CrazyIvan Documentation

Release 2.0.0

AO

Apr 17, 2019

Contents:

CHAPTER 1

Overview

Crazy Ivan aims to solve several problems necessary to enabling Augmented Reality experiences that multiple individuals can interact with. It is one critical piece in joining the digital world and the physical world together seamlessly. On top of this unique functionality is a robust, reliable, and secure core, capable of enabling multi-user animation, design, gaming, and more. Crazy Ivan is up to the challenge of both the digital and physical realms, and that makes it unique.

This is a service designed to store 'Scenes'. A Scene is a collection of Renderable Objects (possibly associated to some geographic coordinate), which users can move between. These scenes can be anything, ranging from levels in a traditional video game, to apartments in a building.

Devices can join and leave scenes as they move through the world, and as they do we build a network of relationships that can be used to determine the transformations needed for other devices. Crazy Ivan also serves as a UDP Server, communicating Updates out to registered devices.

As you decide on the foundation for your next application, you should choose one that will grow with you, both in terms of scale and in terms of technology. It's time to stop settling for 'good enough' when it comes to interactive, shared visuals.

Detailed documentation can be found on ReadTheDocs.

CHAPTER 2

Features

- Storage of Scenes (Groups of virtual objects & user devices)
- Track User Devices moving between Scenes to create a connected network of coordinate systems
- Means to store manual corrections of coordinate system transformations from users
- Efficient calculation of coordinate system transformations between Scenes based on existing data
- Stream updates via UDP to Registered User Devices
- Multi-layered security
- Scalable and Flexible Deployment Strategies

Crazy Ivan is a part of the AO Aesel Project, along with CLyman.

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

2.1 Getting Started with CrazyIvan

Go Home

2.1.1 Docker

Using the Crazy Ivan Docker image is as simple as:

```
docker run --publish=8766:8766 --publish=8764:8764/udp aostreetart/crazyivan:v2
```

However, we also need a running instance of Neo4j to do anything interesting. To get you up and running quickly, a Docker Compose file is provided. To start up a Neo4j instance and a Crazy Ivan instance, simply run the following from the 'compose/min' folder:

```
docker-compose up
```

Alternatively, you can deploy the stack with Docker Swarm using:

```
docker stack deploy --compose-file compose/min/docker-compose.yml ivan-stack
```

Once the services have started, test them by hitting Ivan's healthcheck endpoint:

```
curl http://localhost:8766/health
```

The Transaction (HTTP) API is available on port 8766, and the Event (UDP) API is available on port 8764. Keep in mind that this is not a secure deployment, but is suitable for exploring the *Crazy Ivan API*.

You may also continue on to the discussion of *How to Use Crazy Ivan*.

2.1.2 Shutdown

Shutdown of Crazy Ivan can be initiated with a kill or interrupt signal to the container, or with 'docker stop'. However, at least one udp message must be received afterwards in order to successfully shut down the main event thread. You can send one with:

```
echo "kill" | nc -u $(ip addr show eth0 | grep -Po 'inet \K[\d.]+') 8764
```

Replacing 'eth0' with your network device, if necessary.

2.1.3 Latest Release

Download and unzip the latest release file from https://github.com/AO-StreetArt/CrazyIvan/releases.

Once you have done this, you can run the easy_install script with the -d option to install dependencies and the Crazy Ivan executable. Alternatively, you can simply run the install_deps.sh script from the scripts/ folder, and then run the crazy_ivan executable from the main release folder.

```
./crazy_ivan
```

In order to run CrazyIvan, you will need a Neo4j Server installed locally. Instructions can be found at https://neo4j.com/developer/get-started/, or Neo4j can be started via a Docker image:

```
docker run -d --publish=7474:7474 --publish=7687:7687 --env=NEO4J_AUTH=none --volume=
↪$HOME/neo4j/data:/data --name=database neo4j
```

Either way, the default connection for CrazyIvan will connect without authentication.

You can move on to explore the *Crazy Ivan API*, or check out the *Configuration Section* for more details on the configuration options available when starting CrazyIvan.

You may also continue on to the discussion of *How to Use Crazy Ivan*.

2.1.4 Building from Source

The recommended system for development of CrazyIvan is either Ubuntu 18.04 or CentOS7. You will need gcc 6.0 or greater and gnu make installed to successfully compile the program.

- Ubuntu

```
sudo apt-get install gcc-6 g++-6
export CC=gcc-6
export CXX=g++-6
```

- Redhat

https://www.softwarecollections.org/en/scls/rhscl/devtoolset-6/

Next, you'll need to clone the repository and run the build_deps script. This will install all of the required dependencies for Crazy Ivan, and may take a while to run.

```
git clone https://github.com/AO-StreetArt/CrazyIvan.git
mkdir crazyivan_deps
cp CrazyIvan/scripts/deb/build_deps.sh crazyivan_deps/build_deps.sh
cd crazyivan_deps
sudo ./build_deps.sh
```

You will also need to ensure that the POCO dependency is on the linker path, which can be done with:

```
export LD_LIBRARY_PATH="/usr/local/lib:$LD_LIBRARY_PATH"
```

Now, we can build Crazy Ivan:

```
cd ../CrazyIvan
make
```

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

```
./crazy_ivan
```

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

You may also build the test executable in the tests/ directory with:

```
make tests
```

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

- A Neo4j Server installed locally. Instructions can be found at https://neo4j.com/developer/get-started/

Neo4j can be started via a Docker image:

```
docker run -d --publish=7474:7474 --publish=7687:7687 --env=NEO4J_AUTH=none --volume=
↪$HOME/neo4j/data:/data --name=database neo4j
```

Either way, the default connection for CrazyIvan will connect without authentication.

You can move on to explore the *Crazy Ivan API*, or check out the *Configuration Section* for more details on the configuration options available when starting CrazyIvan.

You may also continue on to the discussion of *How to Use Crazy Ivan*.

2.1.5 Shutdown

Shutdown of Crazy Ivan can be initiated with a kill or interrupt signal to the main thread. However, at least one udp message must be received afterwards in order to successfully shut down the main event thread. You can send one with:

```
echo "kill" | nc -u $(ip addr show eth0 | grep -Po 'inet \K[\d.]+') 8764
```

Replacing 'eth0' with your network device, if necessary.

2.2 How to Use Crazy Ivan

2.2.1 Overview

Use of Crazy Ivan generally boils down to 5 steps:

1. Build Scenes
2. Calibrate Scenes
3. Start Event Streams
4. Register Devices
5. Stream Events

A Scene is a group of objects, that may or may not be associated to a latitude/longitude point. The updates in each scene are made with respect to a single coordinate-system, known as the Scene Coordinate System. There is no single, universal, master coordinate system, but rather these are defined by the relationships between each scene. Once a device has aligned it's coordinate system with that of a single scene, it can move to any other which has a pre-defined relationship. This act of moving between scenes is known as 'registration'.

Event Streams are streams of UDP messages sent from Crazy Ivan to registered devices. Crazy Ivan receives a single UDP message against a scene, and then broadcasts that message out to any devices that need it.

These concepts allow us to build a real-time, collaborative system for use-cases like collaborative animation, gaming, and Augmented Reality applications.

2.2.2 Build Scenes

This task is the starting point for any workflow, and can be accomplished using the *Scene API*.

We can use this API to create all of the Scenes we will need. What scenes we create will vary wildly by use case. For gaming-like applications, each Scene can represent a single level within the game, a single multi-player lobby within the game, or a section of map within a large-scale MMORPG. For augmented-reality applications, a single scene can represent anything from an individual's apartment to a theatre.

2.2.3 Calibrate Scenes

This is an optional step where calibration devices are moved between scenes prior to registration of actual devices. This allows the scene network to be generated prior to devices moving between scenes, guaranteeing that actual devices will be able to move between scenes accurately.

Alternatively, end users can provide the calibration during live-use. Both approaches have drawbacks and benefits, but it is up to the application developers which method to prefer/utilize.

Either way, this is accomplished via the *Registration API*.

2.2.4 Start Event Streams

Prior to using a Crazy Ivan server, each cluster should be assigned particular Scenes to manage the event stream for. This is done using the *Cache API*.

This allows us to ensure that specific instances of Crazy Ivan are streaming to particular groups of devices, so that we can ensure minimal latency in the transmission. It also allows Crazy Ivan to cache scene and device information in-memory, rather than having to hit the database for each Event.

2.2.5 Register Devices

During initial device registration for end-user devices, they may need to calibrate their initial coordinate system with the scene coordinate system. Note that this is only true, in general, for Augmented Reality applications.

Regardless, once devices are first registered, they can then further use the *Registration API* to move between scenes. As they move, they will receive Event Streams for the scenes they are registered to.

As devices move between scenes, they may be moving between different coordinate systems, especially in the case of Augmented Reality. This is where the network we created during calibration comes into play: each device is provided all of the information it needs to resolve the differences between the coordinate systems when it registers for another.

2.2.6 Stream Events

Finally, devices can move about freely between scenes, sending Events which are consumed by other components, and passed to Crazy Ivan for final routing. This is done via the *Event Stream API*

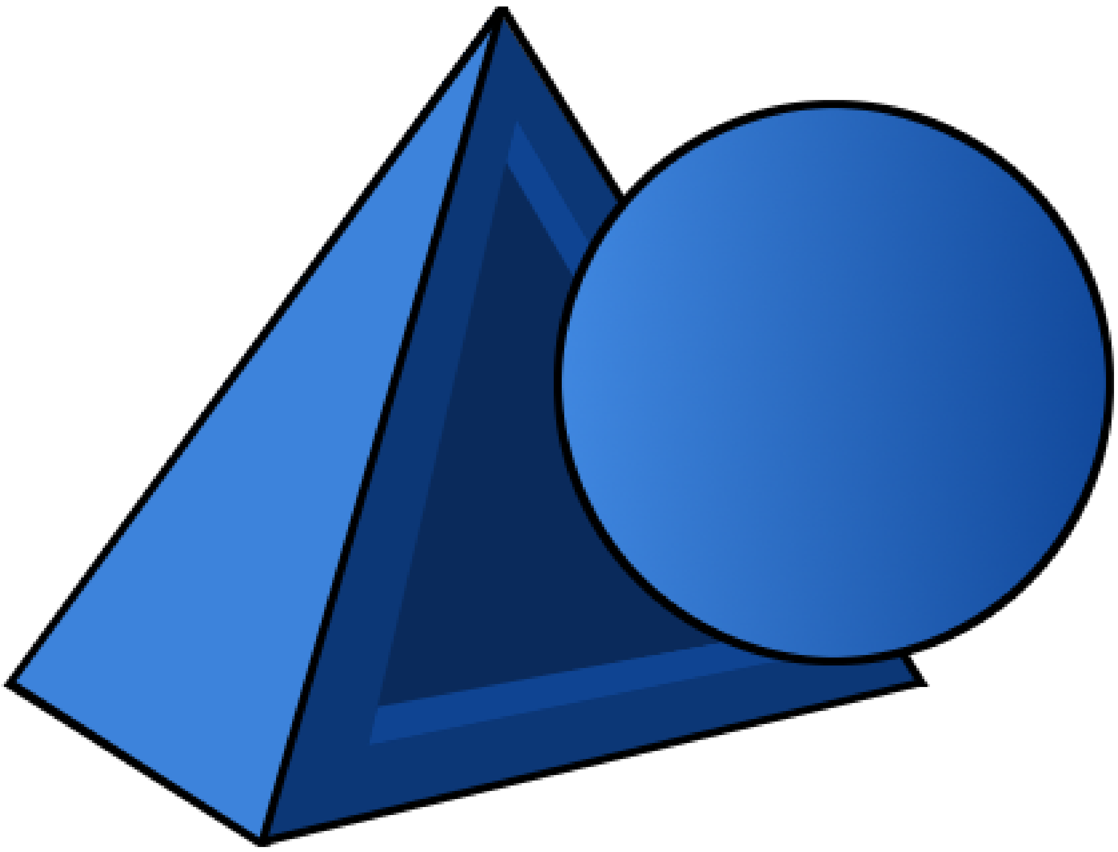
Because of this, Crazy Ivan does not care what is actually in an 'Event'. It only reads the start of the message to find the scene, and beyond that it can contain any text-based message you desire.

Event Streams are designed to be as fast as possible. Communication of events is limited to UDP and/or shared memory, and events can be restricted to specific clusters of Crazy Ivan and other components to ensure minimal network latency.

Optionally, Event Streams can also utilize AES symmetric-encryption, to make sure that the live updates cannot be read by prying eyes.

Go Home

2.3 API Documentation



2.3.1 Scene API

A Scene is a group of Objects associated to a particular Latitude and Longitude. This API exposes CRUD and Query operations for Scenes.

Scene Creation

`POST /v1/scene/`

Create a new scene. A new key will be generated and returned in the response.

Request Headers

- Content-Type – Application/json

Status Codes

- 200 OK – Success

http

```
POST /v1/scene HTTP/1.1
Host: localhost:5885
Content-Type: application/json

{
  "scenes": [
    {
      "name": "testScene",
      "region":"US-MD",
      "latitude":124,
      "longitude":122,
      "assets":["TestAsset10"],
      "tags":["Testing2"]
    }
  ]
}
```

curl

```
curl -i -X POST http://localhost:5885/v1/scene -H 'Content-Type: application/json' --
↪data-raw '{"scenes": [{"name": "testScene", "tags": ["Testing2"], "region": "US-MD",
↪ "longitude": 122, "latitude": 124, "assets": ["TestAsset10"]}]}'
```

wget

```
wget -S -O- http://localhost:5885/v1/scene --header='Content-Type: application/json' -
↪-post-data='{"scenes": [{"name": "testScene", "tags": ["Testing2"], "region": "US-MD
↪", "longitude": 122, "latitude": 124, "assets": ["TestAsset10"]}]}'
```

httpie

```
echo '{
  "scenes": [
    {
      "assets": [
        "TestAsset10"
      ],
      "latitude": 124,
      "longitude": 122,
      "name": "testScene",
      "region": "US-MD",
      "tags": [
        "Testing2"
      ]
    }
  ]
}' | http POST http://localhost:5885/v1/scene Content-Type:application/json
```

python-requests

```
requests.post('http://localhost:5885/v1/scene', headers={'Content-Type': 'application/
↪json'}, json={'scenes': [{'name': 'testScene', 'tags': ['Testing2'], 'region': 'US-
↪MD', 'longitude': 122, 'latitude': 124, 'assets': ['TestAsset10']}]})
```

response

```
HTTP/1.1 200 OK
Location: http://localhost:5885/v1/scene
Content-Type: application/json

{
  "num_records":1,
  "scenes":[{"key":"jklmnop"}]
}
```

Scene Creation

PUT /v1/scene/(*key*)
Create a new scene. The provided key will be assigned to the scene.

Request Headers

- Content-Type – Application/json

Status Codes

- 200 OK – Success

http

```
POST /v1/scene/jklmnop HTTP/1.1
Host: localhost:5885
Content-Type: application/json

{
  "scenes": [
    {
      "name": "testScene",
      "region":"US-MD",
      "latitude":124,
      "longitude":122,
      "assets":["TestAsset10"],
      "tags":["Testing2"]
    }
  ]
}
```

curl

```
curl -i -X POST http://localhost:5885/v1/scene/jklmnop -H 'Content-Type: application/
↪json' --data-raw '{"scenes": [{"name": "testScene", "tags": ["Testing2"], "region":
↪"US-MD", "longitude": 122, "latitude": 124, "assets": ["TestAsset10"]}]}'
```

wget

```
wget -S -O- http://localhost:5885/v1/scene/jklmnop --header='Content-Type:␣
↪application/json' --post-data='{"scenes": [{"name": "testScene", "tags": ["Testing2
↪"], "region": "US-MD", "longitude": 122, "latitude": 124, "assets": ["TestAsset10"]}
↪]}'
```

httpie

```
echo '{
  "scenes": [
    {
      "assets": [
        "TestAsset10"
      ],
      "latitude": 124,
      "longitude": 122,
      "name": "testScene",
      "region": "US-MD",
      "tags": [
        "Testing2"
      ]
    }
  ]
}' | http POST http://localhost:5885/v1/scene/jklmnop Content-Type:application/json
```

python-requests

```
requests.post('http://localhost:5885/v1/scene/jklmnop', headers={'Content-Type':
↪'application/json'}, json={'scenes': [{'name': 'testScene', 'tags': ['Testing2'],
↪'region': 'US-MD', 'longitude': 122, 'latitude': 124, 'assets': ['TestAsset10']}]})
```

response

```
HTTP/1.1 200 OK
Location: http://localhost:5885/v1/scene
Content-Type: application/json

{
  "num_records":1,
  "scenes":[{"key":"jklmnop"}]
}
```

Scene Get

GET /v1/scene/(*key*)

Get a Scene by key.

Status Codes

- 200 OK – Success

http

```
GET /v1/scene HTTP/1.1
Host: localhost:5885
```

curl

```
curl -i http://localhost:5885/v1/scene
```

wget

```
wget -S -O- http://localhost:5885/v1/scene
```

httpie

```
http http://localhost:5885/v1/scene
```

python-requests

```
requests.get('http://localhost:5885/v1/scene')
```

response

```
HTTP/1.1 200 OK
Location: http://localhost:5885/v1/scene
Content-Type: application/json

{
    "msg_type": 2,
    "err_code": 100,
    "num_records": 1,
    "start_record": 0,
    "scenes": [
        {
            "key": "855ca840-c864-11e8-9a4a-309c23d74017",
            "name": "testScene",
            "region": "us-ga",
            "latitude": 100,
            "active": true,
            "longitude": 100,
            "distance": 0,
            "assets": [
                "asset1"
            ],
            "tags": [
                "tag1"
            ],
            "devices": []
        }
    ]
}
```

Scene Update

POST /v1/scene/(*key*)

Update an existing Scene with the specified key.

Request Headers

- Content-Type – Application/json

Status Codes

- 200 OK – Success

http

```
POST /v1/scene/name HTTP/1.1
Host: localhost:5885
Content-Type: application/json

{
```

(continues on next page)

(continued from previous page)

```
  "scenes": [
    {
      "name": "testScene",
      "region":"US-MD",
      "latitude":124,
      "longitude":122,
      "assets":["TestAsset10"],
      "tags":["Testing2"]
    }
  ]
}
```

curl

```
curl -i -X POST http://localhost:5885/v1/scene/name -H 'Content-Type: application/json
↪' --data-raw '{"scenes": [{"name": "testScene", "tags": ["Testing2"], "region": "US-
↪MD", "longitude": 122, "latitude": 124, "assets": ["TestAsset10"]}]}'
```

wget

```
wget -S -O- http://localhost:5885/v1/scene/name --header='Content-Type: application/
↪json' --post-data='{"scenes": [{"name": "testScene", "tags": ["Testing2"], "region
↪": "US-MD", "longitude": 122, "latitude": 124, "assets": ["TestAsset10"]}]}'
```

httpie

```
echo '{
  "scenes": [
    {
      "assets": [
        "TestAsset10"
      ],
      "latitude": 124,
      "longitude": 122,
      "name": "testScene",
      "region": "US-MD",
      "tags": [
        "Testing2"
      ]
    }
  ]
}' | http POST http://localhost:5885/v1/scene/name Content-Type:application/json
```

python-requests

```
requests.post('http://localhost:5885/v1/scene/name', headers={'Content-Type':
↪'application/json'}, json={'scenes': [{'name': 'testScene', 'tags': ['Testing2'],
↪'region': 'US-MD', 'longitude': 122, 'latitude': 124, 'assets': ['TestAsset10']}]})
```

response

```
HTTP/1.1 200 OK
Location: http://localhost:5885/v1/scene
Content-Type: application/json

{
```

(continues on next page)

(continued from previous page)

```
  "num_records":1,
  "scenes":[{"key":"jklmnop"}]
}
```

Scene Deletion

DELETE /v1/scene/(*key*)

Delete a scene.

CAUTION: This will delete all information associated to a scene, including all objects in the scene, and any registrations to devices. Any object which needs to be retained should be moved to another scene prior to deletion.

Status Codes

- 200 OK – Success

http

```
DELETE /v1/scene/name HTTP/1.1
Host: localhost:5885
```

curl

```
curl -i -X DELETE http://localhost:5885/v1/scene/name
```

wget

```
wget -S -O- --method=DELETE http://localhost:5885/v1/scene/name
```

httpie

```
http DELETE http://localhost:5885/v1/scene/name
```

python-requests

```
requests.delete('http://localhost:5885/v1/scene/name')
```

Scene Query

POST /v1/scene/query

Devices can find scenes by any attribute, including distance.

The fields 'latitude', 'longitude', and 'distance' should always appear together if present. The distance provided is taken in kilometers.

Request Headers

- Content-Type – Application/json

Status Codes

- 200 OK – Success

http

```
POST /v1/scene/query HTTP/1.1
Host: localhost:5885
Content-Type: application/json

{
  "scenes":[
    {
      "name":"test",
      "region":"US-MD",
      "latitude":124,
      "longitude":122,
      "assets":["TestAsset10"],
      "tags":["Testing2"]
    }
  ]
}
```

curl

```
curl -i -X POST http://localhost:5885/v1/scene/query -H 'Content-Type: application/
↪json' --data-raw '{"scenes": [{"name": "test", "tags": ["Testing2"], "region": "US-
↪MD", "longitude": 122, "latitude": 124, "assets": ["TestAsset10"]}]}'
```

wget

```
wget -S -O- http://localhost:5885/v1/scene/query --header='Content-Type: application/
↪json' --post-data='{"scenes": [{"name": "test", "tags": ["Testing2"], "region": "US-
↪MD", "longitude": 122, "latitude": 124, "assets": ["TestAsset10"]}]}'
```

httpie

```
echo '{
  "scenes": [
    {
      "assets": [
        "TestAsset10"
      ],
      "latitude": 124,
      "longitude": 122,
      "name": "test",
      "region": "US-MD",
      "tags": [
        "Testing2"
      ]
    }
  ]
}' | http POST http://localhost:5885/v1/scene/query Content-Type:application/json
```

python-requests

```
requests.post('http://localhost:5885/v1/scene/query', headers={'Content-Type':
↪'application/json'}, json={'scenes': [{'name': 'test', 'tags': ['Testing2'], 'region
↪': 'US-MD', 'longitude': 122, 'latitude': 124, 'assets': ['TestAsset10']}]})
```

response

```
HTTP/1.1 200 OK
Location: http://localhost:5885/v1/scene/query
Content-Type: application/json

{
"num_records":1,
  "scenes":[
    {
    "key":"jklmnop",
    "name":"TestScene10",
    "region":"US-MD",
    "latitude":124.0,
    "longitude":122.0,
    "tags":["test","test2"],
    "asset_ids":["asset1","asset2"]
    }
  ]
}
```

2.3.2 Scene Registration API

Devices need to register/de-register to scenes as they move around in the world, and Aesel uses this information to determine what object updates to stream out to that device. This API allows for registration, de-registration, and synchronization of devices to scenes.

Scene Registration

POST /v1/register

Devices are expected to register to scenes as they move through space. This tells Aesel what objects that device needs to receive information on. If the specified scene is not present, then it will be created. The response will contain the transform to the desired scene, as well as the ID of the scene the transform is coming from.

Request Headers

- Content-Type – Application/json

Status Codes

- 200 OK – Success

http

```
POST /v1/register HTTP/1.1
Host: localhost:5885
Content-Type: application/json

{
  "scenes":[
    {
      "key":"jklmnop",
      "devices":[
        {
          "key":"Ud132",
          "hostname": "localhost",
                            "port":4444,
                            "connection_string": "127.0.0.1:4444",
```

(continues on next page)

(continued from previous page)

```
          "transform":{
            "translation":[0,0,0],
            "rotation":[0,0,0]
          }
        }
      ]
    }
  ]
}
```

curl

```
curl -i -X POST http://localhost:5885/v1/register -H 'Content-Type: application/json'␣
↪--data-raw '{"scenes": [{"devices": [{"connection_string": "127.0.0.1:4444",
↪"hostname": "localhost", "port": 4444, "key": "Ud132", "transform": {"translation":␣
↪[0, 0, 0], "rotation": [0, 0, 0]}}], "key": "jklmnop"}]}'
```

wget

```
wget -S -O- http://localhost:5885/v1/register --header='Content-Type: application/json
↪' --post-data='{"scenes": [{"devices": [{"connection_string": "127.0.0.1:4444",
↪"hostname": "localhost", "port": 4444, "key": "Ud132", "transform": {"translation":␣
↪[0, 0, 0], "rotation": [0, 0, 0]}}], "key": "jklmnop"}]}'
```

httpie

```
echo '{
  "scenes": [
    {
      "devices": [
        {
          "connection_string": "127.0.0.1:4444",
          "hostname": "localhost",
          "key": "Ud132",
          "port": 4444,
          "transform": {
            "rotation": [
              0,
              0,
              0
            ],
            "translation": [
              0,
              0,
              0
            ]
          }
        }
      ],
      "key": "jklmnop"
    }
  ]
}' | http POST http://localhost:5885/v1/register Content-Type:application/json
```

python-requests

```
requests.post('http://localhost:5885/v1/register', headers={'Content-Type':
↪'application/json'}, json={'scenes': [{'devices': [{'connection_string': '127.0.0.
↪1:4444', 'hostname': 'localhost', 'port': 4444, 'key': 'Ud132', 'transform': {
↪'translation': [0, 0, 0], 'rotation': [0, 0, 0]}}], 'key': 'jklmnop'}]})
```

response

```
HTTP/1.1 200 OK
Location: http://localhost:5885/v1/scene/name/register
Content-Type: application/json

{
    "msg_type": 4,
    "err_code": 100,
    "num_records": 2,
    "scenes": [
        {
            "key": "20dd78a2-9224-11e8-b492-d850e6db3ad1",
            "active": true,
            "distance": 0,
            "assets": [],
            "tags": [],
            "devices": [
                {
                    "key": "12345",
                    "transform": {
                        "translation": [
                            0,
                            0,
                            0
                        ],
                        "rotation": [
                            0,
                            0,
                            0
                        ]
                    }
                }
            ]
        },
        {
            "key": "123456",
            "active": true,
            "distance": 0,
            "assets": [],
            "tags": [],
            "devices": []
        }
    ]
}
```

Scene De-Registration

POST /v1/deregister

Devices are expected to register to scenes as they move through space. This tells Aesel what objects that device needs to receive information on. De-Registration occurs after a device has left the scene and joined others, and is now ready to stop receiving updates on objects within the old scene.

Note that devices are expected to de-register only after registering with a new scene and performing any necessary corrections. This allows a network of transformations to be created, which can be used to pre-calculate those needed for future registrations.

Request Headers

- Content-Type – Application/json

Status Codes

- 200 OK – Success

http

```
POST /v1/deregister HTTP/1.1
Host: localhost:5885
Content-Type: application/json

{
  "scenes":[
    {
      "key":"jklmnop",
      "devices":[
        {
          "key":"Ud132"
        }
      ]
    }
  ]
}
```

curl

```
curl -i -X POST http://localhost:5885/v1/deregister -H 'Content-Type: application/json
↪' --data-raw '{"scenes": [{"devices": [{"key": "Ud132"}], "key": "jklmnop"}]}'
```

wget

```
wget -S -O- http://localhost:5885/v1/deregister --header='Content-Type: application/
↪json' --post-data='{"scenes": [{"devices": [{"key": "Ud132"}], "key": "jklmnop"}]}'
```

httpie

```
echo '{
  "scenes": [
    {
      "devices": [
        {
          "key": "Ud132"
        }
      ],
      "key": "jklmnop"
    }
  ]
}' | http POST http://localhost:5885/v1/deregister Content-Type:application/json
```

python-requests

```
requests.post('http://localhost:5885/v1/deregister', headers={'Content-Type':
↪'application/json'}, json={'scenes': [{'devices': [{'key': 'Ud132'}], 'key':
↪'jklmnop'}]})
```

Scene Synchronization

POST /v1/align

Aesel will not always be able to supply a device with an accurate transformation upon registering to a scene. In particular, this will happen when the device first registers to a scene with no prior registrations, as well as when the network of transformations is first being built and collected. In these cases, the Device will need to supply Aesel with a correction in order to correct the transformation.

Request Headers

- Content-Type – Application/json

Status Codes

- 200 OK – Success

http

```
POST /v1/align HTTP/1.1
Host: localhost:5885
Content-Type: application/json

{
  "scenes":[
    {
      "key":"jklmnop",
      "devices":[
        {
          "key":"Ud132",
          "transform":{
            "translation":[0,0,0],
            "rotation":[0,0,0]
          }
        }
      ]
    }
  ]
}
```

curl

```
curl -i -X POST http://localhost:5885/v1/align -H 'Content-Type: application/json' --
↪data-raw '{"scenes": [{"devices": [{"transform": {"translation": [0, 0, 0],
↪"rotation": [0, 0, 0]}, "key": "Ud132"}], "key": "jklmnop"}]}'
```

wget

```
wget -S -O- http://localhost:5885/v1/align --header='Content-Type: application/json' -
↪-post-data='{"scenes": [{"devices": [{"transform": {"translation": [0, 0, 0],
↪"rotation": [0, 0, 0]}, "key": "Ud132"}], "key": "jklmnop"}]}'
```

httpie

```
echo '{
  "scenes": [
    {
      "devices": [
        {
          "key": "Ud132",
          "transform": {
            "rotation": [
              0,
              0,
              0
            ],
            "translation": [
              0,
              0,
              0
            ]
          }
        }
      ],
      "key": "jklmnop"
    }
  ]
}' | http POST http://localhost:5885/v1/align Content-Type:application/json
```

python-requests

```
requests.post('http://localhost:5885/v1/align', headers={'Content-Type': 'application/
↪json'}, json={'scenes': [{'devices': [{'transform': {'translation': [0, 0, 0],
↪'rotation': [0, 0, 0]}, 'key': 'Ud132'}], 'key': 'jklmnop'}]})
```

2.3.3 Cache API

The Scene Cache holds registered device information to use for Event Streaming. This enables minimal-latency streaming, but also necessitates that scenes are assigned to Crazy Ivan instances before streaming through that instance.

Cache Add

PUT /v1/scene/cache/(*scene*)
Add a scene to the Scene cache.

Status Codes

- 200 OK – Success

Cache Remove

DELETE /v1/scene/cache/(*scene*)
Remove a scene from the Scene cache.

Status Codes

- 200 OK – Success

2.3.4 Event API

An event is an update from an external system which needs to be streamed out to devices registered to a particular scene. An event is recieved via UDP, and then sent via UDP. Scenes should be assigned to an instance of Crazy Ivan via the Cache API. This allows for streaming of the updates with minimal latency.

Once an event is registered, it can be sent via UDP. If configured, it may be encrypted with an AES-256 symmetric key and salt. The first line of the message should be the Scene ID, with the second line of the message conveying the event. The lines should be separated only by the newline character. Crazy Ivan does not decode the event itself, but rather takes the Scene ID from the first line and uses it to perform streaming to other devices.

This means that the only requirement on the format of the messages is that: - They start with a valid Scene ID that has been added to the Crazy Ivan cache. - Then they contain a new line character. - Then they contain the desired message to send. - The total, encrypted, message is no more than 275 characters

For example:

```
_SCENE_KEY_\n{"key": "_OBJ_KEY_", "scene": "_SCENE_KEY_", "transform": [0.
000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000,
0.000, 0.000, 0.000, 0.000, 0.000]}
```

2.3.5 Asset API

The Asset API provides a rapid way to add or remove Scene Assets.

Cache Add

PUT /v1/scene/(*scene*)**/asset/**
asset Add an Asset to a Scene.

Status Codes

- 200 OK – Success

Cache Remove

DELETE /v1/scene/(*scene*)**/asset/**
asset Remove an Asset from a Scene.

Status Codes

- 200 OK – Success

2.4 Configuration

Crazy Ivan can be configured from one or more sources:

- Environment Variables
- Command Line Arguments
- Consul KV Store
- Vault KV Store
- Properties File

The application gives priority to the values retrieved in the above order. This means that an environment variable setting will override any other setting.

Command Line arguments and Properties File keys are lower case, and separated by periods (ie. 'section.key='). Environment Variables, Vault, and Consul keys are all upper case, and are separated by underscores (ie. 'SECTION_KEY=').

All arguments are prefixed with the application name and profile name (ie. 'section.key' becomes 'ivan.prod.section.key'). The profile name can be changed by providing the command line argument 'profile':

```
./crazy_ivan profile=dev
```

You can store multiple profiles in your configuration sources, and then specify which one to use on startup of each instance.

2.4.1 Cluster Name

The 'cluster' option on the command line or in a properties file, or the 'AOSSL_CLUSTER_NAME' environment variable, will set the name of the cluster. A cluster is a grouping of Crazy Ivan instances, which have been assigned particular scenes to manage. Each Crazy Ivan instance is designed to manage a set number of scenes, and this allows for highly optimized streaming of object updates.

The cluster name will affect both how Crazy Ivan registers with Consul, if provided, as well as the names of cluster-specific security properties.

2.4.2 Vault

Vault Address - Starts Crazy Ivan against a Vault instance. Specified by a collection of arguments:

- vault (Environment Variable VAULT) - the address of the vault instance

```
vault=http://localhost:8200
```

- vault.cert (Environment Variable VAULT_CERT) - the location of the SSL certificate

to use when communicating with Vault. You may also leave this blank to enable SSL encryption without providing a client certificate.

```
vault.cert=
```

- vault.authtype (Environment Variable VAULT_AUTHTYPE) - the authentication type

used by Vault, currently supported options are 'APPROLE' and 'BASIC'

```
vault.authtype=BASIC
```

- vault.un (Environment Variable VAULT_UN) - The Username/Role Id for

authenticating with Vault

```
vault.un=test
```

- vault.pw (Environment Variable VAULT_PW) - The Password/Secret Id for

authenticating with Vault

```
vault.pw=test
```

In addition, the Vault UN and PW can be loaded from files on disk, ‘vault_un.txt’ and ‘vault_pw.txt’. This is the recommended method to set authentication info in CI/CD processes within an application container.

2.4.3 Secure Properties

Secure Properties can be loaded from a properties file for development purposes, but in a Production scenario should always be loaded from a Vault instance. Once Crazy Ivan is connected to a Vault instance, the following properties can be loaded:

- CONSUL_SSL_CERT - The SSL Certificate to use when communicating with Consul
- CONSUL_ACL_TOKEN - The ACL Token to use when communicating with Consul
- NEO4J_AUTH_UN - The Username to authenticate with discovered Neo4j instances
- NEO4J_AUTH_PW - The Password to authenticate with discovered Neo4j instances
- {cluster-name}_TRANSACTION_SECURITY_AUTH_USER - The username which will authenticate with Crazy Ivan over HTTP(s)
- {cluster-name}_TRANSACTION_SECURITY_AUTH_PASSWORD - The password which will authenticate with Crazy Ivan over HTTP(s)
- {cluster-name}_TRANSACTION_SECURITY_HASH_PASSWORD - The password for the hashing algorithm used to hash the password prior to storage.
- {cluster-name}_EVENT_SECURITY_OUT_AES_KEY - The key for the AES-256 encryption used for sending UDP messages.
- {cluster-name}_EVENT_SECURITY_OUT_AES_SALT - The salt used for the AES-256 encryption used for sending UDP messages.
- {cluster-name}_EVENT_SECURITY_IN_AES_KEY - The key for the AES-256 encryption used for receiving UDP messages.
- {cluster-name}_EVENT_SECURITY_IN_AES_SALT - The salt used for the AES-256 encryption used for receiving UDP messages.

Secure properties can be loaded from any configuration source, but when loaded from Vault they should be present at the default path (‘secret/’) in the v2 KV Store.

2.4.4 Consul

Consul Address - Starts Crazy Ivan against a Consul instance. Specified by either the *consul* command line argument or the *AOSSL_CONSUL_ADDRESS* environment variable.

```
./crazy_ivan consul=http://127.0.0.1:8500
```

We may also include the arguments:

- consul.cert (Environment Variable AOSSL_CONSUL_SSL_CERT) - The location of the

SSL Certificate to use when communicating with Consul. You may also leave this blank to enable SSL encryption without providing a client certificate.

```
consul.cert=
```

- consul.token (Environment Variable AOSSL_CONSUL_ACL_TOKEN) - The ACL Token to use when communicating with Consul

This will enable property retrieval from Consul KV Store & registering with Consul on start up.

The Consul ACL Token can alternatively be generated from the Consul Secret Store in Vault.

- consul.token.role - The role configured in Vault to use to generate the Consul ACL Token.

```
consul.token.role=consul-role
```

2.4.5 Properties File

Properties File - Starts Crazy Ivan against a Properties File. Specified by either the *props* command line argument or the *AOSSL_PROPS_FILE* environment variable. For example:

```
./crazy_ivan props=app.properties
```

If no properties file is specified, Crazy Ivan will look for one named *app.properties* in both the current working folder, and in /etc/ivan/.

The consul address can also be specified within the properties file, with the key *consul*.

2.4.6 HTTPS Setup

SSL Context Configuration is performed on startup, if enabled. If the following properties are set, then SSL Certs for Crazy Ivan can be generated dynamically from Vault:

- transaction.security.ssl.ca.vault.active - 'true' or 'false'

```
transaction.security.ssl.ca.vault.active=true
```

- transaction.security.ssl.ca.vault.role_name - the name of the role to use to generate the SSL Cert

```
transaction.security.ssl.ca.vault.role_name=test-role
```

- transaction.security.ssl.ca.vault.common_name - The Common-Name to use on the Certificate

```
transaction.security.ssl.ca.vault.common_name=local
```

Otherwise, SSL Certificate Generation can be configured from a file in the current working directory called 'ssl.properties'.

HTTPS must be enabled with the following parameter:

- transaction.security.ssl.enabled - 'true' or 'false'

```
transaction.security.ssl.enabled=true
```

2.4.7 Neo4j Connection

- Neo4j - A full connection string may be supplied here.

```
neo4j=neo4j://username:password@localhost:7687
```

In Production Scenarios it is recommended to use Neo4j Discovery. If it is set to true, then Crazy Ivan will use Consul to find a Neo4j instance, and will dynamically find new instances when it encounters many consecutive failures. This is controlled by the property:

- neo4j.discover - 'true' or 'false'.

```
neo4j.discover=true
```

When enabled, you will want to utilize the secure properties 'NEO4J_AUTH_UN' and 'NEO4J_AUTH_PW' in Vault, in order to store the authorization info for Neo4j securely.

2.4.8 Other Values

There are a number of other options that Crazy Ivan can be provided on startup. Below is an overview of the remaining properties:

- Log File - Path on disk to write logs to

```
log.file=ivan.log
```

- Log Level - Debug, Info, Warning, Error

```
log.level=Debug
```

- HTTP host to register with Consul

```
http.host=127.0.0.1
```

- HTTP Port

```
http.port=8766
```

- UDP Port

```
udp.port=8764
```

- Enable Event (UDP) Encryption

```
event.security.aes.enabled=false
```

- Transaction ID's active or inactive. If active, Crazy Ivan will ensure a Transaction Id is stamped on each message.

```
transaction.id.stamp=True
```

- Format for transactions (HTTP traffic). Currently only json is supported.

```
transaction.format=json
```

- Method for streaming events. Currently only udp is supported.

```
event.stream.method=udp
```

- Format for streaming events. Currently only json is supported

```
event.format=json
```

Go Home

2.5 Security

Crazy Ivan has several forms of security, with one form for transactions (HTTP API), and another form for events (UDP API).

2.5.1 Transactions

Transactional Security utilizes SSL and Basic Auth over HTTP (HTTPS). The username/password can be configured in the application configuration, and SSL will require a valid server key and certificate. The locations can then be entered into the ssl.properties file.

The following commands can be used to generate a self-signed SSL cert, along with a client cert. This can be used to test the secured transactional setup.

openssl req -x509 -newkey rsa:4096 -keyout caKey.key -out caCert.pem -days 365

openssl genrsa -out clientKey.key 2048

openssl req -new -key clientKey.key -out clientCert.csr

openssl x509 -req -in clientCert.csr -CA caCert.pem -CAkey caKey.key -CAcreateserial -out MyClient1.crt -days 1024 -sha256

2.5.2 Events

UDP Events utilize AES-256-cbc encryption, with the key, password, salt, and IV set in the application configuration. AES-256 bit keys can be generated with the below command:

openssl enc -aes-256-cbc -k secret -P -md sha1

Where ‘secret’ is a password for generating the key.

Keep in mind that AES encryption is symmetrical, meaning that the encryption keys must be distributed to the clients in order to encrypt traffic between them and Crazy Ivan. The key and salt are delivered to end user devices after a registration transaction, which is both authenticated and encrypted.

2.5.3 Configuration

Secure configuration values should stored in Hashicorp Vault, with full encryption and authentication enabled. Connecting and authenticating to any service requires accessing at least one secure property in Vault, ensuring that any malicious entities must go through Vault to get into any system in the network.

This does mean that your Vault instance should be carefully guarded: it has all of the keys to the castle. However, it is a system designed specifically to guard these secrets, so when used properly it is one of the best safeguards available, along with a healthy dose of common-sense.

2.6 Deployment

This page includes an overview and notes on full production deployment of Crazy Ivan. A step-by-step walkthrough for setting up a secured, single-node deployment is also available in the *Advanced Walkthrough*.

A full deployment of Crazy Ivan involves several steps:

- Consul Setup

- Neo4j Setup
- Vault Setup
- Crazy Ivan Setup

Each component has it's own encryption and authentication layers.

2.6.1 Crazy Ivan Setup

Crazy Ivan instances are deployed in clusters, with each cluster managing a particular set of scenes. The clusters which contain scenes that may interact should all be connected to the same Neo4j cluster. In other words, User Devices cannot move between Crazy Ivan clusters that run against different Neo4j clusters.

Crazy Ivan can load configuration values from Consul and/or Vault, and uses SSL encryption with HTTP Basic Authentication for transactions. Events (sent via UDP) utilize AES symmetric encryption.

Many configuration values are cluster-specific. This allows us to set, for example, separate encryption keys by cluster.

2.6.2 Consul Setup

Deploying a Consul Cluster is covered in detail on the Consul webpage.

Crazy Ivan uses the Consul KV Store for unsecured configuration values, as well as using Consul for Service Discovery. It can utilize SSL encryption, as well as the ACL layer.

2.6.3 Neo4j Setup

Deploying a Neo4j Cluster is covered in detail here. Note that only HA Clusters are currently supported, utilizing Causal Clustering will provide little to no benefit.

Neo4j in containers is also supported. Either way, once Neo4j servers are active, they need to be registered with Consul in order to be picked up by Service Discovery. This can be done with curl, for example:

curl -X PUT -d '{"ID": "neo4j", "Name": "neo4j", "Tags": ["Primary"], "Address": "localhost", "Port": 7687}' http://127.0.0.1:8500/v1/agent/service/register

In addition, there are several schema optimizations that are recommended. To apply them, run the following against any Neo4j servers:

CREATE CONSTRAINT ON (scn:Scene) ASSERT scn.key IS UNIQUE

CREATE INDEX ON :Scene(key)

Note that these are optional, but are highly recommended in production settings.

2.6.4 Vault Setup

Deploying a Vault Cluster is covered in detail on the Vault webpage.

Crazy Ivan can utilize the following Secret Stores:

- Consul - Generate Consul ACL tokens
- PKI - Generate SSL Certs/Keys
- KV - Store secure configuration options

Advanced Walkthrough

Go Home

2.7 Secured Deployment Walkthrough

2.7.1 Overview

A full deployment of Crazy Ivan involves several steps:

- Consul Setup
- Neo4j Setup
- Vault Setup
- Deploy Crazy Ivan

Here, we'll go through each step and deploy a Crazy Ivan instance which uses encryption and authentication for all communications, and stores sensitive configuration values securely in vault. We will focus on configuration and startup of the above applications, and it is assumed that you have either installed all of the above either from their latest official release, or have running Docker Images of each. You'll also need openssl installed, in order to generate SSL certs.

For those using a containerized infrastructure (ie. Docker Containers), there are a few additional steps you will need to take.

- You may need to bind volumes to each container in order to provide each container

with the correct certificates/keys for SSL/TLS encryption. In a full production deployment, the best way to provide these to each container is via orchestration architecture, such as Kubernetes, Ansible, etc. For the case of this walk-through, however, no such architecture is needed. * If you are going to network your containers together, you'll need to provide SSL Certificates with Common Names that match to each container name for Neo4j, Consul, and Vault. Otherwise, you may get certificate validation errors.

2.7.2 SSL Setup

We will have to generate SSL Certificates for every service, and in this walk-through we'll be self-signing them. This is not a good idea for a production environment, where you should be getting your certificates signed by a valid CA.

We're going to start by adding an entry to the /etc/hosts file. This is to ensure that the hostname we use resolves to only 127.0.0.1, and not ::1. Add the following line to the file:

You will need to enter 'local' as the Common Name during Certificate Generation, this will prevent certificate errors from occurring for the tutorial. Keep in mind that you will need to use your actual host and domain names here for a production deployment.

In order to generate the CA certs we'll use to self-sign the server certificates, run the following:

```
sudo mkdir /var/ssl
sudo mkdir /var/ssl/consul
sudo mkdir /var/ssl/vault
sudo mkdir /var/ssl/neo4j
sudo openssl genrsa -des3 -out /var/ssl/ca.key 4096
sudo openssl req -new -x509 -days 365 -key /var/ssl/ca.key -out /var/ssl/ca.crt
```

Next, we'll add the CA Certificate to the system trusted certificates, to prevent certificate errors during the tutorial. On Redhat/CentOS:

```
sudo cp /var/ssl/ca.crt /etc/pki/ca-trust/source/anchors/
sudo update-ca-trust
```

Ubuntu users can follow the steps here: https://askubuntu.com/questions/73287/how-do-i-install-a-root-certificate

2.7.3 Consul Setup

Before we do anything else, we should go ahead and generate the SSL certificate that Consul will use:

```
sudo openssl genrsa -out /var/ssl/consul/clientKey.key 2048
sudo openssl req -new -key /var/ssl/consul/clientKey.key -out /var/ssl/consul/
↪clientCert.csr
sudo openssl x509 -req -in /var/ssl/consul/clientCert.csr -CA /var/ssl/ca.crt -CAkey /
↪var/ssl/ca.key -CAcreateserial -out /var/ssl/consul/MyClient1.crt -days 1024 -sha256
```

Now, generate an encryption key for Consul gossip:

```
consul keygen
```

Then, take this value and save it in the file 'consul_config.json':

```
{
      "acl_datacenter": "dc1",
      "acl_master_token": "as3cr3t",
      "acl_default_policy": "deny",
      "acl_down_policy": "extend-cache"
  "encrypt": "your-encryption-key-here",
  "encrypt_verify_incoming": true,
  "encrypt_verify_outgoing": true
}
```

Now, we can startup the agent:

```
mkdir consul_data
consul agent -server -bootstrap -data-dir consul_data/ -bind=127.0.0.1 -config-file␣
↪consul_config.json -ui``
```

After this, we'll need to generate an Agent ACL token:

```
curl --request PUT --header "X-Consul-Token: b1gs33cr3t" --data '{"Name": "Agent Token
↪", "Type": "client", "Rules": "{\"key\":{\"\":{\"policy\":\"write\"}},\"node\":{\"\
↪":{\"policy\":\"write\"}},\"service\":{\"\":{\"policy\":\"write\"}},\"agent\":{\"\":
↪{\"policy\":\"write\"}},\"session\":{\"\":{\"policy\":\"write\"}}}"}' http://127.0.
↪0.1:8500/v1/acl/create
```

This will generate a token, that needs to be added into the Consul config file. We'll also go ahead and add our HTTPS information to enable encryption:

```
{
      "acl_datacenter": "dc1",
      "acl_master_token": "b1gs33cr3t",
      "acl_default_policy": "deny",
      "acl_down_policy": "extend-cache"
  "acl_agent_token": "agent-token-here"
  "encrypt": "encryption-key-here",
  "encrypt_verify_incoming": true,
```

(continues on next page)

(continued from previous page)

```
  "encrypt_verify_outgoing": true,
  "addresses": {
    "https": "0.0.0.0"
  },
  "ports": {
    "https": 8289
  },
  "key_file": "/var/ssl/consul/clientKey.key",
  "cert_file": "/var/ssl/consul/MyClient1.crt",
  "ca_file": "/var/ssl/ca.crt"
}
```

Once the agent is restarted with the new configuration, both encryption and authentication fully enabled.

2.7.4 Neo4j Setup

Once again, we'll start by creating SSL Certificates for Neo4j. Create the directory /var/ssl/neo4j. Then, run the below commands to generate a self-signed certificate (in production, you should use a certificate signed by a valid CA).

```
sudo openssl genrsa -des3 -out /var/ssl/neo4j/serv.key 1024
sudo openssl req -new -key /var/ssl/neo4j/serv.key -out /var/ssl/neo4j/server.csr``
sudo openssl x509 -req -days 365 -in /var/ssl/neo4j/server.csr -CA /var/ssl/ca.crt -
↪CAkey /var/ssl/ca.key -set_serial 01 -out /var/ssl/neo4j/server.crt``
sudo openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in /var/ssl/neo4j/serv.
↪key -out /var/ssl/neo4j/server.key``
```

Create the folder /var/ssl/trusted/neo4j, and copy the /var/ssl/neo4j/server.crt file into it.

Then, update the following settings in your Neo4j configuration file:

```
dbms.ssl.policy.default.trusted_dir=/var/ssl/trusted/neo4j
dbms.ssl.policy.default.public_certificate=/var/ssl/neo4j/server.crt
dbms.ssl.policy.default.private_key=/var/ssl/neo4j/server.key
dbms.ssl.policy.default.base_directory=/var/ssl/neo4j/
dbms.connector.https.enabled=true
dbms.connector.https.listen_address=:7473
dbms.connector.bolt.enabled=true
dbms.connector.bolt.tls_level=REQUIRED
```

Neo4j uses a pre-set configuration location for SSL certs to be used by the Bolt connector. In order to install our self-signed certs for use with Bolt, we need to copy them into the folder Neo4j is expecting, with the correct names.

```
sudo cp /var/ssl/neo4j/server.crt /var/lib/neo4j/certificates/neo4j.cert
sudo cp /var/ssl/neo4j/server.key /var/lib/neo4j/certificates/neo4j.key
```

Now, restart the Neo4j server. Once the server is started, it will need to be registered for discovery with Consul. This can be done with curl, for example:

```
curl -X PUT --header "X-Consul-Token: b1gs33cr3t" -d '{"ID": "neo4j", "Name": "neo4j",
↪ "Tags": ["Primary"], "Address": "local", "Port": 7687}' http://127.0.0.1:8500/v1/
↪agent/service/register
```

In addition, the username/password for the instance is normally set on startup in the UI. Be sure to take note of this, as we'll need it to configure Crazy Ivan.

Once Neo4j is started, login to the web browser and run the following

```
CREATE CONSTRAINT ON (scn:Scene) ASSERT scn.key IS UNIQUE
CREATE INDEX ON :Scene(key)
```

2.7.5 Vault Setup

Now, let's generate our SSL Certificate for Vault:

```
sudo openssl genrsa -out /var/ssl/vault/clientKey.key 2048
sudo openssl req -new -key /var/ssl/vault/clientKey.key -out /var/ssl/vault/
↪clientCert.csr``
sudo openssl x509 -req -in /var/ssl/vault/clientCert.csr -CA /var/ssl/ca.crt -CAkey /
↪var/ssl/ca.key -CAcreateserial -out /var/ssl/vault/MyClient1.crt -days 1024 -
↪sha256``
```

We'll be configuring Vault to use the Consul Storage backend, which means we are going to need an ACL token for Vault to use:

```
curl --request PUT --header "X-Consul-Token: b1gs33cr3t" --data '{"Name": "Agent Token
↪", "Type": "client", "Rules": "{\"key\":{\"vault/\":{\"policy\":\"write\"}},\"node\
↪":{\"\":{\"policy\":\"write\"}},\"service\":{\"vault\":{\"policy\":\"write\"}},\
↪"agent\":{\"\":{\"policy\":\"write\"}},\"session\":{\"\":{\"policy\":\"write\"}}}"}
↪' http://127.0.0.1:8500/v1/acl/create
```

Copy the resulting token, then save the below as a file 'vault_config.hcl':

Before starting the Vault server, you may need to add the CA certificate you generated to your system chain. On CentOS/Redhat, this can be done by copying the CA certificate into the /etc/pki/ca-trust/source/anchors directory, and then refreshing the certificate chain:

```
sudo cp /var/ssl/ca.crt /etc/pki/ca-trust/source/anchors
sudo update-ca-trust
```

You may need to reference the documentation for your particular OS otherwise.

Now, we can start the Vault server:

```
vault server -config=vault_config.hcl
```

In a separate terminal, we'll need to configure the Vault.

```
export VAULT_ADDR='https://local:8200'
vault operator init``
```

Save the unseal keys and root key output when we initialize the vault above.

Next, we will unseal the Vault. We'll need to run this operation 3 times, with 3 unique unseal keys.

```
vault operator unseal
```

Before we continue configuring the Vault, we need to login. Be sure to enter the root key you saw during Vault Initialization.

```
vault login root-key-here
```

Our next step is enabling authentication in Vault. Save the following to a file 'vault_admin_policy.hcl':

Now we can enable userpass authentication, and create a user and policy.

```
vault auth enable userpass
vault write auth/userpass/users/test password=test policies=admins
vault policy write admins vault_admin_policy.hcl
```

Now, we can enable our other secrets engines:

```
vault secrets enable -version=2 kv
vault secrets enable pki
vault secrets enable consul
vault secrets tune -max-lease-ttl=8760h pki
```

We'll need to setup Vault to use a management token from Consul:

```
curl --header "X-Consul-Token: b1gs33cr3t" --request PUT --data '{"Name": "sample",
↪"Type": "management"}' http://127.0.0.1:8500/v1/acl/create
```

Copy the resulting token, and pass it to Vault to use:

```
vault write consul/config/access address=127.0.0.1:8500 token=your-token-here
```

To complete the Consul Secrets Engine configuration, we can add a role which Crazy Ivan can use to generate consul ACL tokens.

```
vault write consul/roles/new-role policy=$(base64 <<< 'key "" {policy="read"} service
↪"" {policy="write"}')
```

Next, let's finish the PKI Secrets Engine configuration, which will allow Crazy Ivan to generate SSL Certificates from Vault on startup.

First, we have Vault generate an internal CA certificate (Note that this is not advised in Production scenarios), and signing information:

```
vault write pki/root/generate/internal common_name=my-website.com ttl=8760h
vault write pki/config/urls issuing_certificates="http://127.0.0.1:8200/v1/pki/ca"␣
↪crl_distribution_points="http://127.0.0.1:8200/v1/pki/crl"``
```

Finally, we'll set up another role that allows for generation of SSL Certificates

```
vault write pki/roles/pki-role allowed_domains=local allow_subdomains=true max_ttl=72h
```

2.7.6 Crazy Ivan Setup

Before starting Crazy Ivan, we'll want to populate some configuration values.

Non-secure configuration options can be set in Consul. Most of the defaults will work for us here, so we'll just go ahead and enable authentication in Crazy Ivan HTTPS requests:

```
curl --header "X-Consul-Token: b1gs33cr3t" --request PUT --data 'single' https://
↪local:8500/v1/kv/ivan/prod/IVAN_PROD_TRANSACTION_SECURITY_AUTH_TYPE
```

Secure configuration options can be set in Vault. Let's setup our core encryption information in Vault. First, we enter Event (UDP) encryption settings:

```
vault kv put secret/IVAN_PROD_TEST_EVENT_SECURITY_IN_AES_SALT IVAN_PROD_TEST_EVENT_
↪SECURITY_IN_AES_SALT=test
vault kv put secret/IVAN_PROD_TEST_EVENT_SECURITY_IN_AES_KEY IVAN_PROD_TEST_EVENT_
↪SECURITY_IN_AES_KEY=test
```

(continues on next page)

(continued from previous page)

```
vault kv put secret/IVAN_PROD_TEST_EVENT_SECURITY_OUT_AES_SALT IVAN_PROD_TEST_EVENT_
↪SECURITY_OUT_AES_SALT=test
vault kv put secret/IVAN_PROD_TEST_EVENT_SECURITY_OUT_AES_KEY IVAN_PROD_TEST_EVENT_
↪SECURITY_OUT_AES_KEY=test
```

Next, we setup our authentication information for Neo4j:

```
vault kv put secret/IVAN_PROD_NEO4J_AUTH_UN IVAN_PROD_NEO4J_AUTH_UN=neo4j
vault kv put secret/IVAN_PROD_NEO4J_AUTH_PW IVAN_PROD_NEO4J_AUTH_PW=neo4j
```

Finally, we provide the authentication options for Transactions (HTTP(s)):

```
vault kv put secret/IVAN_PROD_TRANSACTION_SECURITY_AUTH_USER IVAN_PROD_TRANSACTION_
↪SECURITY_AUTH_USER=test
vault kv put secret/IVAN_PROD_TRANSACTION_SECURITY_AUTH_PASSWORD IVAN_PROD_
↪TRANSACTION_SECURITY_AUTH_PASSWORD=test
vault kv put secret/IVAN_PROD_TRANSACTION_SECURITY_HASH_PASSWORD IVAN_PROD_
↪TRANSACTION_SECURITY_HASH_PASSWORD=test
```

Full details on configuration options can be found in the Configuration section of the documentation. Finally, you can start Crazy Ivan with:

```
./crazy_ivan ivan.prod.vault=https://local:8200 ivan.prod.vault.cert= ivan.prod.vault.
↪authtype=BASIC ivan.prod.vault.un=test ivan.prod.vault.pw=test ivan.prod.consul.
↪token.role=new-role ivan.prod.consul=https://local:8289 ivan.prod.consul.cert= ivan.
↪prod.cluster=test ivan.prod.neo4j.discover=true ivan.prod.neo4j.ssl.ca.file=/var/
↪ssl/ca.crt ivan.prod.transaction.security.ssl.ca.vault.active=true ivan.prod.
↪transaction.security.ssl.ca.vault.role_name=pki-role ivan.prod.transaction.security.
↪ssl.ca.vault.common_name=local.local
```

Several files will be created on startup, with the extensions '.key' and '.pem'. These are all of the certificates and keys that Crazy Ivan is using to encrypt the HTTPS connection.

Make sure your server is up using the health check endpoint:

```
curl --user test:test https://local.local/health
```

Go Home

2.8 Architecture

In order to allow for real-time, distributed visualization, one of the key problems that needs to be solved is ensuring that coordinate systems between various user devices and objects are synchronized. A key abstraction in this case is a 'Scene' which is an arbitrary collection of objects and devices. A device can register/de-register from any scene, as well as apply corrections to the coordinate system relationship between it and the scene it's registered to.

This is done by storing relationships between scenes and devices, and then using these to build relationships between scenes themselves. When devices move between these scenes, they will apply corrections. As they apply corrections, we will build a set of known mappings between scenes which should allow users to move without needing any corrections by returning the pre-calculated differences.

2.8.1 Technical Overview

Crazy Ivan is designed to be used as a service within a larger architecture. It will take in CRUD messages for scenes, as well as track user device registrations (both over HTTP).

Running Crazy Ivan requires an instance of Neo4j to connect to in order to perform most functions. Neo4j serves as the back-end database for Crazy Ivan.

Crazy Ivan can also be deployed with Consul as a Service Discovery and Distributed Configuration architecture. This requires the Consul Agent to be deployed that Crazy Ivan can connect to.

Crazy Ivan can be deployed securely using Vault as a secret store and/or intermediate CA.

2.8.2 Object Change Streams (Events)

Object Change Streams ensure that all registered User Devices remain up to date about objects within their scenes. Crazy Ivan receives UDP updates from outside sources, with a specific format, and then forwards the message, once again via UDP, to all registered devices.

The changes streams are designed to be high-speed and high-volume. Crazy Ivan can process many messages in parallel, and registration information is kept up-to-date in a cache for immediate retrieval. A separate background thread periodically loads updated values from Neo4j.

2.8.3 Clustering

Scene-specific clustering is a central idea in Crazy Ivan. This is an idea borrowed from large-scale MMORPG's, in which large maps are broken apart and each piece is run by separate servers. This allows for horizontal scaling of the system to cover additional real-estate, physical or digital.

A cluster name can be provided by Crazy Ivan on startup, and other applications should use this cluster name to identify the appropriate Crazy Ivan to send messages to.

Note that Crazy Ivan clusters using different Neo4j clusters will not be able to store or calculate cross-scene transformations for coordinate systems.

Go Home

2.9 Dependencies

Go Home

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

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

2.9.1 RapidJson

RapidJson is a very fast JSON parsing/writing library.

RapidJson is released under an MIT License.

2.9.2 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.

2.9.3 NeoCpp

NeoCpp is a wrapper on LibNeo4j, which is used to communicate with Neo4j, a Graph Based Database.

NeoCpp is released under an Apache 2 License. LibNeo4j is released under an Apache 2 License.

2.9.4 LibUUID

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

LibUUID is released under a BSD License.

2.9.5 POCO

The POCO Project is a set of libraries for building networked C++ applications.

It is released under the Boost Software License.

2.9.6 Boost

The Boost Project is a set of C++ libraries, that are primarily used for UDP Processing.

It is released under the Boost Software License.

2.9.7 GLM

GLM is the OpenGL math library, and is used to perform transformation calculations between scenes and/or devices.

This is licensed under an MIT license.

2.9.8 Automatic Dependency Resolution

For Ubuntu 16.04 or Centos7, the build_deps.sh scripts should allow for automatic resolution of dependencies.

Developers can utilize the Vagrant image, which will install dependencies in the VM.

End-Users can run the Docker image, which will install dependencies in the container.

2.9.9 Other Acknowledgements

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

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

2.10 Developer Notes

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

2.10.1 Vagrant

We provide a Vagrantfile to setup a development environment, but this requires that you install Vagrant. Once you have Vagrant installed, cd into the main directory and run:

```
vagrant up
```

Once the box starts, you can enter it with:

```
vagrant ssh
```

The Project folder on your machine is synced to the /vagrant folder in the VM, so you will need to move there before building. Once in that folder, you can build the executable and tests:

```
make && make test
```

2.10.2 Packer

A Packer file is provided, which can be used with Hashicorp Packer. Configuration is provided for building a Docker Image, which can be executed with:

```
packer build packer.json
```

This will create a tagged image, which can then be pushed with

```
docker push aostreetart/crazyivan:v2
```

2.10.3 Docker

The Crazy Ivan Docker Hub Repository contains the latest Docker images for Crazy Ivan.

2.10.4 Running Test Cases

Building the tests can be done with:

```
make test
```

Tests cases are run using Catch2 (https://github.com/catchorg/Catch2), a few examples are shown below:

Run all tests:

```
./tests/tests
```

Run only the unit tests:

```
./tests/tests [unit]
```

Run only the integration tests:

```
./tests/tests [integration]
```

2.10.5 Continuous Integration

Travis CI is used to run automated tests against Crazy Ivan each time a commit or pull request is submitted against the main repository. The configuration for this can be updated via the .travis.yml file in the main folder of the project repository.

Latest CI Runs

2.10.6 Documentation

Documentation is built using Sphinx and hosted on Read the Docs.

Updates to documentation can be made in the docs/ folder of the project repository, with files being in the .rst format.

2.10.7 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)

Go Home

2.11 Algorithm Design

2.11.1 Overview

Anyone visiting this page without any prior knowledge of Crazy Ivan should visit both the *Main Page* and the *Usage Page* in order to become familiar with the basic purpose and functionality of the system.

Crazy Ivan stores Scenes and User Devices as nodes in Neo4j, and Transformations (ie. Translation & Rotation) as the edges between nodes. The creation of this network of nodes and edges is the basis of how Crazy Ivan is able to calculate relationships between arbitrary scenes.

The algorithm has two core components:

1. Network Creation
2. Transformation Calculation

2.11.2 Network Creation

We build a scene-scene link when we receive a Scene Synchronization message. This corresponds to a device-supplied correction of the transformation between it's local coordinate system and the specified scene (scene A) coordinate system.

Upon receipt of this message, we check for other scenes the device is registered to and, for each scene we find (scene B), we either:

- Create a new scene-scene link between scene A and scene B.
- Overwrite an existing scene-scene link between scene A and scene B.

Given translation T_A from the device to Scene A, and translation T_B from the device to Scene B, the scene-scene translation is calculated as:

$$T_B * T_A^{-1} \tag{2.1}$$

Given local rotation R_A from the device to Scene A, and local rotation R_B from the device to Scene B, the scene-scene rotation is calculated as:

$$R_A^{-1} * R_B \tag{2.2}$$

with all links represented as 4x4 matrix transformations. The direction of this transformation is from Scene A to Scene B.

Proof

Given coordinate systems A, B, and C, translations T_B and T_C, and rotations R_B and R_C such that, for any point a in A, a can be represented as a point b in B by:

$$b = T_B * a * R_B \tag{2.3}$$

and a can also be represented as a point c in C by:

$$c = T_C * a * R_C \tag{2.4}$$

Apply matrix multiplication to equation (3):

$$a = T_B^{-1} * b * R_B^{-1} \tag{2.5}$$

Using substitution, we find that:

$$c = T_C * T_B^{-1} * b * R_B{-}1 * R_C \tag{2.6}$$

2.11.3 Transformation Calculation

When a device registers to a scene (Scene B), Crazy Ivan checks to see if there are any pre-existing registrations to other scenes for that device. If there are, then Crazy Ivan will take the first scene it finds (Scene A) and use it to calculate a transformation that allows it to move from Scene A to Scene B. In order to find this, Crazy Ivan uses a shortest-path algorithm (built in to Neo4j) to find a route from the node for scene A to the node for scene B, traversing the transformations between them. The resulting series of transformations is then:

- Inverted if necessary, to ensure the same direction on all transformations
- Converted to matrix representation
- multiplied together (locations using LHS multiplication, rotations with RHS multiplication)

This yields a final transformation which can move directly from scene A to scene B.

Proof

Given a positive integer N > 2 and another positive integer 1 < n < N, we have a series of Scenes $S_1, S_2, ..., S_N$, connected by a series of Transforms $T_1, T_2, ..., T_{N-1}$, each containing a translation $T_N(tr)$ and a rotation $T_N(rot)$. We know that an element s_n in S_n can be represented as an element s_{n+1} in S_{n+1} by:

$$s_{n+1} = T_n(tr) * s_n * T_n(rot) \tag{2.7}$$

Given another positive integer m such that m + n < N, we will show by induction that:

$$s_{n+m} = T_{n+m-1}(tr) * T_{n+m-2}(tr) * ... * T_n(tr) * s_n * T_n(rot) * ... * T_{n+m-2}(rot) * T_{n+m-1}(rot) \tag{2.8}$$

For our base case, we will consider that n = 1. In this case, equation 7 simplifies to:

$$s_2 = T_1(tr) * s_1 * T_1(rot) \tag{2.9}$$

Equation 8 simplifies to:

$$s_{m+1} = T_m(tr) * T_{m-1}(tr) * ... * T_1(tr) * s_1 * T_1(rot) * ... * T_{m-1}(rot) * T_m(rot) \tag{2.10}$$

We can also prove this by induction. Our base case will be m=1, in which case the above simplifies to:

$$s_2 = T_1(tr) * s_1 * T_1(rot) \tag{2.11}$$

This is the same as equation 9, which we already know to be true in this case.

Given some k < m, we assume that:

$$s_k = T_k(tr) * T_{k-1}(tr) * ... * T_1(tr) * s_1 * T_1(rot) * ... * T_{k-1}(rot) * T_k(rot) \tag{2.12}$$

Then, by equation 7, we know that:

$$s_{k+1} = T_k(tr) * s_k * T_k(rot) \tag{2.13}$$

Using substitution, we find that:

$$s_{k+1} = T_k(tr) * T_{k-1}(tr) * ... * T_1(tr) * s_1 * T_1(rot) * ... * T_{k-1}(rot) * T_k(rot) \tag{2.14}$$

Now, we have proven the base case of our inductive argument. We can now assume that, for some j < n, that the following is true:

$$s_{j+m} = T_{j+m-1}(tr) * T_{j+m-2}(tr) * ... * T_j(tr) * s_j * T_j(rot) * ... * T_{j+m-2}(rot) * T_{j+m-1}(rot) \tag{2.15}$$

By Equation 7, we know that:

$$s_{(j+m)+1} = T_{j+m}(tr) * s_{j+m} * T_{j+m}(rot) \tag{2.16}$$

Using substitution, we find that:

$$s_{j+m+1} = T_{j+m}(tr) * T_{j+m-1}(tr) * ... * T_j(tr) * s_j * T_j(rot) * ... * T_{j+m-1}(rot) * T_{j+m}(rot) \tag{2.17}$$

This concludes our inductive proof, as the above equation is the same as Equation 8.

Go Home

Crazy Ivan

/v1