] % or 'automatic' for automatic LAN node configuration
}.
{logging, [
    {level, trace}, % levels: off, fatal, error, warn, info, debug, trace
    {redirect, undefined or (Node)}
]}.
```

The default configuration runs a variety of integration tests which are used to test CloudI:

- hexpi
- http
- http_req
- zeromq
- msg_size
- messaging

Some of the tests are explained within "6 - Services". All of the tests can be found within the source code repository in src/tests. There is no reason to keep the tests within the configuration once you start using CloudI for your own integration.

Top

4 - General Questions

4.1 - How do I integrate external software with CloudI?

There are many integration points for external software to become CloudI services or utilize CloudI services. The current integration points are:

- CloudI API
- ZeroMQ
- HTTP
- Supported databases
  - CouchDB
  - memcached
  - MySQL
  - PostgreSQL

- Tokyo Tyrant

CloudI API

The CloudI API provides a light-weight interface for creating services in C++/C, Erlang, Java, Python, and Ruby. Services subscribe to receive requests from other services using the CloudI API "subscribe" function call. The subscribe function call takes a suffix string that is expected to contain a path using a forward slash '/' (e.g., /cloudi/api/json_rpc/). However, the service configuration provides the prefix for the subscription function call, so "/cloudi /api/" is provided as a configuration prefix (for the Service API service) but the subscribe function call only needs to be called with the string "json_rpc/" so that a subscription takes place for any services sending requests to "/cloudi /api/json_rpc/", which is called a "name".

The requests are load balanced across all the services that have subscribed to the same name during the lookup to find the request destination. There is a service configuration parameter called the "destination refresh" that determines how the internal CloudI load balancing occurs when a request is sent from that service. The possible destination refresh values are:

- lazy_closest
- lazy_random
- immediate_closest
- immediate_random
- none

The "none" destination refresh is used for services that never send requests (i.e., they only receives requests) and creates an error that terminates the service if the service does send a request. The "lazy" prefix destination refresh methods use an older cached value for determining service destinations, so services that communicate primarily with long-lived services can use a "lazy" prefix destination refresh for more scalable communication. The "immediate" prefix destination refresh methods always use current information for determining service destinations, so services that communicate primarily with short-lived services can always send to relevant destinations. The "closest" suffix destination refresh methods always prefer services that exist on the local CloudI node, over remote CloudI nodes. The "random" suffix destination refresh methods load balances evenly across all services on all CloudI nodes.

The following functions exist in the CloudI API for sending a request:

- send_async
- send_sync
- mcast_async

The "send" prefix functions send a binary message (uninterpreted raw data) to a single service name (which is then load balanced among the available services). If the service name does not exist, the request will be retried until

the request timeout elapses and no binary data will be returned (i.e., returning no data is equivalent to a timeout). If a service receives a request while handling an older request, the request is queued based on its priority, where -128 is the highest priority, 0 is the default priority and 127 is the lowest priority. The "mcast" prefix function provides publish functionality, so a binary message is published to all services that have subscribed to a single service name. However, the "mcast" prefix function is slightly different from other publish functionality because it returns all the transaction ids (UUIDs used to uniquely identify a request among all CloudI nodes) so that responses (if any are returned) may be retrieved. A service can utilize publish behavior that doesn't return data by simply returning no data (since returning no data is equivalent to a timeout). The "async" suffix functions (i.e., asynchronous) only return the transaction id of the sent request(s) so that the response may be queried with the "recv_async" function. The "recv_async" function can also be used with a null UUID to return the oldest response that was received. If no services are available for the name of the destination, the "async" suffix function will block until the destination is found to send the request by retrying the send until the timeout elapses (i.e., the asynchronous sends are asynchronous after the send takes place). The "sync" suffix function will block until a response is returned or the timeout elapses. If a response is returned with no data, a timeout will be returned instead. If the request destination name is blocked by an Access Control List (ACL) entry, a timeout will be returned immediately from the send function.

When a service receives a request, it is passed as a parameter to the callback function. The callback function was specified as an argument to the "subscribe" function. However, in Erlang all requests use the same callback function which is cloudi_service_handle_request/11. Within the callback function any send or receive operations can take place. When the callback function wants to terminate it can either return a result or forward the request to another service name by using the "return" function or the "forward" function, respectively. If the service does not want to return a response, the service can simply call "return" with an empty binary response value and it will be interpreted as if the request timeout elapsed. Using the "forward" function will decrease the request timeout slightly (by 100ms) to prevent requests from causing persistent traffic.

The Access Control List (ACL) is simply a list of strings that define patterns that must be explicitly allowed or denied when determining if a service can send to the service name. A pattern uses "*" to represent a ".+" regex (one or more characters) while "**" is forbidden, which is the same format used for service subscriptions. Previous ACL usage in CloudI used prefixes, which is still valid, but a "*" is appended to create a pattern. This means that only patterns are accepted and no exact service names will be valid ACL strings.

If an ACL pattern is both allowed and denied, the pattern is denied (deny takes precedence). When defining ACLs, it is possible to use Erlang atoms to

represent lists of string patterns so that logical groupings are created. The ACL atoms are then able to be specified anywhere an ACL string might be present. So, it is best to group ACL string patterns based on context to simplify the configuration specification.

The CloudI API external service requests are limited to 2GB when using the tcp protocol. Using the udp protocol for external services is experimental (it limits service requests to the minimum of both the loopback MTU and the buffer size). The buffer size for external services is typically set to 16384 bytes because that is a power of 2 closest to the MTU of the loopback device (normally 16436 on Linux). External service configuration can specify the number of threads per process and the number of processes which should be spawned, so that each thread receives an instance of the CloudI API. This means that there can be one ioloop per thread per process for maximum throughput.

ZeroMQ

ZeroMQ integration provides a way of connecting to external ZeroMQ messaging or other CloudI nodes by using ZeroMQ as the messaging bus. The cloudi_service_zeromq service is an Erlang service that provides ZeroMQ integration by defining a set of mappings between service names and the ZeroMQ destinations. To use ZeroMQ with CloudI, you need to make sure and enable ZeroMQ with the configuration script (with "./configure --with-zeromq"). The cloudi_service_zeromq configuration (in the cloudi.conf file or through the Service API services_add/1 function) allows key/value tuples with the following key atoms: outbound, inbound, publish, subscribe, push, and pull, which are the following ZeroMQ equivalents: ZMQ_REQ, ZMQ_REP, ZMQ_PUB, ZMQ_SUB, ZMQ_PUSH, and ZMQ_PULL, respectively. The value is a tuple that contains a mapping key/value where the key is the service name suffix and the value is the list of ZeroMQ endpoints. However, the publish and subscribe ZeroMQ configuration is slightly more complex because instead of a service name, it contains a list of key/value ZeroMQ subscription mapping where the key is the service name suffix and the value is the ZeroMQ subscription string. The example configuration file entry below should illustrate the ZeroMQ service configuration:

```
% an entry in the cloudi.conf configuration file
% that uses the ZeroMQ service
{internal,
    "/tests/zeromq/",
    % inbound/outbound message paths much be acyclic
    % (if they are not, you will receive a erlzmq EFSM error
    %  because the ZeroMQ REQ has received 2 zmq_send calls)
    cloudi_service_zeromq,
    % outbound ZeroMQ requests connect a CloudI name to a ZeroMQ endpoint
    [{outbound, {"zigzag_start", ["ipc:///tmp/cloudizigzagstart"]}},
    % inbound ZeroMQ replies connect a ZeroMQ endpoint to a CloudI name
     {inbound, {"zigzag_step1", ["ipc:///tmp/cloudizigzagstart"]}},
     {outbound, {"zigzag_step1", ["inproc://zigzagstep1"]}},
     {inbound, {"zigzag_step2", ["inproc://zigzagstep1"]}},
```

```
% ZeroMQ publish connects a CloudI name to a ZeroMQ (subscribe) name
% as {CloudI name (suffix), ZeroMQ name for message prefix}
% for any number of endpoints
 {publish, {[{"zigzag_step2", "/zeromq/step2"}],
            ["inproc://zigzagstep2a",
             "ipc:///tmp/cloudizigzagstep2b",
             "inproc://zigzagstep2c",
             "ipc:///tmp/cloudizigzagstep2d"]}},
% ZeroMQ subscribe connects a CloudI name to a ZeroMQ (subscribe) name
% as {CloudI name (suffix), ZeroMQ name for subscribe setsocketopt}
% for any number of endpoints
 {subscribe, {[{"zigzag_step3a", "/zeromq/step2"},
               {"zigzag_step3b", "/zeromq/step2"}],
              ["inproc://zigzagstep2a",
               "ipc:///tmp/cloudizigzagstep2b",
               "inproc://zigzagstep2c",
               "ipc:///tmp/cloudizigzagstep2d"]}},
 {outbound, {"zigzag_step3a", ["inproc://zigzagstep3"]}},
 {inbound, {"zigzag_finish", ["inproc://zigzagstep3"]}}],
immediate_closest,
5000, 5000, 5000, [api], undefined, 2, 5, 300, []}
```

HTTP

The Erlang service cloudi_service_http accepts HTTP traffic and makes the HTTP requests CloudI requests where the HTTP path in the URL is used as the service name. By default, the HTTP method is specified as a suffix on the HTTP path (e.g., "/index.html/get") but this can be disabled with the "use_method_suffix" configuration parameter. When a HTTP request is received the corresponding service name will be called with the request contents (uncompressed, if the request was compressed). The headers are passed within the "request info" as key-value pairs that is request meta-data. The content type of the response is either forced by the configuration (with "content_type") or it is determined by the file extension on the service name.

Supported Databases

All the supported databases can be accessed by CloudI services. The CloudI Erlang service that provides database support (e.g., cloudi_service_db_pgsql, cloudi_service_db_mysql, etc.) uses the database name as the service name suffix. Services can send requests to the database service name in the appropriate format to interact with the database. The format to send is either SQL for an SQL database or a command tuple if it is a NoSQL database (e.g., {'set', "key", "value"}).

Top

4.2 - How do I control CloudI dynamically?

CloudI's configuration can be changed dynamically while it is running by using the Service API. The Service API can be used by any CloudI services.

However, typical usage of the Service API would use raw HTTP requests or JSON-RPC over HTTP. A complex example of using the Service API through JSON-RPC over HTTP with python code can be found in src/tests/service_api/run.py. Some simpler examples of using the Service API can be found at src/tests/service_api/path.py, src/tests/service_api/logging_off.py and src/tests/service_api/logging_on.py.

Top

4.3 - How do I use Publisher/Subscriber messaging?

The simplest way to use publisher/subscriber functionality is to use the CloudI API functions "mcast_async" for publishing and "subscribe" for subscribing. For more details please refer to the CloudI API documentation.

Top

4.4 - How do I use Remote Procedure Calls (RPC)?

Remote procedure calls can easily be used within CloudI services with a CloudI API "send_sync" function call. The RPC procedure name is used as a service name suffix and the RPC parameters are stored in the request body. The request body is simply uninterpreted binary data, so no format is imposed on the user of the CloudI API. Any request meta-data should be specified as key-value pairs within the "request info" parameter. The "response info" parameter can be used for response meta-data in the same way. For more details please refer to the CloudI API documentation.

Top

4.5 - How do I create Web Services?

Web Services are simply CloudI services that accept incoming HTTP traffic coming from the cloudi_service_http service. The request body is either the body of the uncompressed PUT or POST request, or it is the GET query string. For more details please refer to the CloudI API documentation.

Top

4.6 - How do I use Access Control Lists (ACLs)?

Access Control Lists (ACLs) are used to explicitly allow or deny requests from being sent to service name patterns. A pattern uses "*" to represent a ".+" regex (one or more characters) while "**" is forbidden, which is the same format used for service subscriptions. Two separate ACL parameters are specified for each service configuration to allow or deny destinations. If an ACL is not provided, the atom 'undefined' is used instead. An ACL is provided as a list of strings that are service name patterns. Instead of a string, an atom alias

may be provided that was defined in the 'acl' configuration so that the service configuration is simpler and more consistent (i.e., without strings that are replicated among the service configuration entries). A fake sample from a configuration file can illustrate how this works:

```
{acl, [
    {alias1, ["/service/name/prefix1", "/service/name/prefix2*", alias2]},
    {alias2, ["/subsystem1/prefix1*", "/subsystem2/prefix1"]}
]}.
{services, [
    {internal,
     (ServiceNamePrefix),
     (ErlangModuleName),
     (ModuleInitializationList),
     (DestinationRefreshMethod),
     (InitializationTimeout),
     (DefaultAsynchronousTimeout),
     (DefaultSynchronousTimeout),

     % ACL DENY LIST
     % (e.g, valid values could be: undefined or [alias1] or [alias2] or etc.)
     (DestinationDenyList),

     % ACL ALLOW LIST
     % (e.g, valid values could be: undefined or [alias1] or [alias2] or etc.)
     (DestinationAllowList),

     (ProcessCount),
     (MaxR),
     (MaxT),
     (ServiceOptionsPropList)},
    {external,
     (ServiceNamePrefix),
     (ExecutableFilePath),
     (ExecutableCommandLineArguments),
     (ExecutableEnvironmentalVariables),
     (DestinationRefreshMethod),
     (Protocol),
     (ProtocolBufferSize),
     (InitializationTimeout),
     (DefaultAsynchronousTimeout),
     (DefaultSynchronousTimeout),

     % ACL DENY LIST
     % (e.g, valid values could be: undefined or [alias1] or [alias2] or etc.)
     (DestinationDenyList),

     % ACL ALLOW LIST
     % (e.g, valid values could be: undefined or [alias1] or [alias2] or etc.)
     (DestinationAllowList),

     (ProcessCount),
     (ThreadCount),
     (MaxR),
     (MaxT),
     (ServiceOptionsPropList)},
]}.
...
```

The Service API supports dynamically starting services by supplying a 'services' list in the same format as the configuration file. The Service API also supports defining multiple 'acl' aliases that may be referenced from dynamically configured services.

Top

4.7 - How do I Migrate a Service from a Failed or Failing Node?

A migration would imply that there is unavoidable latency during a switchover from a failed node to a healthy node. To avoid failover latency and improve scalability, services are replicated on all nodes. Proper service implementation dictates that services will only cache data. All dynamic state a service uses should be accessed and/or stored by a database. To communicate with a database, a service should use the CloudI API to send requests to a configured CloudI database integration service. The implementation of services that avoids state-keeping within the service's data structures is required to make sure a service is scalable, fault-tolerant and can recover from a failure without losing a significant amount of data.

So, a service should not need to be migrated from a node. If a node has failed there are many possible courses of action:

- Shutdown CloudI on the Failed Node
- Stop the Service on the Failed Node by using the Service API
- Disconnect the Failed Node from the Network to Diagnose in Isolation

Since services are replicated on other nodes the system is fault-tolerant and can operate without a failed node.

Top

4.8 - Can I use Regular Expressions with Service Names (URLs)?

A simpler substitute for regular expressions is provided for matching CloudI service names. The "*" character (a wildcard character) is used to match 1 or more character within a service name (i.e., a regex of '.+'). However, the sequence "**" is invalid and will cause the operation to fail. Any number of wildcard characters can be used with the subscribe and unsubscribe CloudI API functions to create service names that match many patterns. While this approach may seem unusual, it helps keep service name lookups both efficient and parallel (i.e., within the Erlang code, without any need to call an external regex integration library).

Another possibility is just using explicit service names, even when the service name contains a dynamic parameter. Using all possible service names

is bounded by the memory available. To give an idea of the memory consumption, on a 64-bit machine using service names that contain a single dynamic integer, 1 million integers used within 1 million subscribe CloudI API calls will consume roughly 100 MB of RAM when the CloudI service is ran (i.e., the service that performs the subscribe CloudI API calls). All other CloudI services that use a "lazy" destination refresh method will replicate the service name data structure, so that will increase the node's memory consumption. So, depending on your needs and your memory limitations, you may want to use explicit service names or service names with wildcard characters.

Top

4.9 - Is the CloudI API thread-safe?

The CloudI API is not thread-safe (i.e., it is not reentrant) because it is meant to be used by individual threads that are configured within the CloudI service configuration (e.g., using the CloudI configuration file or the CloudI Service API). This approach avoids any lock contention issues outside of the Erlang VM.

Top

4.10 - How can CloudI requests take advantage of cache coherency, minimum network latency, and any logical grouping?

To provide better computing node grouping, service names should uniquely describe the context of the node. If the context is provided, then there is a natural grouping for CloudI requests with any CloudI API usage that uses the associated service name(s).

Top

4.11 - Why not just use Erlang directly?

Erlang does naturally support integration in the following ways:

- NIF (Native Interface Function)
- port drivers
- port
- cnode

NIFs and port drivers can integrate with external source code (normally only C or C++) as a dynamic library that is loaded by the Erlang VM. This approach is the most efficient and the most error-prone (any memory corruption impacts the Erlang VM to create new and exciting system crashes, sabotaging the fault-tolerance the Erlang VM provides). A port is an external executable ran as a separate OS process linked to the Erlang VM,

communicating over pipes. A cnode is a separate executable communicating as an Erlang node with the distributed Erlang protocol.

CloudI's external service execution is most similar to Erlang port integration. However, CloudI provides many additional features for external services that are normally not present:

- A service container abstraction for simpler Service Oriented Architecture (SOA) development
- Protocol agnostic transactional communication with a small consistent API (i.e., the CloudI API)
- Service name grouping with pattern matching
- Every service request receives a unique UUID (Universally Unique IDentifier)
- All external services are managed with fault-tolerance constraints, in the same way as Erlang/OTP supervisors
- Every service request can utilize 255 possible request priorities
- TCP sockets are used instead of pipes to maximize atomic send throughput (the localhost MTU is 16436 on Linux, but pipes are typically 4096 on Linux (PIPE_BUF))
- stdout and stderr output is logged automatically to the single CloudI log
- A service does not require a node connection (like a cnode does), so it doesn't have service-count scalability problems (due to a distributed Erlang node being a member of a fully connected network topology)

The CloudI API is consistent for all the supported programming languages, which makes it easier to move service functionality inbetween programming languages or inbetween services. All external CloudI services communicate in the same way and all service requests are processed in the same way, to create a consistent integration framework. Using CloudI naturally reduces the complexity of integration source code so that errors are more specific to the business logic being developed, because CloudI is continuously tested to ensure it provides both a stable and dependable integration framework.

Top

5 - Migrating to CloudI

5.1 - Performance Considerations

There is a latency penalty for communicating with a non-Erlang CloudI service because of the extra binary encoding and decoding when using the socket that connects the CloudI Erlang VM to the non-Erlang CloudI service

Operating System (OS) process' thread. The preemption of an Erlang VM scheduler thread by a CloudI service OS thread may degrade Erlang VM performance because of a mismatch between the kernel scheduler and the Erlang VM scheduler. The kernel scheduler only knows when data is available to a process while the Erlang VM is able to schedule based on message queue size. So, the Erlang VM scheduling is able to intelligently schedule CloudI services more so than the kernel scheduling. However, the problem is unavoidable with current OSes and can be minimized by having a sufficiently large number of non-Erlang CloudI service threads and/or processes created to handle the throughput. The mismatch between the kernel scheduler and the Erlang VM scheduler is minimized by CloudI's management of CloudI requests, since an external service thread is only provided a single request at a time (and the mismatch is required to provide fault-tolerance by isolating the memory used by external services from the Erlang VM memory).

When the number of requests sent to a service name exceeds the number of service processes, the services will begin to queue new requests while handling older requests (roughly, the distribution of requests to processes is random, so it may queue slightly early). A priority parameter can be used if there is differing importance for various service requests (priority is normally used when there is a data dependency that needs to be solved). The priority parameter is 0 by default, but -128 is the highest priority and 127 is the lowest priority, so that provides much room for representing asynchronous data dependencies (synchronous data dependencies would use a pipe pattern) or simply processing time priority.

Top

5.2 - Scalability Considerations

CloudI uses distributed Erlang for communicating between CloudI nodes (i.e., machines). Distributed Erlang creates a fully-connected network topology which makes the cluster size of CloudI nodes limited to about 50 to 100 nodes (not yet tested). The node count limitation could easily be surpassed by using ZeroMQ to bridge CloudI clusters. However, it was anticipated that with multi-core technology advancements, the need for very large CloudI clusters would be diminished in the immediate future. The databases that CloudI uses are much more likely to need large node counts to facilitate large amounts of data which can be accessed as key/value pairs or with Map/Reduce.

Top

5.3 - Stability and Fault-Tolerance Considerations

CloudI software release and versioning utilizes semantic versioning to make any upgrade considerations more explicit. Any other stability concerns are related to CloudI integration.

CloudI requests are not sent in a way that is meant to be persistent to simplify error-handling. Otherwise, fault-tolerant messaging would preserve requests that are irrelevant and/or erroneous at a future time. Instead, CloudI requests can cause a service to crash which means that the request is not handled by another service since it is unclear whether the request is erroneous or the service is buggy. CloudI requests also have a certain lifetime defined by the request timeout, so that the relevance of the request data is limited by the timeout. The request timeout acts to conserve processing time for the most relevant data and the services that require the data. If data needs to be fault-tolerant, the data should be stored within a database.

Error-handling should always be local (i.e., internal to the service) where the errors are most relevant. Any invalid or corrupt service data can terminate the service and will trigger a restart of the service based on its configuration parameters. A service should never be allowed to function in a zombie-state since this would only complicate performance, testing, debugging and development.

The service must exit whenever an unrecoverable error occurs. If a CloudI request causes an exception, the request will fail but the service will not be restarted. So, services should always have proper exception handling, to make sure the context of any errors is explicit (otherwise, the service source code will become difficult to maintain if any CloudI requests fail). Without service exception handling, the exception will cause exception information to be logged, however, the information may be minimal (this depends on the limits of the programming language used).

The non-Erlang CloudI services receive their own Operating System (OS) process, so they are well isolated from the Erlang VM's memory. However, Erlang CloudI services could be written with malevolent intentions which would make CloudI unstable or erroneous. This means that Erlang CloudI service code must have a greater amount of implicit trust that the programmer is not trying to cause problems. With non-Erlang CloudI services there isn't as much concern about whether there are problems within the software, since the errors receive isolation within the CloudI framework.

Top

5.4 - Integration Considerations

The stdout and stderr of any non-Erlang CloudI service is captured and sent separately to be logged by CloudI with the associated Operating System (OS) process id. The CloudI API makes sure that both the stdout and the stderr streams are unbuffered within an external CloudI service, so the output will be logged as quickly as possible within the CloudI log as error data (for stderr data) or as info data (for stdout data).

Any Erlang CloudI services can utilize CloudI's logging (ideally for

information related to service problems or failures) for asynchronous logging (logging that does not carry a performance penalty).

Top

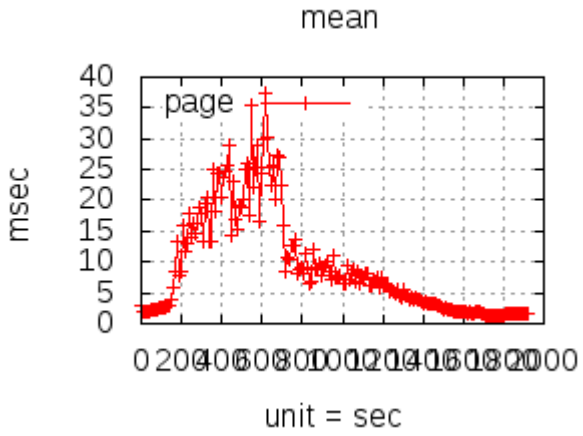
5.5 - Load Testing

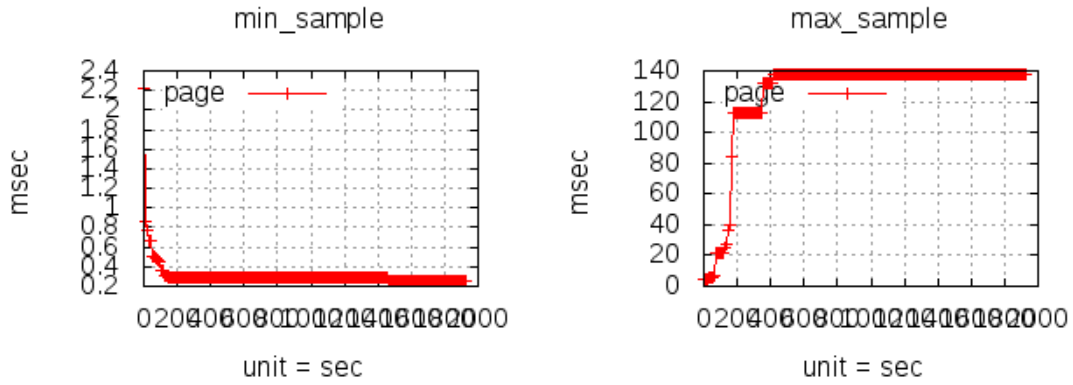
As part of the 1.0.0 CloudI release, the http_req test was used to do a load test of CloudI incoming HTTP requests. The loadtest shows how the results vary when interacting with separate programming languages that CloudI supports with the CloudI API. 20 thousand concurrent connections were used with a dynamic XML request/response protocol sent at a frequency of 10 thousand requests per second. All external services (i.e., non-Erlang services) were only given a single OS process with no thread usage. All programming languages received the same amount of load, so it helps provide a comparison for performance within the CloudI framework. To find out more, look at the raw results in src/tests/http_req/loadtest/results_v1_0_0/201206_20k_10kreqs/ or the configuration in src/tests/http_req/loadtest/results_v1_0_0/201206_20k_10kreqs/setup/.

For the 1.1.0 CloudI release, a comparison was done of misultin and cowboy (two Erlang HTTP servers), with 20k and 40k concurrent connections at 10k req/s. The summary with the 1.0.0 results is available here and the raw results exist in src/tests/http_req/loadtest/results_v1_1_0/.

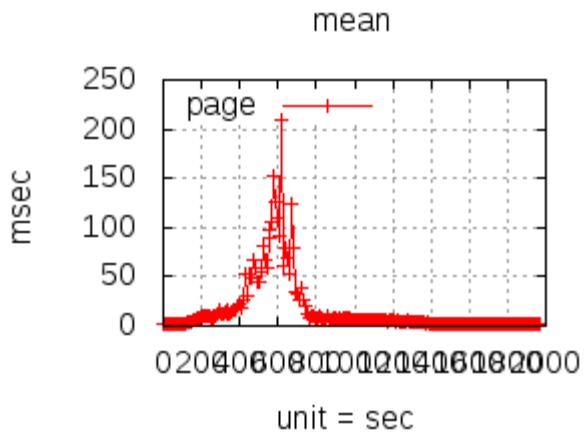
The latency graphs below show service request performance during the 1.0.0 loadtest of 20k at 10k req/s (with misultin and R15B01):

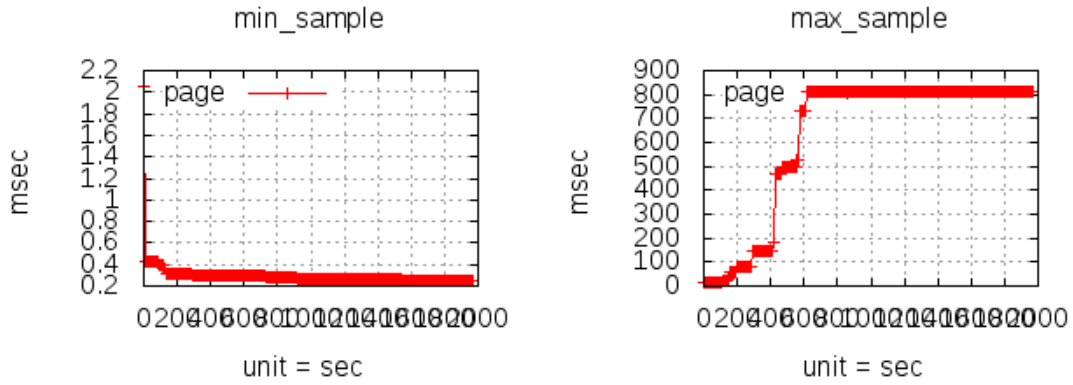
- C++/C



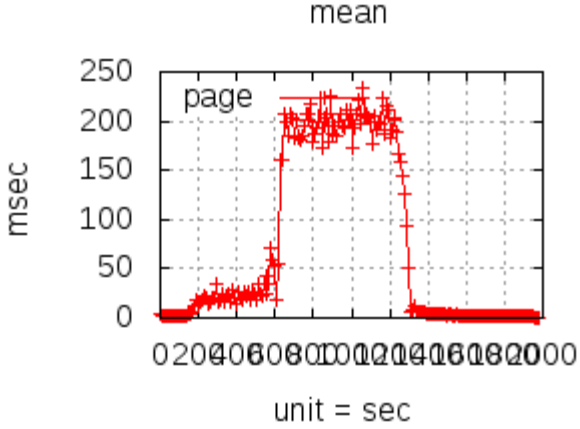


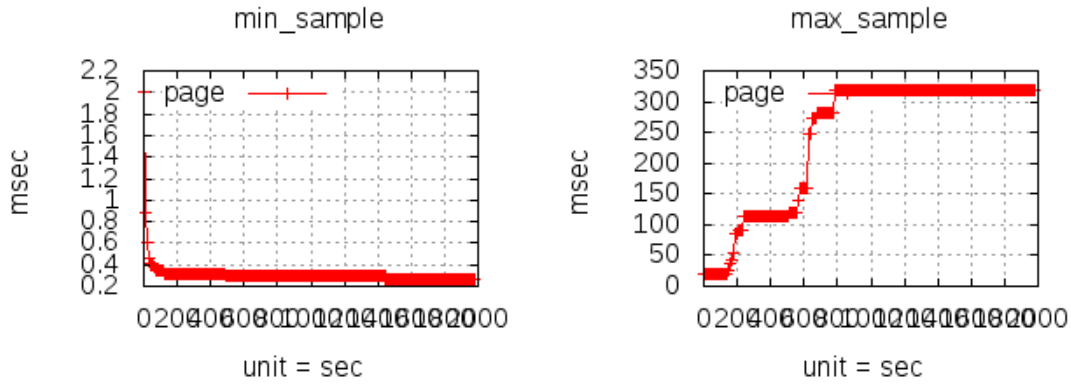
- Erlang



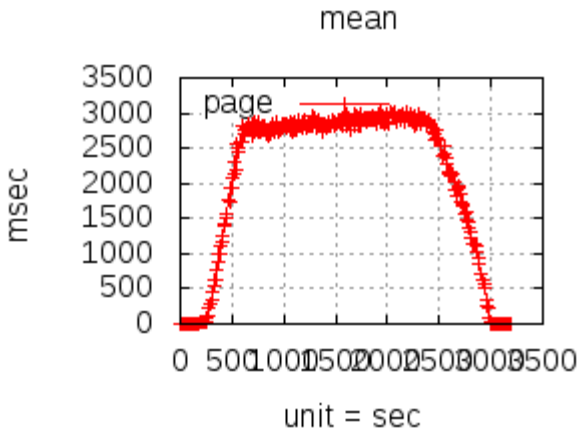


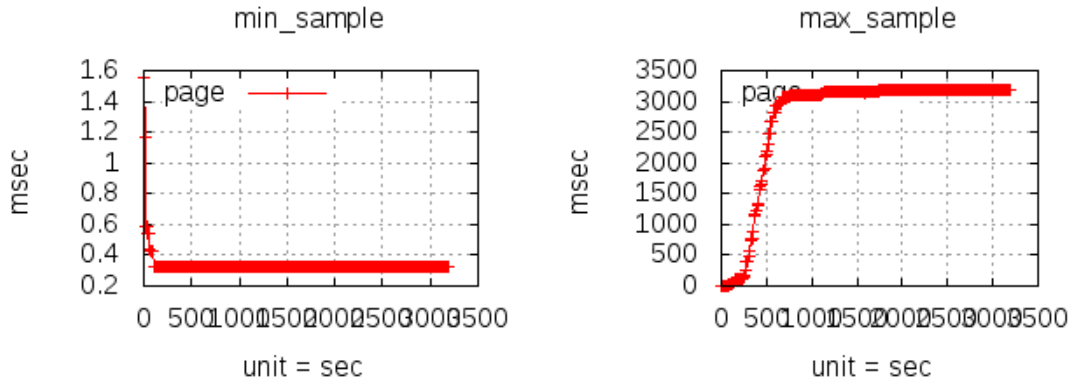
- Java



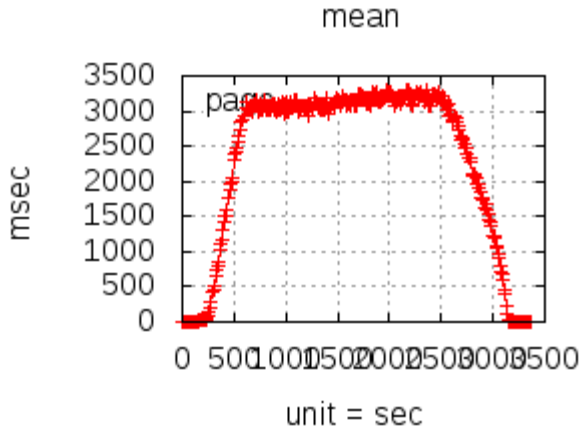


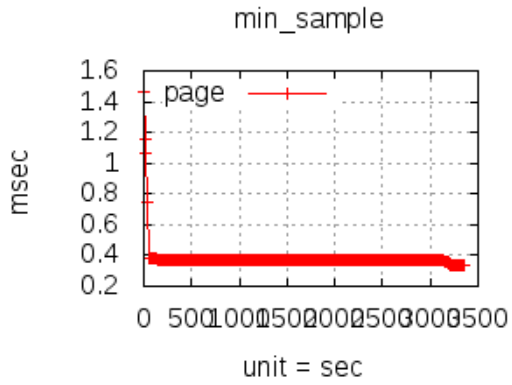
- Python

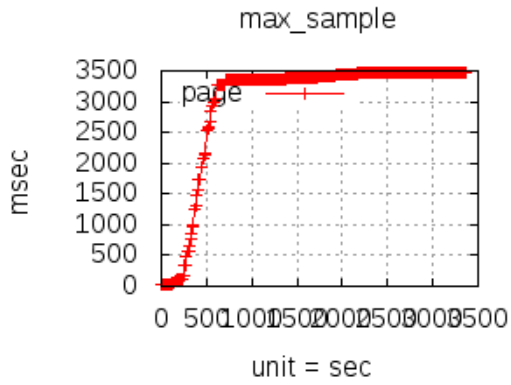




- Ruby







Top

6 - Services

6.1 - C++/C Service Implementation

There are separate header files that provide both a C CloudI API (cloudi.h) and a C++ CloudI API (cloudi.hpp) which are mutually exclusive. The header files do not bring in external dependencies but both require the standard C++ library as a link-time dependency. Some of the integration tests that provide example usage of the C++/C CloudI API are:

- src/tests/hexpi (C++ example with threads) Hexadecimal PI Test
- src/tests/http_req (C example without threads) HTTP Request Test

For more information, please refer to "4.1 - How do I integrate external software with CloudI?".

Top

6.2 - Erlang Service Implementation

Erlang CloudI services use the cloudi_service behavior to create an "internal" service (all non-Erlang CloudI services are "external"). The cloudi_service behavior requires that the service implement the following functions:

- cloudi_service_init/3
- cloudi_service_handle_request/11
- cloudi_service_handle_info/3
- cloudi_service_terminate/2

Many examples of Erlang CloudI services exist within the CloudI source code because the Erlang CloudI services provide integration with external systems like the supported databases (CouchDB, PostgreSQL, etc.), the supported messaging (HTTP, ZeroMQ, etc.), and the Service API functionality. Some example usage of the Erlang CloudI API includes:

- cloudi_service_filesystem Service for Caching Static File Data
- cloudi_service_work_manager Service For Caching Database Updates
- cloudi_service_timers service for Sending Service Messages On Timers
- Hexadecimal PI Test Load Balancer

For more information, please refer to "4.1 - How do I integrate external software with CloudI?".

Top

6.3 - Java Service Implementation

The Java CloudI API uses synchronous IO on file descriptors for an efficient light-weight interface. Some of the integration tests that provide example usage of the Java CloudI API include:

- src/tests/http HTTP Test

For more information, please refer to "4.1 - How do I integrate external software with CloudI?".

Top

6.4 - Python Service Implementation

The Python CloudI API provides a simple interface for making Python CloudI services. Some of the integration tests that provide example usage of the Python CloudI API include:

- src/tests/http HTTP Test
- src/tests/zeromq ZeroMQ Test

An example configuration (from the default CloudI configuration) is provided below:

```
{external,
    "/tests/http/",
    "@PYTHON@",
    "tests/http/service/service.py 4 tcp 16384",
    [],
    none, tcp, 16384,
    5000, 5000, 5000, [api], undefined, 1, 4, 5, 300, []}
```

There are two implementations of the Python CloudI API: a pure-Python CloudI

API (module "cloudi") and a Python/C CloudI API (module "cloudi_c"). Just specify the Python library by the module imported, since the same interface is provided within both choices. The "cloudi_c" module has been shown to provide a speedup greater than 400% when compared with the "cloudi" module, with both under a load of 10,000 requests/second. For more information, please refer to "4.1 - How do I integrate external software with CloudI?".

Top

6.5 - Ruby Service Implementation

The Ruby CloudI API provides a simple interface for making Ruby CloudI services. Some of the integration tests that provide example usage of the Ruby CloudI API include:

- src/tests/http HTTP Test

For more information, please refer to "4.1 - How do I integrate external software with CloudI?".

Top

6.6 - HTTP Integration

HTTP integration with CloudI services uses service names that have a prefix that matches the Uniform Resource Locator (URL) path. A simple example caches static filesystem files recursively so that the file path is the service name suffix (with the "/get" HTTP method suffix at the end, e.g., "index.html/get"). The example can be found in the default CloudI configuration usage of the cloudi_service_filesystem which is shown below:

```
{internal,
    "/tests/http_req/",
    cloudi_service_filesystem,
    [{directory, "tests/http_req/public_html/"}],
    none,
    5000, 5000, 5000, [api], undefined, 1, 5, 300, []}
```

When CloudI is running with this service configuration, the files in the path tests/http_req/public_html/ are browsable at http://127.0.0.1:6464/tests/http_req/.

The incoming HTTP traffic goes through the cloudi_service_http Erlang CloudI service and simply uses the URL path to send a request to the subscribing CloudI service, where the prefix of the service name was set in the service configuration but the suffix of the service name was declared programmatically by calling the CloudI API subscribe function.

Quicker access to static files can be provided by nginx or other simple

HTTP servers, so this is just an internal service example of CloudI HTTP integration (CloudI is normally for dynamic requests that require both scalability and fault-tolerance).

Other simple HTTP integration examples can be found among the integration tests:

- src/tests/http HTTP Test (with curl file-based test requests and Python/Ruby/Java services)
- src/tests/http_req HTTP Request Test (with a C service)

To prevent HTTP requests from going to internal services, Access Control List (ACL) entries can be added that prevent the cloudi_service_http Erlang CloudI service from sending to the internal services. The ACL entries would be service name patterns that include the internal services in a list that is referenced directly (i.e., literally as a list of string) or indirectly by an atom that represents the list of strings. The ACL entries would be specified for the cloudi_service_http service configuration's deny list. If service names are named consistently so that the service name represents a path which is a destination in a tree or hierarchy, then there should be no problems when adding or removing services dynamically (since the ACL entries will remain valid for the consistent service name pattern usage).

For more information, please refer to "4.1 - How do I integrate external software with CloudI?".

Top

6.7 - ZeroMQ Integration

ZeroMQ integration is provided by the cloudi_service_zeromq Erlang CloudI service. The CloudI configuration uses the cloudi_service_zeromq service to create service names that represent ZeroMQ messaging endpoints. There are three ZeroMQ configuration examples in the default CloudI configuration which are (partially) shown below:

```
% Zig-Zag test
{internal,
    "/tests/zeromq/",
    % inbound/outbound message paths much be acyclic
    % (if they are not, you will receive a erlzmq EFSM error
    %  because the ZeroMQ REQ has received 2 zmq_send calls)
    cloudi_service_zeromq,
    % outbound ZeroMQ requests connect a CloudI name to a ZeroMQ endpoint
    [{outbound, {"zigzag_start", ["ipc:///tmp/cloudizigzagstart"]}},
    % inbound ZeroMQ replies connect a ZeroMQ endpoint to a CloudI name
     {inbound, {"zigzag_step1", ["ipc:///tmp/cloudizigzagstart"]}},
     {outbound, {"zigzag_step1", ["inproc://zigzagstep1"]}},
     {inbound, {"zigzag_step2", ["inproc://zigzagstep1"]}},
    % ZeroMQ publish connects a CloudI name to a ZeroMQ (subscribe) name
    % as {CloudI name (suffix), ZeroMQ name for message prefix}
    % for any number of endpoints
```

```
     {publish, {[{"zigzag_step2", "/zeromq/step2"}],
                ["inproc://zigzagstep2a",
                 "ipc:///tmp/cloudizigzagstep2b",
                 "inproc://zigzagstep2c",
                 "ipc:///tmp/cloudizigzagstep2d"]}},
    % ZeroMQ subscribe connects a CloudI name to a ZeroMQ (subscribe) name
    % as {CloudI name (suffix), ZeroMQ name for subscribe setsocketopt}
    % for any number of endpoints
     {subscribe, {[{"zigzag_step3a", "/zeromq/step2"},
                   {"zigzag_step3b", "/zeromq/step2"}],
                  ["inproc://zigzagstep2a",
                   "ipc:///tmp/cloudizigzagstep2b",
                   "inproc://zigzagstep2c",
                   "ipc:///tmp/cloudizigzagstep2d"]}},
     {outbound, {"zigzag_step3a", ["inproc://zigzagstep3"]}},
     {inbound, {"zigzag_finish", ["inproc://zigzagstep3"]}}],
    immediate_closest,
    5000, 5000, 5000, [api], undefined, 2, 5, 300, []},
% Chain inproc test (50 endpoints in a sequential call path)
{internal,
    "/tests/zeromq/",
    cloudi_service_zeromq,
    [{outbound, {"chain_inproc_start", ["inproc://chainstep1"]}},
     {inbound, {"chain_inproc_step1", ["inproc://chainstep1"]}},
     {outbound, {"chain_inproc_step1", ["inproc://chainstep2"]}},
     {inbound, {"chain_inproc_step2", ["inproc://chainstep2"]}},
...
     {outbound, {"chain_inproc_step48", ["inproc://chainstep49"]}},
     {inbound, {"chain_inproc_step49", ["inproc://chainstep49"]}},
     {outbound, {"chain_inproc_step49", ["inproc://chainstep50"]}},
     {inbound, {"chain_inproc_finish", ["inproc://chainstep50"]}}],
    immediate_closest,
    5000, 5000, 5000, [api], undefined, 2, 5, 300, []},
% Chain ipc test (25 endpoints in a sequential call path)
{internal,
    "/tests/zeromq/",
    cloudi_service_zeromq,
    [{outbound, {"chain_ipc_start", ["ipc:///tmp/cloudichainstep1"]}},
     {inbound, {"chain_ipc_step1", ["ipc:///tmp/cloudichainstep1"]}},
     {outbound, {"chain_ipc_step1", ["ipc:///tmp/cloudichainstep2"]}},
     {inbound, {"chain_ipc_step2", ["ipc:///tmp/cloudichainstep2"]}},
...
     {outbound, {"chain_ipc_step23", ["ipc:///tmp/cloudichainstep24"]}},
     {inbound, {"chain_ipc_step24", ["ipc:///tmp/cloudichainstep24"]}},
     {outbound, {"chain_ipc_step24", ["ipc:///tmp/cloudichainstep25"]}},
     {inbound, {"chain_ipc_finish", ["ipc:///tmp/cloudichainstep25"]}}],
    immediate_closest,
    5000, 5000, 5000, [api], undefined, 2, 5, 300, []}
```

The three cloudi_service_zeromq Erlang CloudI services are used by the ZeroMQ integration test to test the ZeroMQ messaging when the integration test service starts. ZeroMQ configuration within CloudI is dynamic through usage of the Service API. For more information, please refer to "4.1 - How do I integrate external software with CloudI?".

Top

7 - Databases

7.1 - CouchDB Integration

The cloudi_service_db_couchdb internal service accepts requests from other CloudI services. The service expects database commands supplied as Erlang tuples or atoms. When the service receives data from an external service the data received is binary and should be a string that contains Erlang terms that is the database command. The command result is returned as binary to an external service. An internal service can send the command as Erlang terms and will receive Erlang terms for the result. The service name used to communicate with the database is the configured database service name prefix with the database name appended (i.e., "/db/couchdb /cloudi_tests" in the example below).

An example configuration for a single database that is represented as a single service is below:

```
    {internal,
        "/db/couchdb/",
        cloudi_service_db_couchdb,
        [{database, "cloudi_tests"},
         {timeout, 20000}, % ms
         {hostname, "127.0.0.1"},
         {port, 5984}],
        none,
        5000, 5000, 5000, undefined, undefined, 1, 5, 300, []}
```

Top

7.2 - memcached Integration

The cloudi_service_db_memcached internal service accepts requests from other CloudI services. The service expects database commands supplied as Erlang tuples or atoms. When the service receives data from an external service the data received is binary and should be a string that contains Erlang terms that is the database command. The command result is returned as binary to an external service. An internal service can send the command as Erlang terms and will receive Erlang terms for the result. The service name used to communicate with the database is the configured database service name prefix with the database name appended (i.e., "/db/memcached /cloudi_tests" in the example below).

An example configuration for a single database that is represented as a single service is below:

```
    {internal,
        "/db/memcached/",
```

```
        cloudi_service_db_memcached,
        [{database, "cloudi_tests",
          [{"127.0.0.1", 11211, 1}]}],
        none,
        5000, 5000, 5000, undefined, undefined, 1, 5, 300, []}
```

The list of host-port-connection_count tuples is used for providing continuum hashing of database keys. Using continuum hashing avoids rehashing all the keys (i.e., cached-misses) when a memcached node fails.

Top

7.3 - PostgreSQL Integration

The cloudi_service_db_pgsql internal service accepts requests from other CloudI services. The service expects SQL input and provides the query result either as an Erlang tuple or as binary encoded data based on whether the input was binary or a list. All data coming from external services is received as binary and is returned as binary that can be used to determine the result of a query. Internal services are able to send SQL as an Erlang list and will then receive a tuple from the database driver that is the result of the query. The service name used to communicate with the database is the configured database service name prefix with the database name appended (i.e., "/db/pgsql/cloudi_tests" in the example below).

An example configuration for a single database that is represented as a single service is below:

```
    {internal,
        "/db/pgsql/",
        cloudi_service_db_pgsql,
        [{database, "cloudi_tests"},
         {timeout, 20000}, % ms
         {hostname, "127.0.0.1"},
         {username, "cloudi"},
         {password, "XXXXXXXXX"},
         {port, 5432}],
        none,
        5000, 5000, 5000, undefined, undefined, 1, 5, 300, []}
```

Top

7.4 - MySQL Integration

The cloudi_service_db_mysql internal service accepts requests from other CloudI services. The service expects SQL input and provides the query result either as an Erlang tuple or as binary encoded data based on whether the input was binary or a list. All data coming from external services is received as binary and is returned as binary that can be used to determine the result of a query. Internal services are able to send SQL as an Erlang list and will then receive a tuple from the database driver that is the result of the query. The service name used to communicate with the database is the

configured database service name prefix with the database name appended (i.e., "/db/mysql/cloudi_tests" in the example below).

An example configuration for a single database that is represented as a single service is below:

```
    {internal,
        "/db/mysql/",
        cloudi_service_db_mysql,
        [{database, "cloudi_tests"},
         {timeout, 20000}, % ms
         {encoding, utf8},
         {hostname, "127.0.0.1"},
         {username, "cloudi"},
         {password, "XXXXXXXXX"},
         {port, 3306}],
        none,
        5000, 5000, 5000, undefined, undefined, 1, 5, 300, []}
```

Top

7.5 - Tokyo Tyrant Integration

The cloudi_service_db_tokyotyrant internal service accepts requests from other CloudI services. The service expects database commands supplied as Erlang tuples or atoms. When the service receives data from an external service the data received is binary and should be a string that contains Erlang terms that is the database command. The command result is returned as binary to an external service. An internal service can send the command as Erlang terms and will receive Erlang terms for the result. The service name used to communicate with the database is the configured database service name prefix with the database name appended (i.e., "/db/tokyotyrant /cloudi_tests" in the example below).

An example configuration for a single database that is represented as a single service is below:

```
    {internal,
        "/db/tokyotyrant/",
        cloudi_service_db_tokyotyrant,
        [{database, "cloudi_tests"},
         {timeout, 20000}, % ms
         {hostname, "127.0.0.1"},
         {port, 1978}],
        none,
        5000, 5000, 5000, undefined, undefined, 1, 5, 300, []}
```

Top

7.6 - Other Database Integration

Other databases can easily be integrated with CloudI. The best database integration uses a database driver implemented completely in Erlang and uses

a cloudi_service_db_name module to implement CloudI service integration with the cloudi_service behavior. By using a database driver written in Erlang the source code is naturally more scalable and fault-tolerant. If the database driver used an Erlang NIF or an Erlang port driver instead, the driver would not be isolated from the Erlang VM (though the implementation might be more efficient). The database driver would typically communicate with the database by using a socket with TCP.

Database integration can be done in other complex ways if required, but the integration approach previously mentioned is a typical approach used within the CloudI framework.

Top