CLyman Documentation

Release 1.0.0

AO

Apr 16, 2018

Contents:

Go Home

CHAPTER 1

Docker

The easiest way to get started with CLyman is with Docker

If you do not have Docker installed, please visit the link above to get setup before continuing.

The first thing we need to do is setup the Docker Network that will allow us to communicate between our containers:

```
docker network create dvs
```

Before we can start CLyman, we need to have a few other programs running first. Luckily, these can all be setup with Docker as well:

```
docker run -d --name=registry --network=dvs consul
```

```
docker run -d --network=dvs --name=document-db mongo
```

This will start up a single instance each of Mongo and Consul. Consul stores our configuration values, so we'll need to set those up. You can either view the Consul Documentation for information on starting the container with a Web UI, or you can use the commands below for a quick-and-dirty setup:

```
docker exec -t registry curl -X PUT -d 'mongodb://document-db:27017/' http://
localhost:8500/v1/kv/clyman/Mongo_ConnectionString
```

```
docker exec -t registry curl -X PUT -d 'mydb' http://localhost:8500/v1/kv/
clyman/Mongo_DbName
```

```
docker exec -t registry curl -X PUT -d 'test' http://localhost:8500/v1/kv/
clyman/Mongo_DbCollection
```

```
docker exec -t registry curl -X PUT -d 'True' http://localhost:8500/v1/kv/
clyman/StampTransactionId
```

```
docker exec -t registry curl -X PUT -d 'Json' http://localhost:8500/v1/kv/
clyman/Data_Format_Type
```

Then, we can start up CLyman:

```
docker run --name clyman --network=dvs -p 5555:5555 -d aostreetart/clyman
-consul-addr=registry:8500 -ip=localhost -port=5555 -log-conf=CLyman/log4cpp.
properties
```

This will start an instance of CLyman with the following properties:

- Connected to network 'dvs', which lets us refer to the other containers in the network by name when connecting.
- Listening on localhost port 5555
- Connected to Consul Container

We can open up a terminal within the container by:

```
docker exec -i -t clyman /bin/bash
```

The 'stop_clyman.py' script is provided as an easy way to stop CLyman running as a service. This can be executed with:

```
python stop_clyman.py hostname port
```

For a more detailed discussion on the deployment of CLyman, please see the *Deployment Section* of the documentation.

CHAPTER 2

Using the Latest Release

In order to use the latest release, you will still need to start up the applications used by CLyman, namely Mongo and Consul. This can be done using the docker instructions above, or by installing each to the system manually. Instructions: * Mongo * Consul

Then, download the latest release from the Releases Page

Currently, pre-built binaries are available for:

- Ubuntu 16.04
- CentOS7

Unzip/untar the release file and enter into the directory. Then, we will use the easy_install.sh script to install CLyman. Running the below will attempt to install the dependencies, and then install the CLyman executable:

```
sudo ./easy_install.sh -d
```

If you'd rather not automatically install dependencies, and only install the executable, then you can simply leave off the '-d' flag. Additionally, you may supply a '-r' flag to uninstall CLyman:

```
sudo ./easy_install -r
```

Once the script is finished installing CLyman, you can start CLyman with:

```
sudo systemctl start clyman.service
```

The 'stop_clyman.py' script is provided as an easy way to stop CLyman running as a service. This can be executed with:

```
python stop_clyman.py hostname port
```

Note: The CLyman configuration files can be found at /etc/clyman, and the log files can be found at /var/log/clyman.

CHAPTER 3

Building from Source

The recommended system for development of CLyman is either Ubuntu 16.04 or CentOS7. You will need gcc 5.0 or greater installed to successfully compile the program.

```
git clone https://github.com/AO-StreetArt/CLyman.git
```

```
mkdir clyman_deps
```

```
cp CLyman/scripts/linux/deb/build_deps.sh clyman_deps/build_deps.sh
```

```
cd clyman_deps
```

```
sudo ./build_deps.sh
```

```
cd ../CLyman
```

```
make
```

This will result in creation of the clyman executable, which we can run with the below command:

```
./clyman
```

When not supplied with any command line parameters, CLyman will look for an app.properties file and log4cpp.properties file to start from.

You may also build the test modules with:

```
make tests
```

In order to run CLyman from a properties file, you will need:

- You need to have a Mongo Server installed locally. Instructions can be found at https://docs.mongodb.com/getting-started/
- You will also need a Kafka server running locally, instructions can be found at https://kafka.apache.org/quickstart

Continue on to the *Configuration Section* for more details on the configuration options available when starting CLyman.

CHAPTER 4

Configuration

4.1 Properties File

CLyman can be configured via a properties file, which has a few command line options:

- `./clyman` - This will start CLyman with the default properties file, app.properties
- `./clyman -config-file=file.properties` - This will start CLyman with the properties file, file.properties. Can be combined with -log-conf.
- `./clyman -log-conf=logging.properties` - This will start CLyman with the logging properties file, logging.properties. Can be combined with -config-file.

The properties file can be edited in any text editor.

4.2 Consul

Clyman can also be configured via a Consul Connection, in which we must specify the address of the consul agent, and the ip & port of the Inbound ZeroMQ Connection.

- `./clyman -consul-addr=localhost:8500 -ip=localhost -port=5555` - Start Clyman, register as a service with consul, and configure based on configuration values in Consul, and bind to an internal 0MQ port on localhost
- `./clyman -consul-addr=localhost:8500 -ip=tcp://my.ip -port=5555 -log-conf=logging.properties` - Start Clyman, register as a service with consul, and configure based on configuration values in Consul. Bind to an external 0MQ port on tcp://my.ip, and configure from the logging configuration file, logging.properties.

We can also use both a properties file and a Consul connection, in which case the properties file is used to define the ip and port of the inbound ZeroMQ connection, while Consul is used for registration and all other configuration retrieval.

- `./clyman -consul-addr=localhost:8500 -config-file=file.properties`

When configuring from Consul the keys of the properties file are equal to the expected keys in Consul.

4.3 Logging

The Logging Configuration File can also be edited with a text file, and the documentation for this can be found at http://log4cpp.sourceforge.net/api/classlog4cpp_1_1PropertyConfigurator.html Note that logging configuration is not yet in Consul, and always exists in a properties file.

The logging configuration provided shows all of the logging modules utilized by CLyman during all phases of execution, and all of these should be configured with the same names (for example, log4cpp.category.main).

Clyman is built with many different logging modules, so that configuration values can change the log level for any given module, the log file of any given module, or shift any given module to a different appender or pattern entirely. These modules should always be present within configuration files, but can be configured to suit the particular deployment needs.

4.4 Startup

CLyman can be started with an option to wait for a specified number of seconds prior to looking for configuration values and opening up for requests. This is particularly useful when used with orchestration providers, in order to ensure that other components are properly started (in particular, in order to allow time for Consul to be populated with default configuration values).

- `./clyman -wait=5` - This will start CLyman with the default properties file, and wait 5 seconds before starting.

4.5 Configuration Key-Value Variables

Below you can find a summary of the options in the Properties File or Consul Key-Value Store:

4.5.1 DB

- Mongo_ConnectionString - The string used to connect to the Mongo instance (example: mongodb://localhost:27017/)
- Mongo_DbName - The Mongo Database to connect to within the cluster
- Mongo_DbCollection - The Mongo Collection to utilize for storing documents

4.5.2 0MQ

- 0MQ_InboundConnectionString - The connectivity string for the inbound 0MQ Port (example: tcp://*:5555)

4.5.3 Kafka Connection

- KafkaBrokerAddress - The address and port of the Kafka Broker to send Object Updates to

4.5.4 Behavior

- DataFormatType - Are we communicating via JSON or Protocol Buffers
- StampTransactionId - True or False, do we stamp Transaction IDs on messages that do not have them already
- AtomicTransactions - True or False, do we enforce atomic transactions across all instances of CLyman for any given object. This guarantees that updates will be processed in the order they are received across the entire CLyman network.

Go Home

CHAPTER 5

Deployment

Note: At this time, CLyman has no built-in security or encryption mechanisms. Until such time, it is not recommended to deploy CLyman in Production.

The easiest methodology of deployment for CLyman is using Docker. At this time, it has not been tested with either Docker Compose or Docker Swarm.

This page will be updated after larger scale testing has been performed with CLyman.

Go Home

CHAPTER 6

API Overview

The Clyman API utilizes either JSON or Protocol Buffers, based on what the server is configured to process. In either case, the field names and message structure remains the same. This document will focus on the JSON API, but with this knowledge and the DVS Interface Protocol Buffer files, the use of the Protocol Buffer API should be equally clear.

Response Messages follow the same format as inbound messages, with a few additional fields.

To start with, here is an example JSON message which will create a single object:


```
{
    “msg_type”: 0,
    “operation”: 10,
    “transaction_id”: “12352”,
    “num_records”: 1,
    “objects”: [
    {
    “key”: “ABCDEF131”,
    “name”: “Test Object8”,
    “type”: “Mesh”,
    “subtype”: “Cube”,
    “owner”: “123”,
    “scene”: “DEFGHI8”,
    “frame”: 1,
    “timestamp”: 123456789,
    “translation”: [0, 0, 0],
    “quaternion_rotation”: [0, 0, 0, 0],
```

"euler_rotation": [0, 0, 0],

"scale": [1, 1, 1],

"assets": ["Asset_4"]

}

]

}

Let's take a look at the individual fields.

6.1 Object List

The Object List is the highest level wrapper in the API. It only contains 6 keys, one of which is an array of objects.

- msg_type – 0 for create, 1 for update, 2 for retrieve, 3 for delete, 4 for query, 5 to acquire a Device Lock, and 6 to release a Device Lock. The message type applies to all objects in the objects array.
- operation - An optional field used during updates to specify the type of operation. 10 is append, and 11 is remove. Remove operations can be used to remove assets from an object.
- err_code - An integer error code for the response, full list of codes can be found in the appendix.
- err_msg - A string error message for the response, will not be present when no error was encountered.
- transaction_id – An ID to distinguish a transaction within a larger network of applications
- num_records – This lets us give Clyman a maximum number of values to return from a query
- objects – An array containing objects

6.2 Object

A single Object (or Obj3, from here on out), is represented by a single element of the array from the "objects" key of the object list.

- key – Obj3 Key value (UUID)
- name – Name of the Obj3
- type – A string type of the Obj3. Normal types are 'Mesh', 'Curve', etc. However, no limitations are placed on what types may be entered.
- subtype – Similar to type, meant to differentiate between different types and allow for use of basic primitives which can be communicated from device to device very quickly. For example, 'Cube', 'Sphere', etc.
- owner – Identifier for the owner of the Obj3
- scene – ID For the Scene containing the object
- frame – The frame of the Object transformations
- timestamp – The timestamp of the Object transformations
- translation – X, Y, and Z values for the translation of the object from it's origin
- euler_rotation – X, Y, and Z values for the local rotation of the object about the respective axis, in radians
- quaternion_rotation - W, X, Y, and Z values for the local quaternion rotation of the object, in radians

- scale – X, Y, and Z values for the scaling of the object
- assets – An Array of identifiers for "assets", which should be downloaded by users in order to view the object.

CHAPTER 7

Field Mapping

Field	Data Type	Create	Get	Update	Delete	Query	Lock	Unlock	Overwrite
msg_type	Integer	X	X	X	X	X	X	X	X
operation	Integer			*					
transaction_id	String	*	*	*	*	*	*	*	*
num_records	String					*			
key	String		X	*	X	*	*	*	*
name	String	X		*		*	*	*	*
type	String	*		*		*	*	*	
subtype	String	*		*		*	*	*	
owner	String	*		*		*	*	*	
scene	String	X		*		*	*	*	*
frame	Integer	*		*		*	*	*	*
timestamp	Integer	*		*		*	*	*	*
translation	Array - Double	X		*		*	*	*	*
euler_rotation	Array - Double	X		*		*	*	*	*
quaternion_rotation	Array - Double	X		*		*	*	*	*
scale	Array - Double	X		*		*	*	*	*
assets	Array - String	X		*			*	*	

X – Required

* - Optional

CHAPTER 8

Message Types

8.1 Object Create

Create a new Obj3. Returns a unique key for the object.

8.2 Object Retrieve

The object retrieve message will retrieve an object by key, and return the full object

8.3 Object Update

Object updates can be used to either update basic object attributes (name, type, etc), or to apply transformations to the object. Transformations will be applied in the order that they are received.

8.4 Object Destroy

Destroy an existing Obj3 by key. Basic success/failure response.

8.5 Object Query

This will query objects by attributes other than their keys

8.6 Device Lock Acquire

Subset of Object Update, uses 'owner' field as the key to acquire a lock on an object. This ensures that no other devices update the object, until the lock is released.

8.7 Device Lock Release

Subset of Object Update, uses 'owner' field as the key to release a lock on an object.

8.8 Object Overwrite

An Object Overwrite is designed to be a high-speed update, primarily used for live feeds. Rather than sending in transformations from the current state of the object, as in the Object Update message, here the Object sends the current total state of the object, rather than the difference. This completely overwrites the transform information of the object, and streams that information out to other devices via Apache Kafka.

This message can utilize either a Key or a Name-Scene combination to perform the overwrite.

CHAPTER 9

Appendix A: JSON Message Samples

9.1 Inbound

9.1.1 Object Create

{ “msg_type”: 0, “transaction_id”: “12354”, “num_records”: 1, “objects”: [

{ “key”: “ABCDEF133”, “name”: “Test Object10”, “type”: “Mesh”, “subtype”: “Cube”, “owner”: “123”, “scene”: “DEFGHI10”, “translation”: [0, 0, 0], “quaternion_rotation”: [0, 0, 0, 0], “euler_rotation”: [0, 0, 0], “scale”: [1, 1, 1], “assets”: [“Asset_5”]

}

]

}

9.1.2 Object Retrieve

{ “msg_type”: 2, “transaction_id”: “123464”, “num_records”: 256, “objects”: [

{ “key”: “59ab6e44ac48b7000148c86a”

}

]

}

9.1.3 Object Update

{ “msg_type”: 1, “operation”: 10, “transaction_id”: “123464”, “num_records”: 1, “objects”: [

{ “key”: “59ab6e44ac48b7000148c86a”, “name”: “Test Object 123464”, “type”: “Curve”, “subtype”: “Sphere”, “owner”: “456”, “scene”: “DEFGHIJ123464”, “translation”: [0, 0, 1], “quaternion_rotation”: [0, 0, 0, 0], “euler_rotation”: [0, 0, 0], “scale”: [1, 1, 2], “assets”: [“Asset_5”]

}

]

}

9.1.4 Object Overwrite

{ “msg_type”: 1, “operation”: 10, “transaction_id”: “123464”, “num_records”: 1, “objects”: [

{ “key”: “59ab6e44ac48b7000148c86a”, “name”: “Test Object 123464”, “scene”: “DEFGHIJ123464”, “translation”: [0, 0, 1], “quaternion_rotation”: [0, 0, 0, 0], “euler_rotation”: [3.14, 0, 0], “scale”: [1, 1, 2]

}

]

}

9.1.5 Object Destroy

{ “msg_type”: 3, “transaction_id”: “123463”, “num_records”: 1, “objects”: [

{ “key”: “59ab6e44ac48b7000148c869”

}

]

}

9.1.6 Object Query

{ “msg_type”: 4, “transaction_id”: “123463”, “num_records”: 1, “objects”: [

{ “name”: “Test Object 123463”

}, {

“name”: “Test Object 123464”

}

]

}

9.1.7 Object Lock

{ “msg_type”: 5, “transaction_id”: “123465”, “num_records”: 1, “objects”: [

{ “key”: “59ab6e44ac48b7000148c86b”, “name”: “Test Object 123465”, “type”: “Mesh”, “subtype”: “Cube”, “owner”: “10”, “scene”: “DEFGHIJ123465”, “translation”: [0, 0, 1], “quaternion_rotation”: [0, 0, 0, 0], “euler_rotation”: [0, 0, 0], “scale”: [1, 1, 2], “assets”: [“Asset_5”]

}

]

}

9.1.8 Object Unlock

{ “msg_type”: 6, “transaction_id”: “123465”, “num_records”: 1, “objects”: [

{ “key”: “59ab6e44ac48b7000148c86b”, “name”: “Test Object 123465”, “type”: “Mesh”, “subtype”: “Cube”, “owner”: “10”, “scene”: “DEFGHIJ123465”, “translation”: [0, 0, 1], “quaternion_rotation”: [0, 0, 0, 0], “euler_rotation”: [0, 0, 0], “scale”: [1, 1, 2], “assets”: [“Asset_5”]

}

]

}

9.2 Response

9.2.1 Object Create

{ “msg_type”:0, “err_code”:100, “num_records”:1, “objects”:[

{ “key”:”59ab6e44ac48b7000148c86b”, “transform”:[1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0]

}

]

}

9.2.2 Object Update

{ “msg_type”:1, “err_code”:100, “num_records”:1, “objects”:[

{ “key”:”59ab6e44ac48b7000148c86b”, “name”:”Test Object 123465”, “scene”:”DEFGHIJ123465”, “type”:”Mesh”, “subtype”:”Cube”, “owner”:”456”, “transform”:[1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,2.0,2.0,0.0,0.0,0.0,0.0,1.0], “assets”: [“Asset_5”]

}

]

}

9.2.3 Object Overwrite

{“msg_type”:7,”err_code”:100,”num_records”:0,”objects”:[]}

9.2.4 Object Retrieve

{ “msg_type”:2, “err_code”:100, “num_records”:1, “objects”:[

{ “key”:”59ab6e44ac48b7000148c869”, “name”:”Test Object8”, “scene”:”DEFGHI8”, “type”:”Mesh”, “subtype”:”Cube”, “owner”:”123”, “transform”:[1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0], “assets”: [“Asset_5”]

}

]

}

9.2.5 Object Destroy

{ “msg_type”:3, “err_code”:100, “num_records”:1, “objects”:[

{ “key”:”5951dd759af59c00015b1408”, “transform”:[1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0]

}

]

}

9.2.6 Object Query

{ “msg_type”:4, “err_code”:100, “num_records”:2, “objects”:[

{ “name”:”Test Object 123465”, “scene”:”DEFGHIJ123465”, “type”:”Mesh”, “subtype”:”Cube”, “owner”:”456”, “transform”:[1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0], “assets”: [“Asset_5”]

}, {

“name”:”Test Object 123456”, “scene”:”DEFGHIJ123456”, “type”:”Curve”, “subtype”:”Sphere”, “owner”:”456”, “transform”:[1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0], “assets”: [“Asset_5”]

}

]

}

9.2.7 Object Lock

{ “msg_type”:5, “err_code”:100, “num_records”:1, “objects”:[

{ “key”:”59ab6e44ac48b7000148c86b”, “name”:”Test Object 123465”, “scene”:”DEFGHIJ123465”, “type”:”Mesh”, “subtype”:”Cube”, “owner”:”10”, “transform”:[1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,4.0,6.0,0.0,0.0,0.0,0.0,0.0,1.0]

}

]

}

9.2.8 Object Unlock

{ “msg_type”:6, “err_code”:100, “num_records”:1, “objects”:[

{ “key”:”59ab6e44ac48b7000148c86b”, “name”:”Test Object 123465”, “scene”:”DEFGHIJ123465”, “type”:”Mesh”, “subtype”:”Cube”, “owner”:”10”, “transform”:[1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,8.0,14.0,0.0,0.0,0.0,0.0,1.0]

}

]

}

CHAPTER 10

Appendix B: Error Codes

NO_ERROR = 100

Operation was successful

ERROR = 101

An unknown error occurred

NOT_FOUND = 102

Data was not found

TRANSLATION_ERROR = 110

JSON/Protocol Buffer parsing error

PROCESSING_ERROR = 120

Unknown error occurred during processing stage of execution

BAD_MSG_TYPE_ERROR = 121

An invalid msg_type was recieved (valid values are integers from 0 to 4)

INSUFF_DATA_ERROR = 122

Insufficient data received on message to form a valid response

LOCK_EXISTS_ERROR = 123

A Device Lock Exists on the Object

Go Home

CHAPTER 11

Architecture

This is designed to be used as a microservice within a larger architecture. This will take in CRUD messages for objects in 3 space, and track location, rotation, and scaling. It will also ensure that any updates are sent out on a separate port to allow streaming to all user devices.

A .proto file is included to allow generating the bindings for any language which are generated by the protocol buffer compiler (https://developers.google.com/protocol-buffers/). Alternatively, Clyman can be configured to accept JSON messages.

0-Meter (https://github.com/AO-StreetArt/0-Meter) has been developed to allow easy testing of the module with JSON message formats.

CLyman can also be deployed with Consul (https://www.consul.io/) as a Service Discovery and Distributed Configuration architecture. This requires a Consul Agent to be deployed that CLyman can connect to.

CLyman depends on Mongo (database), and Kafka (data streaming) instances in order to function.

CHAPTER 12

Design

In order to allow for real-time, distributed visualization, one of the key problems that needs to be solved is allowing for real-time communication between devices regarding an object's position, orientation, and scaling. This aims to solve that problem by providing a messaging API that is used to update objects, and a corresponding feed outbound which tells all devices about the update.

Go Home

CHAPTER 13

Dependencies

Go Home

CLyman is built on top of the work of many others, and here you will find information on all of the libraries and components that CLyman uses to be successful.

Licenses for all dependencies can be found in the licenses folder within the repository.

13.1 CppKafka

CppKafka is a wrapper on top of librdkafka, which provides quick and easy access to pushing Kafka messages.

CppKafka is released under a BSD License.

13.2 ZeroMQ

Zero MQ is a lightweight messaging library that CLyman uses to communicate. It is fast, versatile, and has bindings for many major languages.

Zero MQ is released under an LGPL License.

13.3 CppZmq

CppZmq is the C++ binding for libzmq, which was written in C.

CppZmq is released under an MIT License.

13.4 Log4cpp

Log4Cpp is a logging library based on Log4j.

Log4Cpp is released under an LGPL License.

13.5 Eigen

Eigen is a Linear Algebra library.

Eigen is released under an MPL License.

13.6 RapidJson

RapidJson is a very fast JSON parsing/writing library.

RapidJson is released under an MIT License.

13.7 AO Shared Service Library

AOSSL is a collection of C++ wrappers on many of the C libraries listed here.

AOSSL is released under an MIT License.

13.8 LibBson & LibMongoc

LibBson and LibMongoc are used to communicate with Mongo, a Document Based Database.

LibBson and LibMongoc are released under Apache 2 Licenses.

13.9 LibUUID

LibUUID is a linux utility for generating Universally Unique ID's.

LibUUID is released under a BSD License.

13.10 LibCurl

LibCurl is a ubiquitous networking library.

LibCurl is released under an MIT License.

13.11 LibProtobuf

LibProtobuf and the Protocol Buffer Compiler comprise a serialization system which CLyman can use to communicate in lieu of JSON. You can find more information about Protocol Buffers at the Google Developer Site

The Protocol Buffer License is unique yet very unrestrictive. For more information please see the license itself

13.12 DVS Interface

Finally, we also depend on the DVS Interface Library which houses a collection of .proto files for this project.

DVS Interface is released under an MIT License.

13.13 Automatic Dependency Resolution

For Ubuntu 16.04 & Debian 7, the build_deps.sh script should allow for automatic resolution of dependencies.

13.14 Other Acknowledgements

Here we will try to list authors of other public domain code that has been used:

```
René Nyffenegger - Base64 Decoding Methods
```

CHAPTER 14

Developer Notes

This page contains a series of notes intended to be beneficial for any contributors to CLyman.

14.1 Development Docker Image

Generating a development Docker Image is made easy by the DebugDockerfile. This image is unique in that it does not enter directly into CLyman, but rather installs all of the necessary dependencies and then waits.

First, execute the below command from the root folder of the project to build your local debug image: `docker build --no-cache --file DebugDockerfile -t "aostreetart/clyman:debug" .`

Once this completes, run your image with the below command: `docker run --name clyman -p 5555:5555 -d aostreetart/clyman:debug`

You can update the port number to whatever you like, and keep in mind that you may also need to connect the container to a docker network, depending on your configuration. For example: `docker run --name clyman --network=dvs -p 5555:5555 -d aostreetart/clyman:debug`

Finally, you can open up a terminal within the box with: `docker exec -i -t clyman /bin/bash`

The container will have CLyman and all it's dependencies pre-installed, so you can get right to work!

14.2 Generating Releases

The release_gen.sh script is utilized to generate releases for various systems. It accepts three command line arguments:
* the name of the release: crazyivan-*os_name-os_version* * the version of the release: we follow semantic versioning
* the location of the dependency script: current valid paths are linux/deb (uses apt-get) and linux/rhel (uses yum)

Read About CLyman Automated Testing

Go Home

CHAPTER 15

Automated Testing

Crazy Ivan uses Travis CI for automated testing.

Within the Travis CI Configuration, several steps are executed to complete full functional testing:

- Set up Docker instances of Mongo, and Consul, and then populate the KV Store in Consul with several configuration values.
- Build a new Docker Image for Crazy Ivan and start it.
- Download 0-Meter. This is a custom tool developed for 0MQ load testing, and is used to send a series of messages to Crazy Ivan over the course of the tests. The configuration for 0-Meter CI Tests can be found in the ci/ folder.
- Run 0-Meter to send a series of messages, some expected to fail and others to succeed, to Crazy Ivan. Validate the err_code field in the response.
- If all tests pass, then push the newly built image to Docker Hub.

Note that unit tests are performed within the Dockerfile itself, so that the Docker build will fail if any unit tests fail. If you are adding unit tests to Crazy Ivan, you should add them within the Dockerfile as well.

Go Home

CHAPTER 16

CLyman

16.1 Overview

CLyman is a C++ microservice which synchronizes high-level 3-D object attributes across many user devices. The goal is to synchronize the position, rotation, and scale of virtual objects projected into a real space.

This service is intended to fill a small role within a larger architecture designed to synchronize 3D objects across different client programs. It is highly scalable, and many instances can run in parallel to support increasing load.

Detailed documentation can be found on ReadTheDocs.

16.2 Features

- Storage of 3-D Objects Location, Rotation, Scaling
- Enable real-time change feeds on the objects which are stored
- Connect to other services over Zero MQ using JSON or Google Protocol Buffers.
- Configurable Logic
- Scalable microservice design

CLyman is a part of the AO Aesel Project, along with Crazy Ivan. It therefore utilizes the DVS Interface library, also available on github. It utilizes the Obj3.proto file for inbound communications when configured to read protocol buffers.

Stuck and need help? Have general questions about the application? Reach out to the development team at clyman@emaillist.io