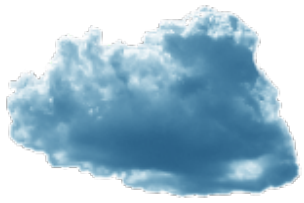
CloudI 

Frequently Asked Questions

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 - 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.

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 internal messaging. CloudI messaging enforces realtime constraints using timeouts, so that message 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 receives the isolation of Operating Systems processes and is called an external service. External services can utilize the CloudI API with any threading library to achieve greater scalability and reduce internal latency. CloudI API usage in Erlang creates an internal service which utilizes light-weight Erlang processes. Examples of using the CloudI API are provided as integration tests or internal services.

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

Top

1.3 - 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

1.4 - Is Commercial support available for CloudI?

- Integration Development
- Operations Maintenance
- Software License Agreements

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

Top

1.5 - 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 list_pg.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

1.6 - Who develops CloudI?

Michael Truog

Top

1.7 - 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 "1.2 - What is CloudI?".

Top

1.8 - 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 "1.2 - What is CloudI?".

Top

1.9 - Does CloudI support REST?

Yes! CloudI is a system that enforces RESTful development practices. For more details please refer to "1.2 - What is CloudI?".

Top

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

Yes! CloudI can be used as a Platform as a Service (PaaS) and is the first PaaS open source project (not CloudFoundry). However, 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 "1.2 - What is CloudI?".

Top

1.11 - 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:	http://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/master
Releases:	http://sourceforge.net/api/file/index/project-id/281423/mtime/desc/limit/20/rss
Private Cloud Computing News:	http://www.google.com/reader/shared/cloudi.news

Top

2.5 - Presentations

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

2.6 - 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. 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. make install
```

Currently, it is best to install into a local directory with a command like:

```
./configure --prefix=`pwd`/../cloudi_install
```

More work will be occurring on the build system (autoconf/automake/rebar) to make sure the deployment is more typical for UNIX systems (i.e., all configuration in "etc" and all logs in "/var/logs") but this work is not yet complete.

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, Ruby (≥ 1.9), and Erlang (≥ R14B01). It is likely that Erlang will need to be compiled from source, since R14B01 (and preferably R14B02) are new enough that they have not yet been added to package managers. Dependencies as they are packaged for different operating systems are listed below:

Operating System	Packages
Ubuntu (apt-get install <package(s)>)	• g++ • libboost-thread-dev • libboost-dev • default-jdk • python • ruby1.9 • libgmp3-dev

Top

3.2 - Installation Options

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

--prefix="/path/to/install/"	Specify an Installation Path
--with-zeromq	Include ZeroMQ support

For more installation configuration option details, please refer to

src/INSTALL.

Top

3.3 - OS X Installation

To install CloudI dependencies on OSX you either need macports or homebrew. With macports, CloudI configuration can be executed with:

```
CXXFLAGS="-I/opt/local/include" LDFLAGS="-L/opt/local/lib" ./configure
```

Top

3.4 - Running CloudI

If CloudI is installed in PREFIX (set with the configure script --prefix= command line argument), then CloudI can be started with:

```
PREFIX/bin/cloudi start
```

To stop the running CloudI node, execute:

```
PREFIX/bin/cloudi stop
```

When CloudI is running, CloudI logging output will be appended to PREFIX/logs/cloudi.log (the installation will change in the future to put the log into /var/log/cloudi.log if CloudI is installed with sufficient permissions).

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 messages 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 Job 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 messages to "/cloudi/api/json_rpc/", which is called a "name".

The messages are load balanced across all the services that have subscribed to the same name during the lookup to find the message destination. There is a service configuration parameter called the "destination refresh" that determines how the internal CloudI load balancing occurs when a message 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 messages (i.e., only receives messages and returns a result) and creates an error that kills the service if the service does send a message. 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 message:

- 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 message will be retried until the message timeout elapses and no binary data will be returned (i.e., returning no data is equivalent to a timeout). 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 message 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 message(s) so that the message response may be queried with the "recv_async" function. 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 message 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 message 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 message, 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 messages use the same callback function which is cloudi_job_handle_request/8. 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 message timeout elapsed.

The Access Control List (ACL) is simply a list of strings that define prefixes that must be explicitly allowed or denied when determining if a

service can send to the service name. If a prefix is both allowed and denied, the prefix is denied (deny takes precedence). When defining ACLs, it is possible to use Erlang atoms to represent lists of string prefixes so that logical groupings are created. The ACL atoms are then able to be specified anywhere a normal ACL string might be present. So, it is best to group ACL string prefixes based on context to simplify the configuration specification.

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_job_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_job_zeromq configuration (in the cloudi.conf file or through the Job API jobs_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_job_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_job_misultin accepts HTTP traffic and makes the HTTP requests CloudI requests where the HTTP path in the URL is used as the service name. So, when an HTTP request is received the corresponding service name will be called with the request contents (uncompressed, if the request was compressed). The headers are not passed with the request. However, cloudi_job_misultin configuration can change how the HTTP headers are interpreted. 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_job_db_pgsql, cloudi_job_db_mysql, etc.) uses the database name as the service name suffix. Services can send messages 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 Job API. The Job API can be used by any CloudI services. However, typical usage of the Job API would use raw HTTP requests or JSON-RPC over HTTP. An example of using the Job API through JSON-RPC over

HTTP can be found in src/tests/job_api/run.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 by using the procedure name as a service name suffix and putting the RPC parameters into the request body. The request body is simply uninterpreted binary data, so no format is imposed on the user of the CloudI API. 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_job_misultin service. The request body is either the body of the uncompressed POST request or the GET query string. The headers are not passed to the CloudI service because all HTTP header interpretation is handled within the cloudi_job_misultin service. 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 messages from being sent to service name prefixes. 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 prefixes. 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 provide

display how this works:

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

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

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

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

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

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

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

The Job API supports dynamically starting services by supplying a 'jobs' list in the same format as the configuration file. The Job 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 messages 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 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 Job 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

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.

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 Considerations

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.

Top

6 - Services

6.1 - C/C++ Service Implementation

The C CloudI API provides an interface for both C or C++. The integration tests that provide example usage of the 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_job behavior to create an "internal" service (all non-Erlang CloudI services are "external"). The cloudi_job behavior requires that the service implement the following functions:

- cloudi_job_init/3
- cloudi_job_handle_request/8
- cloudi_job_handle_info/3
- cloudi_job_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 Job API functionality. Some example usage of the Erlang CloudI API includes:

- cloudi_job_filesystem Service for Caching Static File Data
- cloudi_job_work_manager Service For Caching Database Updates
- cloudi_job_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 simply uses blocking IO on file descriptors for an efficient light-weight interface. The integration tests that provide example usage of the Java CloudI API are:

- 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. The integration tests that provide example usage of the Python CloudI API are:

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

To keep the stdout stream of the Python service flushed to the CloudI log, add {"PYTHONUNBUFFERED", "true"} to the configuration's list of environment variables. An example configuration (from the default CloudI configuration) is provided below:

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

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. The integration tests that provide example usage of the Ruby CloudI API are:

- 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. The example can be found in the default CloudI configuration usage of the cloudi_job_filesystem which is shown below:

```
{internal,
    "/tests/http_req/",
    cloudi_job_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_job_misultin Erlang CloudI service and simply uses the URL path to send a message 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.

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_job_misultin Erlang CloudI service from sending to the internal services. The ACL entries

would be service name prefixes 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_job_misultin 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 prefix 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_job_zeromq Erlang CloudI service. The CloudI configuration uses the cloudi_job_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_job_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_job_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_job_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_job_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 Job 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_job_db_couchdb internal service accepts messages 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_job_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_job_db_memcached internal service accepts messages 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_job_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_job_db_pgsql internal service accepts messages 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_job_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_job_db_mysql internal service accepts messages 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_job_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_job_db_tokyotyrant internal service accepts messages 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_job_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_job_db_name module to implement CloudI service integration with the cloudi_job 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 previously mentioned is a typical approach used within CloudI.

Top