Throughout this document, replace VERSION with the current version of the CCU library. For example ccu-VERSION becomes ccu-1.0 if the version is 1.0.

The python -u option causes the python interpreter to buffer stdout and stderr, so this option is important if you want to see output as soon as it is written. This is particularly important for logger.py and ccuhub.py, but I tend to add it to all python programs.

CCU Integration Instructions

This file is a "quickstart" guide for getting started integrating an analytic into the CCU environment.

Table of Contents

Prerequisites

Python and Pip

You must be using a recent version of python 3.x. The library is developed and tested with Python 3.10 and pip 22, but it should work with a wide variety of versions.

Install Python Libraries (ZeroMQ, . . .)

```
cd python
pip install -r requirements.txt
```

If you are using an older version of Python, you might need to install pyyaml as well.

Install the CCU Integration Library

The ccu library is in the python/dist directory, but you should install it from Artifactory:

```
pip install https://artifactory.sri.com/artifactory/cirano-pypi-local/ccu-1.0-py3-none-any
```

You will need to login using your Artifactory user name and API Key.

Test with commands like this:

```
python -c "from ccu import CCU; print(CCU.__version__)"
```

Java

You must be using a recent version of Java. The library is developed and tested with Java 17. It is also known to build with Java 11.

Install Docker (or make sure it is already installed)

On Windows

Follow the instructions here: `https://docs.docker.com/desktop/windows/install/`. You do need to reboot your computer.

On MacOS

Follow the instructions here: `https://docs.docker.com/desktop/mac/install/`.

On Linux

Follow the instructions here: `https://docs.docker.com/engine/install/ubuntu/`. Docker containers using GPUs will require installing the nvidia-container-toolkit package.

Testing the Installation

After installation, Docker Desktop gives you some commands to test the installation. I think `docker run hello-world` or something like that. Or you can run these commands:

```
docker version
docker container list
docker image list
```

The second two commands should not print anything out if this is a new installation.

Terminology

- A image is like a program (an executable). There is just one, but it can have different tags or versions.
- An container is like a process (a running program). There can be many at once.

Messages

Messages are central to how CCU analytics communicate with each other.

In Python, all CCU messages are dictionaries, which can be converted to JSON. That means a dictionary where the keys are strings, and the data can be strings, numbers (integers or floats), binary data (strings encoded as base64), lists, or dictionaries.

Similarly, in Java all messages are maps, which can be converted to JSON. That means a map where the keys are strings, and the data can be strings, integers, floats, binary data encoded as base64 strings, lists, or maps. Messages can also be obtained in JSON format directly; this works better for floating point values.

Messages are documented in the Messages.md file (which creates Messages.pdf and Messages.html files). So if you want to know the exact name or fields or values of a specific message, read that documentation.

All messages should contain a type, which identifies what sort of message it is, a UUID, and a datetime.

Remember that all data that goes in these messages must be simple Python or Java data types (integer, float, string, binary array, list or dictionary). More complex Numpy data types (for example) can not be part of messages, so you need to convert that data.

Also remember that containers are required to ignore all messages that they do not process. Therefore, every CCU.recv call should be followed by an if statement which limits processing to messages the container cares about and ignores all other messages.

Finally, remember that containers should not fail just because there is a problem with one message. If there is a problem with one message processing that message should end (with a good error message to standard out / logging) and processing should continue with the next message.

Changing Your Python Code

In order to integrate your code, you will need to change it in order to read data from message queues and write results to different messages queues.

You should start out by:

1. Reading the material below.
2. Looking at the example code and reading README-Samples.md.
3. Take a look at the documentation here: python/docs/index.html

You will need to include these libraries:

```
import zmq
from ccu import CCU
```

Creating a Message

All messages should contain a type, which identifies what sort of message it is, a UUID, and a datetime. You can create a dictionary with these three fields by using the base_message function, and then adding whatever data you want, like this:

```
message = CCU.base_message('my_message_type',
                           count=102
                           llr=4.2
                           string_data='my_important_data'
                           binary_data=CCU.binary_to_base64string(b'binary data'))
print(message)
```

Or you can add the fields one at a time, like this:

```
message = CCU.base_message('my_message_type')
message['count'] = 102
message['llr'] = 4.2
message['string_data'] = 'my_important_data'
message['binary_data'] = CCU.binary_to_base64string(b'binary data')
print(message)
```

If you have a Numpy float64 age that you want to send in a message, you must convert to a Python float. You can do it like this:

```
message['exact_age'] = numpy.float64(exact_age).item()
```

Sending Messages

The core code for sending a message in Python is this:

```
# Do this just once when starting up your analytic.
socket = CCU.socket(CCU.queues['RESULT'], zmq.PUB)

# Do this every time you need to send a message (created as described above).
CCU.send(socket, message)
```

Getting Messages

The core code you need to read data from a queue in Python is:

```
# Do this just once when starting up your analytic.
socket = CCU.socket(CCU.Queue.VIDEO_MAIN, zmq.SUB)
```

```
# Do this every time you need to block until a message arrives.
# There is also a CCU.recv() method which will not block but will return
# null if there is no message available, and a CCU.recv_throw() method
# which will not block either, but will throw an exception if no message
# is available.
message = CCU.recv_block(socket)
```

- You must import zmq and ccu.
- You create the socket only once per process, and then recv as many messages as you want.
- The socket you create will vary based on what data you want. The common feeds are (there are more):
  - VIDEO_MAIN: main color video feed.
  - AUDIO_ENV: sound from the environment (the other person)
  - AUDIO_SELF: sound from the operator (the user)
  - RESULT: For TA1 -> TA2 messages, and this includes ASR and MT results.
  - ACTION: For TA2 -> Hololens messages.
- In addition to recv_block shown above, there is a recv call, which will return None if no message is available, and a recv_throw, which will throw an zmq.Again exception if there is no message available.

Before actually using a message you should check that it is properly formatted, and you can do this in two ways or in both ways.

First, you can check that the message contains the fields that your container needs. For example:

```
    CCU.check_message_contains(message,['asr_type', 'asr_text', 'uuid'])
```

will return True if the message contains all three fields: asr, type, asr_text, and uuid. This can be put into a processing loop like this:

```
if not CCU.check_message_contains(message,['asr_type', 'asr_text', 'uuid']):
    logger.warning('Got an asr_result message which did not have a asr_type or an asr_text,
    continue
```

Second, you can check that the message contains the fields that a message of its type is expected to have:

```
CCU.check_message(message)
```

Just calling this function will print out messages describing what fields were expected but not there, but will allow your container to try to process the message.
It is recommended to call both of these functions in this order. The second check command will fail if the message does not contain a field that is really needed.

```
CCU.check_message(message)
if not CCU.check_message_contains(message,['asr_type', 'asr_text', 'uuid']):
    logger.warning('Got an asr_result message which did not have a asr_type or an asr_text,
    continue
```

A TA1 Loop

Common TA1 analytics will operate in a loop, reading events from the Hololens and writing messages to the RESULT queue when they find something of interest. An example is below.

```
# Very simple example of a TA1 process.  Not Realistic.
# This container listens for translations and emotions.
# If it gets an angry emotion with an LLR higher than 2.0, then all translations after
# that are considered rude, until another emotion occures or an angry with an LLR 2.0 or lo

# This example focuses on checking messages and adding the right trigger_id data.

import logging
```

```
import zmq

from ccu import CCU

logging.basicConfig(datefmt='%Y-%m-%d %I:%M:%S',
                    format='%(asctime)s %(levelname)s %(message)s',
                    level=logging.DEBUG)
logging.info('TA1 Example Running')

gesture = CCU.socket(CCU.queues['RESULT'], zmq.SUB)
result = CCU.socket(CCU.queues['RESULT'], zmq.PUB)

rude = False
emotion_uuid = None

while True:
    # Waits for a translation.

    # Start out by waiting for a message on the gesture queue.
    message = CCU.recv_block(gesture)

    # Once we get that message, see if it is a pointing message, since
    # that is the only thing we care about.
    if message['type'] == 'translation' or message['type'] == 'emotion':
        # The first thing we should do is check that the message is legal:
        CCU.check_message(message)

        if message['type'] == 'emotion':
            # This function test for the fields we are actually going to use, so if it fail
            # we don't do anything else.
            if not CCU.check_message_contains(message,['name', 'llr', 'uuid']):
                logger.warning('Got an emotion message which did not have a name or a llr,
                continue

            if message['name'] == 'angry' and message['llr'] > 2.0:
                rude = True
                emotion_uuid = message['uuid']
            else:
                rude = False
                emotion_uuid = None

        if message['type'] == 'translation':
            # This function test for the fields we are actually going to use, so if it fail
            # we don't do anything else.
            if not CCU.check_message_contains(message,['asr_type', 'asr_text', 'uuid']):
                logger.warning('Got an asr_result message which did not have a asr_type or
                continue

            logging.debug(f'Found an asr_result with rudeness {rude}.')

            if rude:
                # Creates a simple messsage of type 'rudeness_detected'.
                # The base_message function populates the 'uuid'
                # and 'datetime' fields in the message (which is really just a
```

```
# Python dictionary).
new_message = CCU.base_message('rudeness_detected',
                               text=message['asr_text'],
                               provenance=message['uuid'],
                               trigger_id=[message['uuid'], emotion_uuid])

# Shows how to add fields to an existing message and also how to check for
# a message one at a time.
if 'start_seconds' in message:
    new_message['start_seconds'] = message['start_seconds']
else:
    logger.warning('Translation message does not have a start_seconds fiel
if 'end_seconds' in message:
    new_message['end_seconds'] = message['end_seconds']
else:
    logger.warning('Translation message does not have a end_seconds field,

# Then we check that it has the required fields, and does not have any ext
# But even if it does, we still send it out. (Maybe we should not but we d
if not CCU.check_message(new_message, strict=False):
    logger.warning('Created a message without the required fields, but sen

# Finally, we send it.
CCU.send(result, new_message)
```

When you access fields in a message, it is recommended that you check that the field exists before using it. That way your container will containue to run even if it receives a poorly formed message which is missing required data.

The best way to do this is once "up front" with the CCU.check_message_contains(message, fields) function, but you can also do it with the python "in" operator just before you use the field. Both examples are shown above.

General Python Notes

If your container requires a GPU, then include code like the following to fail quickly and with a specific error message, if there is no GPU:

```
if not torch.cuda.is_available():
    logger.error('No GPU and one is required.)
    sys.exit(-1)
```

Changing Your Java Code

You can see many more examples here: https://gitlab.sri.com/ccu/samples/java

You will need to include these libraries:

```
import org.zeromq.SocketType;
import org.zeromq.ZMQ;
import org.zeromq.ZContext;

import com.sri.speech.ccu.CCU;
```

And almost certainly these as well:

```
import java.util.Map;

import org.json.JSONObject;
```

Creating a Message

All messages should contain a type, which identifies what sort of message it is, a UUID, and a datetime. You can create a dictionary with these three fields by using the base_message function, and then adding whatever data you want, like this:

```
message = CCU.baseMessage("point");
message.put("direction", 30);
message.put("height", 30);
```

Sending Messages

The core code for sending a message in Java is this:

```
// Do this just once when starting up your analytic.
ZMQ.Socket socket = CCU.socket(CCU.Queue.AUDIO_ENV, SocketType.PUB);

// Do this every time you need to send a message (created as described above).
CCU.send(socket, message);
```

Getting Messages

The core code you need to read data from a queue in Java is:

```
// Do this just once when starting up your analytic.
ZMQ.Socket socket = CCU.socket(CCU.Queue.AUDIO_ENV, SocketType.SUB);

// Do this every time you need to block until a message arrives.
// There is also a CCU.recv() method which will not block but will return
// null if there is no message available.
Map message = CCU.recvBlock(socket);
```

you can convert the message to a string like this:

```
String jsonString = new JSONObject(message).toString();
```

- You must import org.zeromq.SocketType, org.zeromq.ZMQ, org.zeromq.ZContext, com.sri.speech.ccu.CCU, and usually org.json.JSONObject.
- You create the socket only once per process, and then recv as many messages as you want.
- The socket you create will vary based on what data you want. The common feeds are (there are more):
  - VIDEO_MAIN: main color video feed.
  - AUDIO_ENV: sound from the environment (the other person)
  - AUDIO_SELF: sound from the operator (the user)
  - RESULT: For TA1 -> TA2 messages, and this includes ASR and MT results.
  - ACTION: For TA2 -> Hololens messages.
- In addition to recv_block shown above, there is a recv call, which will return None if no message is available, and a recvThrow, which will throw an ZMQException exception if there is no message available.

Building An Image and Running a Container

The current target host for CCU is a Tensor Book with these specs:

```
Ubuntu 20.04, Cuda 11.6, GPU is NVIDIA 3080 w 16G, System has 64G
```

Of course, if you have a different version of CUDA installed locally, you might need to build different containers for local use and for Tensor book use.

Also, just because we have Ubuntu 20 on the host machine does **not** mean you should use Ubuntu 20 as the base for your containers! Please don't, unless you really need it. Much better to use a "slim" or focused base image as described below.

Updating an Existing Dockerfile

If you already have a Dockerfile for your project, then you can use that by adding these lines:

```
# This installs the CCU Python library.
RUN --mount=type=secret,id=netrc,dst=/root/.netrc \
    pip install https://artifactory.sri.com/artifactory/cirano-pypi-local/ccu-1.0-py3-none-
```

And then by adding this argument to your docker build commands: `--secret id=netrc,src=../netrc`

These commands assume you have a netrc file in the directory above your Dockerfiles, and it has login information for Artifactory like this:

```
machine artifactory.sri.com
login your-user-name-in-artifactory
password your-api-key-from-artifactory
```

Or you can add these lines, but that puts your API key into your source code, which you might not want to do. Anyone who has access to your source code will have access to the CCU Artifactory repo under your name:

```
# This installs the CCU Python library on your container.
RUN pip install https://username:apikey@artifactory.sri.com/artifactory/cirano-pypi-local/c
```

If that does not work in your environment, then you can copy the wheel file onto your local host and then copy it into the container with these two commands:

```
COPY CCU-VERSION-py3-none-any.whl /app
RUN pip install CCU-VERSION-py3-none-any.whl
```

Creating A New Dockerfile

If you don't already have a container, and a Dockerfile to build it, then start by creating a file called Dockerfile, containing this, for a Python based analytic:

```
# This is a sample Dockerfile for creating containers for the CCU System.

# Using a *-slim base container is important because it
# results in much smaller containers.  I am starting off
# with Python 3.10, but you can start with some different.
# If you really need a "full" Linux installation, such as
# Ubuntu, use this: FROM ubuntu:20.04 or whatever base image
# you want.

FROM python:3.10.3-slim

# Required so that the CCU library (and any other programs) can tell if
# they are running in a container or directly on a host.
ENV CCU_CONTAINER="sri-wav2vec"

# If you using pip anywhere in this Dockerfile, update it:

RUN pip install --upgrade pip

# This next block assumes you have a requirements.txt file and want
# to install all those python libraries now.
```

```
COPY requirements.txt /tmp/requirements.txt
RUN pip install --no-cache-dir -r /tmp/requirements.txt && \
    rm /tmp/requirements.txt

# This block is housekeeping, just creates a directory to put
# everything in one place.

RUN mkdir /app
WORKDIR /app

# These are the ports used by the zmq messsage queues.

EXPOSE 11880-11890
EXPOSE 12880-12890

# This installs the CCU Python library on your container.
# Replace username and apikey with your data, but your API key will be
# visible in your git repo.  The documentation on "Updating an Existing Dockerfile"
# tells you how to do this using secrets.
RUN pip install https://username:apikey@artifactory.sri.com/artifactory/cirano-pypi-local/c

# This next block should install your software.  Here that is just one
# COPY command, but you might need to copy a lot more or run various
# "pip install" or other commands.

# Note: Everything you copy onto your container must start out in this
# directory (or a subdirectory) in your host machine.  You can not do
# things like COPY ../somewhere-else /app.  You can only copy from this
# directory, down.

COPY ta1example.py /app

# This command starts up your analysis program.  Once started, it
# should wait for messages which contains the data it needs to run
# its analysis, and post its results to the appropriate queue.
# Logging should be sent to standard output, or written to a file
# in /var/log, or both.

CMD python -u ta1example.py
```

Or this for a Java based analytic:

```
# This is a sample Dockerfile for creating containers for the CCU System.

# Using a *-slim base container is important because it
# results in much smaller containers.  I am starting off
# with Python 3.10, but you can start with some different.
# If you really need a "full" Linux installation, such as
# Ubuntu, use this: FROM ubuntu:20.04 or whatever base image
# you want.

FROM openjdk:17-alpine

# This block is housekeeping, just creates a directory to put
```

```
# everything in one place.

RUN mkdir /app
WORKDIR /app

# These are the ports used by the zmq sockets.

EXPOSE 11880-11890
EXPOSE 12880-12890

# This installs the CCU Python library on your container.
# This command will change in the future, but for now, the "wheel"
# file must be in the local directory.

COPY jeromq-0.5.2.jar json-20220320.jar /app
COPY ccu-1.0.jar /app

# This next block should install your software.  Here that is just one
# COPY command, but you might need to copy a lot more or run various
# "pip install" or other commands.

COPY Sub.class /app

# This command starts up your analysis program.  Once started, it
# should wait for messages which contains the data it needs to run
# its analysis, and post its results to the appropriate queue.
# Logging should be sent to standard output, or written to a file
# in /var/log, or both.

CMD java -cp .:ccu.jar:jeromq-0.5.2.jar:json-20220320.jar Sub
```

Build A Docker Image

Run the following command to build an image:

```
docker build -t sri-youranalytic:version -t sri-youranalytic:latest -f Dockerfile .
docker image list
```

- Name your container using an abbreviation for your company and then the containers name.
- Use two different -t options, one for the version of your container, and the other for latest. Latest will not be added by default if you give a version, and you must give a version so we can track it.
- You can use any versioning scheme you want, just so long as the containers you give us have "bigger" versions as they are updated.
- The -f Dockerfile is optional because "Dockerfile" is the default.
- All files you COPY to the container in the Dockerfile must be in the local directory.

Create a *.yaml Configuration File

In addition, you should have a YAML file named sri-ta1example.yaml (if your container is called ta1example and your are part of SRI). See the example.yaml and example_big.yaml files to get started. Here is an example:

```
# Because this is an SRI example, the name starts with sri-.  If you are at PARC
# or LCC it would start with parc- or lcc- and so on.
name: sri-example
version: 0.3
poc:
```

```
    email: joshua.levy@sri.com
    name: Joshua Levy
testingInstructions: |
    Feeding example-test1.jsonl into the container will cause it to detect faces
    in the image and generate the following outputs: age and sex of each person
    detected; location/scene information of the image; total number of people in
    the scene; gaze information.  The example-out1.jsonl contains the result of
    feeding example-test1.jsonl into it.

# The docker Compose configuration code used by your container.  The example
# below is for an image which only uses a CPU.
dockerCompose:
    image: cirano-docker.cse.sri.com/sri-ta1example
    network_mode: "host"
    volumes:
        - "${CCU_SANDBOX}:/sandbox"

# List of required resources.
# * CPU means only uses CPU (and this is the default if nothing is specified)
# * GPU-CPU means it will use a GPU if one is available at runtime, or CPU if not.
# * GPU means it needs a GPU at runtime.
# * Internet means it accesses the internet at runtime.

resources:
    - GPU
    - Internet

# List of behaviors.  If there are none, this section does not need to exist at all.
# * Slow-Start means the container takes longer than 10 seconds to start up.
# * Not-Realtime means the container takes noticably longer to run than the media it
#   is given.  For audio and video media.

behaviors:
    - Slow-Start

# Inputs and Outputs are lists of queues, each has a list of message types read
# or written in that queue.  Do not further describe each message type,  as this
# is done on the "Messages and Queue" page on Confluence.
inputs:
    - AUDIO_ENV
        - asr_result
outputs:
    - RESULT
        - age
        - sex
        - face_bbox
        - location-scene
        - people_count
        - gaze
```

Finally, you should include one or more JSONL files (used for testing) and matching files showing the results. For example if you have a file sri-ta1example-test1.jsonl then you should have a sri-ta1example-out1.jsonl file with the results of running that test.

Note that YAML has an alternate syntax where lists like resources above can be represented like this, if you prefer:

```
resources: [GPU, Internet]
```

Please use four character indents in YAML files.

Setting Up A CCU Environment / Running and Troubleshooting CCU Containers

Troubleshooting a CCU container means running just the one container you are working on. So your analyic will be run as a container, but the processes which feed it data and log what it does will be python programs running on the host machine. This is the lightest footprint, quickest test cycle way to go.

To troubleshoot CCU containers, we need to do some setup, and then run four processes in seperate shells, terminal, or windows. Each of these steps is described below.

This assumes everything is running on one machine, but you can run different parts on different machines. See "Multiple Machine Environments" in ReferenceDocument.pdf for details.

Setting Up A CCU Environment

Set up the sandbox directory.

Before running containers you need to set up a sandbox directory. You can do this be creating a directory called sandbox, anywhere on your machine and then copying files from the samples/sandbox directory in the distribution to your newly created directory. Then set a shell variable to the full path name of your sandbox directory:

Use commands like this on Linux:

```
cd where-i-do-ccu-development
mkdir sandbox
# This command copies the contents of the sample sandbox directory to your new
# sandbox directory.
cp .../sample/sandbox/* sandbox
export CCU_SANDBOX=`pwd`/sandbox
```

Use commands like this on Windows (cmd window):

```
cd where-i-do-ccu-development
mkdir sandbox
: This command copies the contents of the sample sandbox directory to your new
: sandbox directory.
copy ...\sample\sandbox\* sandbox
set CCU_SANDBOX=%CD%\sandbox
```

On Windows (Powershell window) replace the last command with this:

```
$env:CCU_SANDBOX = ...full path of sandbox...
```

You may want to add the export/set command to to .bashrc file or to scripts you have to setup your CCU work environment.

Update the configuration file.

In the main directory is a ccu-config.yaml file. Copy this file to your sandbox directory and then: 1. Edit it to include the IP address of your host machine. Do not use 127.0.0.1! This will not work from the containers that you are going to run. Find the local IP address of your host machine. On Linux run `ip addr`. On windows, run `ipconfig`.
Or run this: `python -c "import socket; print(f'IP is {socket.gethostbyname(socket.gethostname())`

2. If you don't want debugging information printed out, change the "debug" attribute to "False".

Start ccuhub.py

Whenever you are running CCU containers, you must run the ccuhub.py program. It creates the run time environment for CCU to operate in.

In a seperate shell, window, or terminal, run this command:

```
python -u ccuhub.py
```

You should see something like this:

```
2022-05-19 04:05:44 INFO CCU started in debugging mode.  Level is DEBUG.  IP is 172.20.32.1
2022-05-19 04:05:44 INFO CCU Hub running for version 1.0 on 172.20.32.1.
2022-05-19 04:05:44 INFO with config file C:\Users\e33173\CCU\samples\sandbox\ccu-config.ya
```

Check that the IP addresses are what you expect. They should both be the same. If they actually say "172.20.32.1" you need to update the ccu-config.yaml file. That is my IP address, not your's.

There is also a ccuhub.java program in the java directory.

Start logger.py

The logger.py process prints out all messages that are passed between CCU containers, the Hololens, etc. In a sense, it is the ultimate debugging and monitoring tool, since it shows all possible input messages to your container and the all the output messages created by your container.

In a seperate shell, window, or terminal, run this command:

```
python -u logger.py
```

You should see something like this:

```
2022-05-19 04:17:30 INFO Logger Running
2022-05-19 04:17:30 INFO CCU started in debugging mode.  Level is DEBUG.  IP is 172.20.32.1
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12880
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12881
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12882
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12883
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12884
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12885
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12886
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12887
2022-05-19 04:17:30 DEBUG CCU connecting to tcp://172.20.32.1:12888
```

Make sure the IP maches the ccuhub IP number.

Testing Your CCU Environment

Before testing your CCU environment, you might want to test it, without your analytic. To do this, run the `python -u script.py --jsonl sample-data/sample-each.jsonl` command. You should see output from logger.py. If you do, then the infrastructure is working. You can run this command for any of the jsonl files in the sample-data directory.

Start Your CCU Container

After following the instructions above on "Changing Your Python Code" and "Building A Container", you should have a container you can run with commands like these on Linux:

```
export CCU_SANDBOX=`pwd`/sandbox
docker run -it --rm --publish-all --volume $CCU_SANDBOX:/sandbox sri-youranalytic
docker container list
```

Or like this on Windows cmd:

```
set CCU_SANDBOX=%CWD%/sandbox
docker run -it --rm --publish-all --volume %CCU_SANDBOX%:/sandbox sri-youranalytic
docker container list
```

- The export command only needs to be run once per shell/windows.
- The -it flags let you stop the container with control-c or control-break (on windows).
- The –rm makes it easier to stop and rm containers.
- The –publish-all is important for container to container communications which comes later.

If you don't want to set the CCU_SANDBOX variable, then you can put in the full path directly. For example, the following command will work on Windows Powershell if the sandbox directory in in the current working directory:

```
docker run -it --rm --publish-all --volume ${PWD}/sandbox:/sandbox youranalytic
```

Troubleshooting The Container

Running the first command below will give you a shell running on your container. The other commands can help you troubleshoot:

```
docker run -it --rm --publish-all sri-youranalytic /bin/bash
python --version
ls
pip list
```

Stop A Docker Container

Get the container name with the first command. It is the first field on the line. Then stop the container with the second command:

```
docker container list
docker stop container-name
```

Feeding Messages Into Your Containers

Now, you need a way to feed data into your container. This might be the most complex part of the process. It will differ depending on the messages that your container is expected to react to.

Option 1: Run script.py

The script.py program is a general "message injection" tool. It runs *.jsonl files (which are lists of messages) and then sends those messages to analytic containers via CCU queues. The command to do this is:

```
python -u script.py --jsonl jsonl-file
```

For example:

```
python -u script.py --jsonl ../sample-data/sample-each.jsonl
```

This command is described in much more detail in README-Utilities in the python directory.

Option 2: Run Hololens

If you have a hololens and the CCU software running on it, you can run that to generate messages.

Remember to configure your hololens to use the host IP address of the machine which is hosting all of the CCU containers.

Packaging and Distributing CCU Images

There are several steps in distributing a CCU Image. For this example, I'm assumine that the image name is "sri-example" in the command below:

1. Check you have what you need.

- A CCU image with the right name "insitution-name" (such as: sri-example), which is tagged with both "latest" (in lowercase) and a version number.
- A *.yaml file with the same name as the CCU Image, and this* .yaml file must be checked with a command like this: `python yamlcheck.py sri-example.yaml`
- A collection of *.jsonl files in input, output pairs. Each pair tests the CCU image and contains the input required for the test and the output expected from the test. All of these* .jsonl files should be tested with commands like `python jsonlcheck.py test1-input.jsonl`

2. Make sure all supplimental files are in one directory and then tar (or zip) it up.

Create a directory with the same name as the image, and in that directory place the *.yaml config file and all the input and output* .jsonl files used for testing. You may also put other files in there that might be helpful for integration (or for running your container).

Put the whole directory into one zip, tar, or tar.gz file.

3. Push the container and the *.yaml file to Artifactory

To push the CCU image, run a command like this:

```
docker push cirano-docker.cse.sri.com/sri-example:1.0.0
```

To push the suppliemental files, run a command like this:

```
curl -H 'X-JFrog-Art-Api:API_KEY' -T sri-example.tar "https://artifactory.sri.com/artifact
```

API_KEY is your Artifactory API_KEY, which you can see on your account page on artifactory.sri.com.
Note that the file name does not contain the version number. When you push a new version we will loose the older version. Please keep a copy of your older versions yourself if you want to have them available.

4. Send email.

Although not strictly required, please do send email to joshua.levy@sri.com and lizchen.qui@monash.edu so each TA2 team will know a newer image is available.

Older Instructions

This varies slightly based on what machine you are on, but the basic commands are these:

```
docker save -o sri-ta1example.tar sri-ta1example
ls -l sri-ta1example.tar
tar -tf sri-ta1example.tar
gzip sri-ta1example.tar
ls -l sri-ta1example.tar.gz
```

Creating Jsonl Files

Creating jsonl files is important for testing your code, and so that the integration team can test your containers before integrating them. There are several different ways to do this:

Use Source2stream4ccu Software (for LDC Data or Web URLs)

If your container reacts to audio and/or video, then you can use software written by Michael Youngblood at PARC to convert it into messages. This software includes an installation script and README file and is available here:

https://gitlab.com/public-maksimus/source2stream4ccu

Copy and Modify Existing Jsonl Files (for String Heavy Messages)

If your container reacts to messages which are short and based on text or simple data, then it is pretty easy to copy and edit the existing Jsonl files to create what you want. The sample-data directory contains jsonl files that you can start with, and the sample-each.jsonl file contains a lot of different messages you can clone.

Use Other Software

For more complex messages, there are utility programs that create jsonl files out of other data like audio or images. All of these programs are documented in more detail in the README-Utilities.pdf file, or you can run them with the –help option. These command generate jsonl directly (so you don't need to use ccuhub.py or logger.py):

- If you want to create jsonl files which have translated spoken words in them, use the type2jsonl.py program to create them.
- If you want to create jsonl file which have "video" created from one image, or a small number of images, you can use the image2jsonl.py program to create them.
  `python image2jsonl.py -o headshot.jsonl headshot.jpg`
- If you have wav or mp3 files, you can use the program send_audio.py to convert them to messages.
- The capture_audio_env.py create jsonl or wav files based on a computer's speaker, so you can use it to record environmental sounds.

Using the Hololens

Finally, if your container needs audio or video input, then you can record a jsonl file from the Hololens, in two different ways. The simplest is to just record the Hololens messages using the logger.py utility with a pipeline like this:
`Hololens -> messages -> logger.py -> jsonl file`

If you only want the audio, you can use write_audio.py to record Hololens audio to jsonl, wav, and other file formats.

The README-Utilities.pdf file contains details on the format of Jsonl files in general, and the Messages.pdf file contains details on each message (which are inside the Jsonl files).

How Many Jsonl Files? How Much Testing?

Note: this discussion covers the "smoke testing" that the integration testing team does prior to integration testing. It does not cover the testing you do to make sure your container works.

The goal of your testing is simple: if I test your container and it works, then I should be able to run it in a demo/testing environment, and it should run there, also.

The metric I care about is Mean Time To Failure. If, after testing your container, I use it in a demo, and it fails quickly, that is bad. If I find some bug after a day or two, that is not nearly so bad.

This gets into a long and complex discussion about how much testing is the right amount and in what situations. My basic answer is this:

- All tests should be documented in your YAML file, no matter how many there are.
- All of your tests should run in 5-10 minutes or less.
- At least one test should run "real data" or as close to "real data" as possible.
- At least one test should run badly formed data, and therefore give an error message, not good results.
- If your container works with many data formats, test each one at least once.

- If it generates different types of results, make sure it generates them all sometime during testing.
- If your container works with either a CPU or a GPU, then test both.

Troubleshooting

Processes/Containers Run But No Messages Are Seen

Everything starts up just fine, but no messages seem to be sent or recieved.

The most common cause for this is not using the right IP address. All programs and containers should print out the IP address they are using (and often the ports as well) when they start up. Read the IP Addresses section of the ReferenceDocument for more information.