CDX Documentation

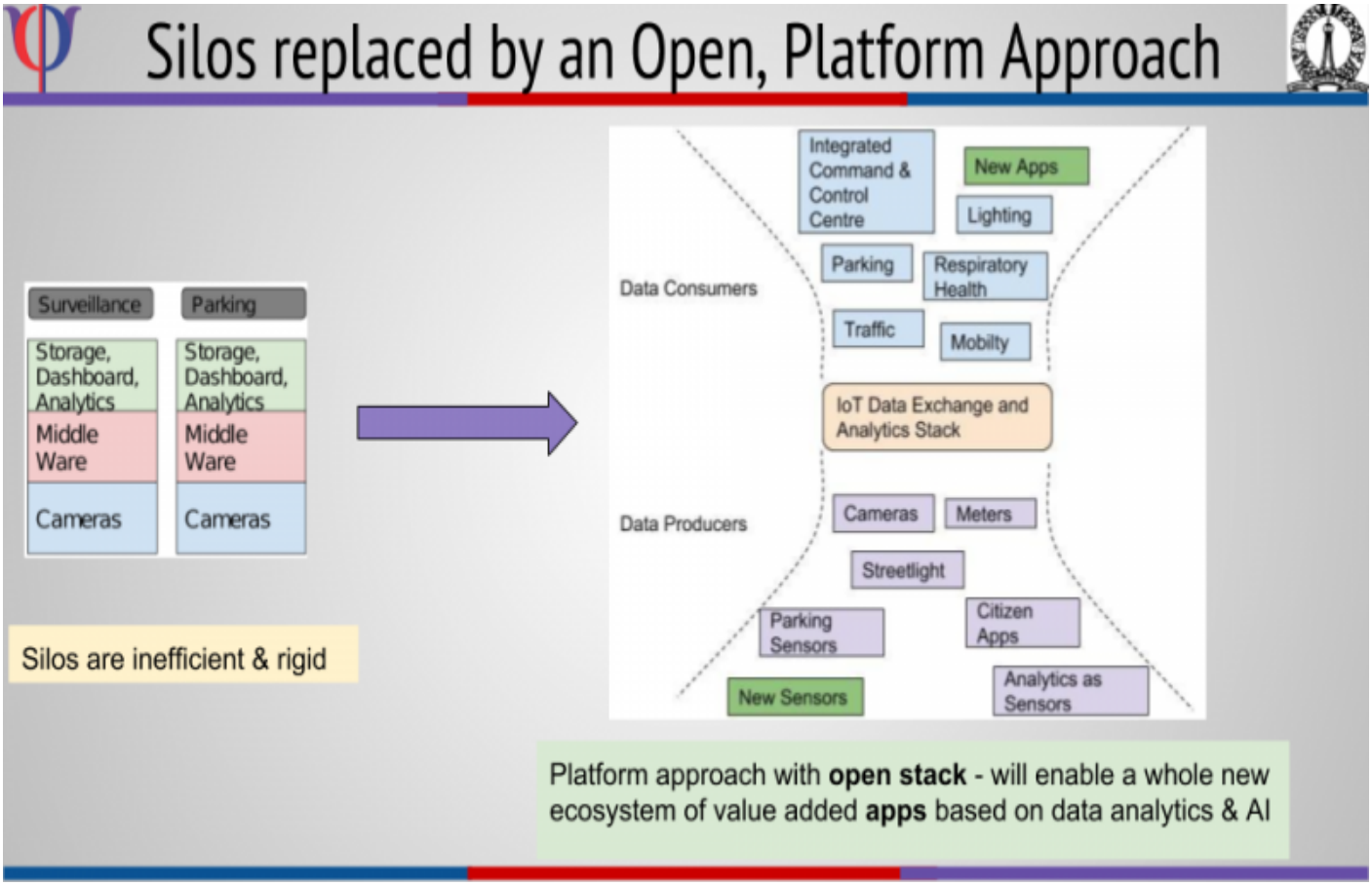
Release 1.0.0

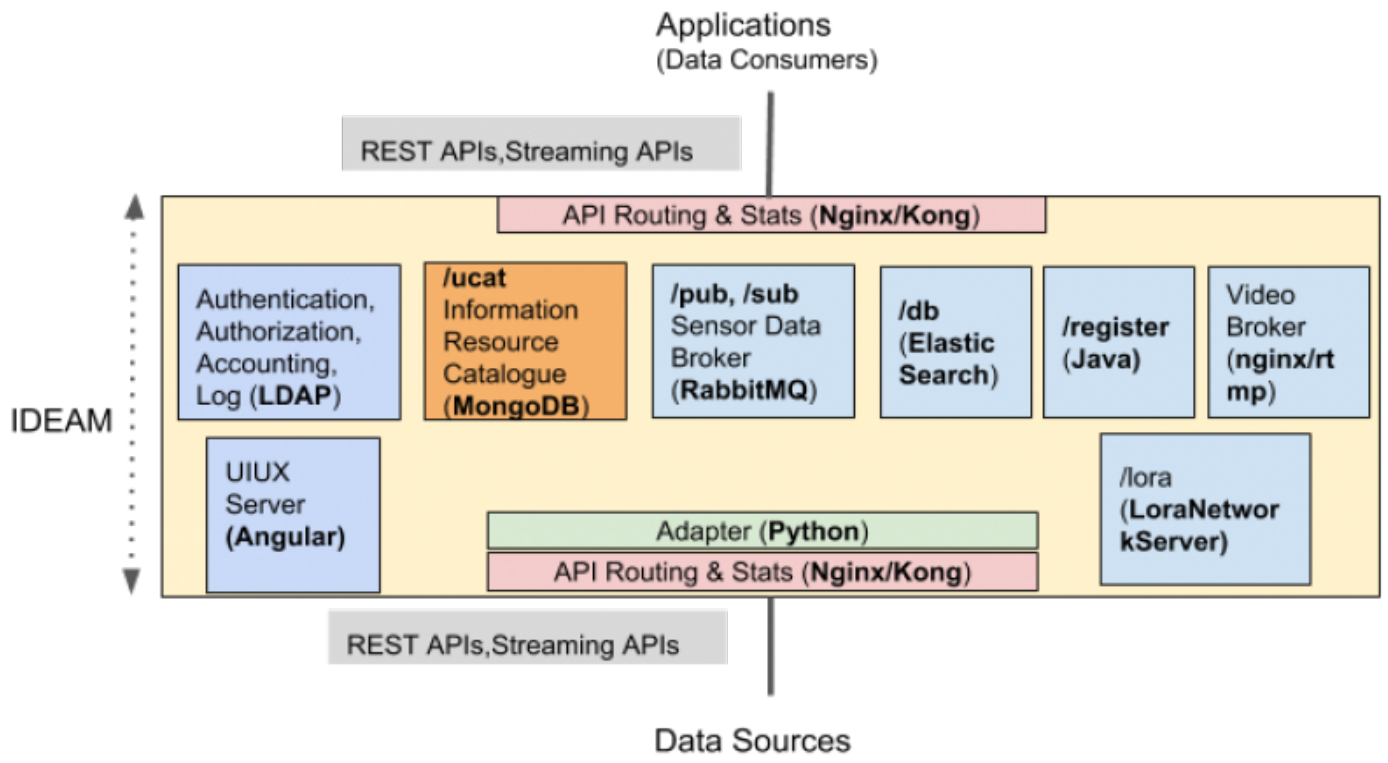
CDX Editor

Sep 13, 2018

Contents

City Data Exchange (CDX) is a high performance, scalable and secure open source IOT platform deployed in docker. It is a comprehensive software solution stack comprising various microservices for enabling seamless data exchange between data producers and consumers.




Applications
(Data Consumers)
REST APIs,Streaming APIs
API Routing & Stats (Nginx/Kong)
IDEAM
Authentication, Authorization, Accounting, Log (LDAP)
/ucat Information Resource Catalogue (MongoDB)
/pub, /sub Sensor Data Broker (RabbitMQ)
/db (Elastic Search)
/register (Java)
Video Broker (nginx/rtmp)
UIUX Server (Angular)
/lora (LoraNetworkServer)
Adapter (Python)
API Routing & Stats (Nginx/Kong)
REST APIs,Streaming APIs
Data Sources

CHAPTER 1

Features

- Supports multiple protocols: HTTP, AMQP and MQTT.
- Has a solid access control framework for data exchange.
- Supports time based data lease options.

CHAPTER 2

Contents:

2.1 Getting Started

This guide will help you to quickly get started with a single node instance of CDX and start publishing data from a device.

2.1.1 Prerequisites

1. Install Docker

```
sudo apt-get install docker.io
```

2. Add permission to user

```
sudo usermod -a -G docker $USER
```

3. Logout and log back in.
4. Run `ifconfig` and note down the name of the interface you're connected to.

```
enp9s0    Link encap:Ethernet  HWaddr f0:76:1c:1e:f0:0e
          inet addr:192.168.10.89  Bcast:192.168.10.255  Mask:255.255.255.0
          inet6 addr: fe80::3598:b26d:8d1e:fbbf/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:868637 errors:0 dropped:0 overruns:0 frame:0
          TX packets:298439 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:354469977 (354.4 MB)  TX bytes:47315921 (47.3 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:7276 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7276 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:10912131 (10.9 MB)  TX bytes:10912131 (10.9 MB)

wlp8s0    Link encap:Ethernet  HWaddr c0:38:96:0d:7c:25
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

In the above example the interface name is `enp9s0`

5. Now run:

```
sudo nmcli device show enp9s0
```

Replace `enp9s0` with your respective interface name. The output should be something like this

```
GENERAL.DEVICE:                         enp9s0
GENERAL.TYPE:                           ethernet
GENERAL.HWADDR:                         F0:76:1C:1E:F0:0E
GENERAL.MTU:                            1500
GENERAL.STATE:                          100 (connected)
GENERAL.CONNECTION:                     IISc_Network
GENERAL.CON-PATH:                       /org/freedesktop/NetworkManager/ActiveConnection/
WIRED-PROPERTIES.CARRIER:               on
IP4.ADDRESS[1]:                         192.168.10.89/24
IP4.GATEWAY:                            192.168.10.1
IP4.ROUTE[1]:                           dst = 169.254.0.0/16, nh = 0.0.0.0, mt = 1000
IP4.DNS[1]:                             8.8.8.8
IP4.DNS[2]:                             168.95.1.1
IP6.ADDRESS[1]:                         fe80::3598:b26d:8d1e:fbbf/64
IP6.GATEWAY:
```

Note down the DNS from `IP4.DNS` section

6. Add DNS obtained in the previous step to the file `/etc/docker/daemon.json` If the file does not exist, create it. The entry should look something like this

```
{"dns": ["8.8.8.8", "8.8.4.4"]}
```

Add the obtained DNS to the above JSON array. So the final entry should look something like this:

```
{"dns": ["8.8.8.8","8.8.4.4","168.95.1.1"]}
```

Of course, the DNS would change according to your network.

7. Add DNS in `/etc/default/docker` file as follows:

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 --dns 168.95.1.1"
```

8. Restart Docker

```
service docker restart
```

2.1.2 CDX Installation

1. Clone CDX git repo

```
git clone https://github.com/rbccps-iisc/ideam.git
```

CDX repository comes with a default configuration file `ideam.conf`:

```
[APIGATEWAY]
https = 8443

[BROKER]
http = 12080
management = 12081
amqp = 12082
mqtt = 12083

[ELASTICSEARCH]
kibana = 13081

[WEBSERVER]
http = 14080

[LDAP]
ldap = 15389

[CATALOGUE]
http = 16080

[KONGA]
http = 17080

[VIDEOSERVER]
rtmp = 18935
hls = 18080
http = 18088

[PASSWORDS]
ldap = ?
broker = ?
cdx.admin = ?
database = ?
```

This file contains details about the ports used by different microservices. It also allows the user to configure passwords that should be used for certain services during installation. By default, the password fields in the config file is set to `?`, which indicates that the system will generate random passwords during runtime.

2. Install CDX

```
cd ideam/
./install
```

2.1.3 Registering your first device

- Once CDX has installed you can now start registering devices with it. Let's create a simple test device for the sake of illustration:

```
sh tests/create_entity.sh testStreetlight
```

- This will give you the details of the registration

```
{
  "Registration": "success",
  "entityID": "teststreetlight",
  "apiKey": "EHQilai5cF_tNmWOwg-oiPdncmRPdfGCIhFHM85zDDW",
  "subscriptionEndPoint": "https://smartcity.rbccps.org/api/{version}/
↪followid=teststreetlight",
  "accessEndPoint": "https://smartcity.rbccps.org/api/{version}/db?
↪id=teststreetlight",
  "publicationEndPoint": "https://smartcity.rbccps.org/api/{version}/publish?
↪id=teststreetlight",
  "resourceAPIInfo": "https://rbccps-iisc.github.io"
}
```

2.1.4 Publishing from your device

- You can now publish data from this device using:

```
sh tests/publish.sh teststreetlight EHQilai5cF_tNmWOwg-oiPdncmRPdfGCIhFHM85zDDW
```

  This will publish `{"body": "testdata"}` to the exchange `teststreetlight.protected`

- That's it! You can similarly register more devices and apps with the middleware.

2.2 Guide

This guide will help you understand the internals and operations of the CDX system.

2.2.1 CDX Configuration

Once we are done with setting up CDX, the configuration file `ideam.conf` will updated with the generated passwords for sub-systems

```
[APIGATEWAY]
https = 8443

[BROKER]
http = 12080
management = 12081
amqp = 12082
mqtt = 12083

[ELASTICSEARCH]
elastic = 9200
kibana = 13081

[WEBSERVER]
http = 14080
```

(continues on next page)

(continued from previous page)

```
[LDAP]
ldap = 15389

[CATALOGUE]
http = 16080

[KONGA]
http = 17080

[VIDEOSERVER]
rtmp = 18935
hls = 18080
http = 18088

[PASSWORDS]
ldap = 721UD9ytc1Qc4ORT
broker = 0Rv7MxG2uLsB2bgq
cdx.admin = 0BsZmPezYrjbqS2CmYjtiP6ZfJfoKg4k
database = 9SWhSOyVHHpIBrYDTI613o0YdCclVXc0MlG75y1VGfx
```

2.2.2 CDX Subsystems

As explained earlier, CDX consists of multiple subsystems such as APIGateway, Data Broker, Media Broker, IoT Database, Authentication, Authorization and Accounting system etc.

In this section, we will have detailed usage guide for every subsystem. Let us first understand if all the subsystems are live and operating in the respective ports as per the config file provided. This can be verified by using the docker command

```
docker ps
```

Which gives us the list of live containers running in the system as follows

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED
↪ STATUS              PORTS
↪                                            NAMES
18e3864c3a46        pantsel/konga          "/app/start.sh"          18 minutes ago
↪ Up 17 minutes       127.0.0.1:17080->1337/tcp
↪                                            konga
80a390e62081        ideam/videoserver      "/usr/sbin/sshd -D"      18 minutes ago
↪ Up 18 minutes       22/tcp, 0.0.0.0:18935->1935/tcp, 0.0.0.0:18080->8080/tcp, 0.0.
↪0.0:18088->8088/tcp                         videoserver
8ed4e4b96ef6        ideam/ldapd            "/usr/sbin/sshd -D"      18 minutes ago
↪ Up 18 minutes       22/tcp, 127.0.0.1:15389->8389/tcp
↪                                            ldapd
67f990e9213c        ideam/webserver        "/bin/sh -c 'tail -f..."   18 minutes ago
↪    Up 18 minutes       127.0.0.1:14080->8080/tcp
↪                                              webserver
47e9730f4184        ideam/elasticsearch    "/usr/sbin/sshd -D"      18 minutes ago
↪ Up 18 minutes       22/tcp, 127.0.0.1:9200->9200/tcp, 127.0.0.1:13081->5601/tcp
↪                                            elasticsearch
a3099a939d50        ideam/broker           "/bin/sh -c 'tail -f..."   18 minutes ago
↪    Up 18 minutes       0.0.0.0:12083->1883/tcp, 0.0.0.0:12082->5672/tcp, 127.0.0.
↪1:12080->8000/tcp, 127.0.0.1:12081->15672/tcp   broker
98af3aa5c65f        ideam/catalogue        "/root/run.sh /usr/s..."   19 minutes ago
↪    Up 18 minutes       22/tcp, 27017/tcp, 28017/tcp, 127.0.0.1:16080->8000/tcp
↪                                            catalogue
```

(continues on next page)

(continued from previous page)

```
306fd29f57c3        ideam/apigateway      "/docker-entrypoint...."   19 minutes ago  ␣
↪   Up 19 minutes       8000-8001/tcp, 8444/tcp, 0.0.0.0:8443->8443/tcp             ␣
↪                                          apigateway
```

Each container is mapped to a specific port of the system to expose the APIs and services offered. If you look at the PORTS column, we have `PORT 8443` of apigateway container mapped to `0.0.0.0:8443`. This exposes all the APIs offered by CDX to be accessed from external system. If we look at webserver, `PORT 8080` is mapped to `127.0.0.1:14080`. This exposes the services offered by the webserver to be accessed only from CDX system.

In order to login to the specific container to view logs or for debugging, CDX provides shell access. The scripts for it is available in `ideam/shells`

```
ls shells/
apigateway*  broker*  catalogue*  elasticsearch*  ldapd*  videoserver*  webserver*
```

For example, to login into webserver, you can execute the following

```
shells/webserver
```

Now you are in webserver container. To exit the container use `CTRL + D`

If you have already registered the test device as explained in the Getting Started section you can skip this step. If not, let us now register a device and observe the sequence of events that happen in each subsystem.

To register a device run the `create_entity.sh` script `tests` directory with the ID `testStreetlight`:

```
sh tests/create_entity.sh testStreetlight
```

In the following subsections, we will observe the impact of registering `testStreetlight`

API Gateway

Broker

Catalog

Database

AAA Server

Web Server

2.3 API Documentation

This is the documentation for all the APIs

2.3.1 Register

API Specification

- **Summary**: The register API is used to register devices and apps with the middleware. They can register as either devices or subscribers, the difference being that subscribers do not publish any data. Every device can

have its own schema, or in other words, meta information about the nature of the information that the device publishes. For more information on schemas visit here

- **Endpoint**: `https://localhost:8443/api/1.0.0/register`
- **Method**: `POST`
- **Required Headers**:

Header Name	Description
apikey	API key of the provider

- **Optional Headers**:

Header Name	Description
security_level	An integer between 1-5

- **Body**: A valid catalogue item as mentioned here
- **Example request**:

```
curl -X POST \
  http://localhost/api/1.0.0/register \
  -H 'apikey: guest' \
  -H 'content-type: application/json' \
  -k -d '{

 "entitySchema": {
 "refCatalogueSchema": "generic_iotdevice_schema.json",
 "resourceType": "streetlight",
 "tags": [
   "onstreet",
   "Energy",
   "still under development!"
 ],
 "refCatalogueSchemaRelease": "0.1.0",
 "latitude": {
   "value": 13.0143335,
   "ontologyRef": "http:\/\/www.w3.org\/2003\/01\/geo\/wgs84_pos#"
 },
 "longitude": {
   "value": 77.5678424,
   "ontologyRef": "http:\/\/www.w3.org\/2003\/01\/geo\/wgs84_pos#"
 },
 "owner": {
   "name": "IISC",
   "website": "http:\/\/www.iisc.ac.in"
 },
 "provider": {
   "name": "Robert Bosch Centre for Cyber Physical Systems, IISc",
   "website": "http:\/\/rbccps.org"
 },
 "geoLocation": {
   "address": "80 ft Road, Bangalore, 560012"
 },
 "data_schema": {
   "type": "object",
```

(continues on next page)

(continued from previous page)

```
  "properties": {
    "dataSamplingInstant": {
      "type": "number",
      "description": "Sampling Time in EPOCH format",
      "units": "seconds",
      "permissions": "read",
      "accessModifier": "public"
    },
    "caseTemperature": {
      "type": "number",
      "description": "Temperature of the device casing",
      "units": "degreeCelsius",
      "permissions": "read",
      "accessModifier": "public"
    },
    "powerConsumption": {
      "type": "number",
      "description": "Power consumption of the device",
      "units": "watts",
      "permissions": "read",
      "accessModifier": "public"
    },
    "luxOutput": {
      "type": "number",
      "description": "lux output of LED measured at LED",
      "units": "lux",
      "permissions": "read",
      "accessModifier": "public"
    },
    "ambientLux": {
      "type": "number",
      "description": "lux value of ambient",
      "units": "lux",
      "permissions": "read",
      "accessModifier": "public"
    },
    "targetPowerState": {
      "type": "string",
      "enum": [
        "ON",
        "OFF"
      ],
      "units": "dimensionless",
      "description": "If set to ON, turns ON the device. If OFF turns OFF the␣
↪device. Writeable parameter. Writeable only allowed for authorized apps",
      "permissions": "read-write",
      "accessModifier": "protected"
    },
    "targetBrightnessLevel": {
      "type": "number",
      "description": "Number between 0 to 100 to indicate the percentage␣
↪brightness level. Writeable only allowed for authorized apps",
      "units": "percent",
      "permissions": "read-write",
      "accessModifier": "protected"
    },
    "targetControlPolicy": {
```

(continues on next page)

(continued from previous page)

```
      "enum": [
        "AUTO_TIMER",
        "AUTO_LUX",
        "MANUAL"
      ],
      "units": "dimensionless",
      "permissions": "read-write",
      "description": "Indicates which of the behaviours the device should␣
↪implement. AUTO_TIMER is timer based, AUTO_LUX uses ambient light and MANUAL is␣
↪controlled by app. Writeable only allowed for authorized apps",
      "accessModifier": "protected"
    },
    "targetAutoTimerParams": {
      "type": "object",
      "permissions": "read-write",
      "properties": {
        "targetOnTime": {
          "type": "number",
          "description": "Indicates time of day in seconds from 12 midnight when␣
↪device turns ON in AUTO_TIMER. Writeable only allowed for authorized apps",
          "units": "seconds",
          "accessModifier": "protected"
        },
        "targetOffTime": {
          "type": "number",
          "description": "Indicates time of day in seconds from 12 midnight when␣
↪device turns OFF in AUTO_TIMER. Writeable only allowed for authorized apps",
          "units": "seconds",
          "accessModifier": "protected"
        }
      }
    },
    "targetAutoLuxParams": {
      "type": "object",
      "permissions": "read-write",
      "properties": {
        "targetOnLux": {
          "type": "number",
          "description": "Indicates ambient lux when device turns ON in AUTO_LUX.
↪ Writeable only allowed for authorized apps",
          "units": "lux",
          "accessModifier": "protected"
        },
        "targetOffLux": {
          "type": "number",
          "description": "Indicates ambient lux when device turns OFF in AUTO_
↪LUX. Writeable only allowed for authorized apps",
          "units": "lux",
          "accessModifier": "protected"
        }
      }
    }
  },
  "additionalProperties": false
},
"serialization_from_device": {
  "format": "protocol-buffers",
```

(continues on next page)

(continued from previous page)

```
   "schema_ref": {
     "type": "proto 2",
     "mainMessageName": "sensor_values",
     "link": "https:\/\/raw.githubusercontent.com\/rbccps-iisc\/applications-
↪streetlight\/master\/proto_stm\/txmsg\/sensed.proto"
   }
 },
 "serialization_to_device": {
   "format": "protocol-buffers",
   "schema_ref": {
     "type": "proto 2",
     "mainMessageName": "targetConfigurations",
     "link": "https:\/\/raw.githubusercontent.com\/rbccps-iisc\/applications-
↪streetlight\/master\/proto_stm\/rxmsg\/actuated.proto"
   }
 },
 "id": "streetlight"
}}'
```

API key

The API key is a globally unique identifier that is tied specifically to a device. It is non-temporal and used to identify one particular device *only* unless otherwise needed. (See <Group API key>) Once a device is registered, all further communications with the middleware from the device must use the API key.

API key can be requested in five security levels. They are as follows:

1. **Level 1**: 64 bit
2. **Level 2**: 128 bit
3. **Level 3**: 256 bit
4. **Level 4**: 512 bit
5. **Level 5**: 1024 bit

Depending upon the criticality of the device in the infrastructure, the desired security level can be requested for. The default level in the absence of an explicit header mentioning it, will be 3 i.e. 256 bit security.

Success Response

- **Code**: `200 OK`
- **Response Body**

```
{
 "Registration": "success",
 "entityID": "streetlight",
 "apiKey": "MZW68JGDCbxFVTUndReIUTNIC5cEIS3Glho14XZfZhp",
 "subscriptionEndPoint": "https://smartcity.rbccps.org/api/{version}/follow?
↪id=streetlight",
 "accessEndPoint": "https://smartcity.rbccps.org/api/{version}/db?id=streetlight",
 "publicationEndPoint": "https://smartcity.rbccps.org/api/{version}/publish?
↪id=streetlight",
 "resourceAPIInfo": "https://rbccps-iisc.github.io"
}
```

This API key will be used to uniquely identify this device. It is recommended that the obtained key is "burnt" into the device after registration.

Error Responses

1. **Problem**: If the provider key is wrong

```
{
 "message": "Invalid authentication credentials"
}
```

Solution: Use a valid provider API key (Refer <Konga>)

2. **Problem**: If the schema does not conform to any of the base schemas or if the catalogue server refuses to accept the schema in its present form

```
{
 "Registration": "failure",
 "Reason": "uCat update Failure"
}
```

Solution: Please revisit the link on how to construct meaningful schemas.

3. **Problem**: If the ID you are using has already been used previously

```
{
 "Registration": "failure",
 "Reason": "ID already used. Please Use a Unique ID for Registration."
}
```

Solution: Try using another ID for the device and register again

4. **Problem**: If the schema is not a valid JSON

```
{
 "Registration": "Failure",
 "Reason": "JSON parse error"
}
```

Solution: Use a tool like JSONLint to validate your JSON

5. **Problem**: If the security level used is outside the valid range

```
{
 "Registration": "Failure",
 "Reason": "Security level must be between 1-5"
}
```

Solution: Use a security level between 1-5

2.3.2 Publish

- **Summary**: The publish API is used to publish data from devices to the middleware. To access this API, the device API key obtained after device registration must be used. Published messages go into the specified exchange of the device. There are 6 exchanges and 4 queues assigned to a device. They are as follows:

Exchanges	Queues
<device_name>.configure	<device_name>
<device_name>.follow	<device_name>.configure
<device_name>.notify	<device_name>.follow
<device_name>.public	<device_name>.notify
<device_name>.protected	<device_name>.priority
<device_name>.private	

Some of the exchanges are pre-bound to queues for convenience. They are as follows:

Exchange	Bound Queue
<device_name>.configure	<device_name>.configure
<device_name>.follow	<device_name>.follow
<device_name>.notify	<device_name>.notify

This is just a default setting and can be changed anytime by the device administrator.

- **Endpoint**: `https://localhost:8443/api/1.0.0/publish/<exchange_name>` Here <exchange_name> specifies the exchange to which the message must be published to.
- **Method**: `POST`
- **Required Headers**:

Header Name	Description
apikey	API key of the device

- **Optional Headers**:

Header Name	Description
routingKey	Topic to which the published message belongs

- **Body**: The message from the device that conforms to the schema specified during registration
- **Example Request**:

```
curl -X POST \
https://localhost:8443/api/1.0.0/publish/streetlight.protected \
-H 'Content-Type: application/json' \
-H 'apikey: yJv7sxFginCYPHq4v4ji-XTLP-Ya0stwAaGGkYuwgrw' \
-H 'routingKey: #' \
-d '{
      "ambientLux": "10000",
      "power": "73.26",
      "caseTemp": 34.5
    }'
```

- **Response**: `200 OK`

  Unless the API key is wrong, this endpoint will not throw any errors. For the sake of security and speed, any publish message will give back a `200 OK` but it may or may not have reached its intended destination.

2.3.3 Subscribe

- **Summary**: The subscribe API is used to get messages from a queue. This is a destructive action on the message as it will no longer be available in the queue after subscription. This API is not streaming, i.e. as data arrives into the queue it is not pushed to the subscribers. However, it needs to be polled regularly to check for new messages.
- **Endpoint**: `https://localhost:8443/api/1.0.0/subscribe/<queue_name>/<msg_count>`

Field	Description
<queue_name>	The queue from which the messages need to be subscribed.
<msg_count>	The number of messages, in integer, to subscribe from the queue

- **Method**: `GET`
- **Required Headers**:

Header Name	Description
apikey	API key of the device

- **Example Request**:

```
curl -X GET \
https://localhost:8443/api/1.0.0/subscribe/streetlight/1 \
-H 'apikey: pp1dGjKzsfHwVzK6g0nnVhR6fqnkKO-SVkXvasPXL6E'
```

- **Example Response**:

```
[
 {
   "data": {
     "ambientLux": 10000,
     "power": "73.26",
     "caseTemp": "34.5"
    },
   "topic": "#",
   "from": "streetlight.protected"
 }
]
```

- **Possible error scenarios**:
  - `Invalid authentication credentials`: Make sure you have provided the right API key
  - If the response is empty: Check if the queue is bound to the right exchange and that the exchange is receiving data from a source.

2.3.4 Follow

- **Summary**: The follow API is a data exchange enabling API that is used to express interest in another entity's data. It could either be for reading the data, or writing, or b oth. It is time-bound and the requestor has to explicitly mention the duration for which they would like to the data. Once a follow request is made, it goes to the <device>.follow queue of the owner's device. The owner can then review the request and issue a `/share`, thereby approving the follow request.

- **Endpoint**: `https://localhost:8443/api/1.0.0/follow`
- **Method**: `POST`
- **Required Headers**:

Header Name	Description
apikey	API key of the device

- **Body**: The body must contain certain specific fields as mentioned below:

Field	Description
entityID	Name of the entity which the requestor is interested in
permission	**Can be any of:** – `read` – `write` – `read-write`
validity	Of the form <Integer><Metric> **Metric can be any of:** – `Y`: Year – `M`: Month – `D`: Day Example: 10D for 10 days, 1Y for 1 Year and so on
requestorID	Name of the requesting device

- **Example Request**:

```
curl -X POST \
https://localhost:8443/api/1.0.0/follow \
-H 'Content-Type: application/json' \
-H 'apikey: ko6A9npXespXwyK85mtfCfmHSDLFYJZMOxScjk9iUJy' \
-d
'{
     "entityID": "device1",
     "permission":"read",
     "validity": "10D",
     "requestorID":"device2"
 }'
```

- **Example Response**:

```
{"status":"Follow request has been made to device1 with permission read at 2018-
↪09-10T09:42:57.226Z"}
```

- **Possible error scenarios**:
  - `Invalid authentication credentials` : Make sure you have provided the right API key
  - `Possible missing fields`: The fields required in the body, as given above, are not all present according to the format.

2.3.5 Share

- **Summary**: The "share" API is a data exchange enabling API which is used to approve "follow" requests. When the share API is called by an entity, it grants the necessary permissions to the requestor's device, i.e, either read access, or write access, or both. Once the process completes, the requesting entity gets a notification in their <device>.notify queue aprising them of the status of the follow request they made. Consequently, the requesting entity can either bind their queue to the requested exchange, or start publishing to it, depending upon the granted permissions.
- **Endpoint**: `https://localhost:8443/api/1.0.0/share`
- **Method**: `POST`
- **Required Headers**:

Header Name	Description
apikey	API key of the device

- **Body**: The body must contain certain specific fields as mentioned below:

Field	Description
entityID	Name of the owner's entity
permission	**Can be any of:** – `read` – `write` – `read-write`
validity	Of the form <Integer><Metric> **Metric can be any of:** – `Y`: Year – `M`: Month – `D`: Day Example: 10D for 10 days, 1Y for 1 year and so on
requestorID	Name of the requesting device

- **Example Request**:

```
curl -X POST \
https://localhost:8443/api/1.0.0/share \
-H 'Content-Type: application/json' \
-H 'apikey: ko6A9npXespXwyK85mtfCfmHGVLFYJZMOxScjk9iUJy' \
-d
'{
     "entityID": "device1",
     "permission":"read",
     "validity": "10D",
     "requestorID":"device2"
 }'
```

- **Example Response**:

```
{"status":"Share request approved for device2 with permission read at 2018-09-
↪09T04:19:33.484Z"}
```

- **Possible error scenarios**:

  - `Invalid authentication credentials` : Make sure you have provided the right API key
  - `Possible missing fields`: The fields required in the body, as given above, are not all present according to the format.

2.3.6 Bind

- **Summary**: The bind API is used to enable binding between queues and exchanges. A device owner has complete control over binding and unbinding exchanges and queues that belong to them. However, if they wish to bind to a different owner's device, they would need to go through the formal process of `follow` and `share`. On the other hand, all devices are free to bind to the public exchange of any other device.
- **Endpoint**: `https://localhost:8443/api/1.0.0/bind/<queue>/<exchange>`

Field	Description
<queue>	Name of the queue which needs to be bound to an exchange.
<exchange>	Exchange name. If this does not belong to the owner, the process of `follow` and `share` must happen.

- **Method**: `GET`
- **Required Headers**:

Header Name	Description
apikey	API key of the device

- **Optional Headers**:

Header Name	Description
routingKey	Topic with which the exchange must be bound to the queue

- **Example Request**:

```
curl -X GET \
https://localhost:8443/api/1.0.0/bind/app/streetlight.protected \
-H 'apikey: SLJTRxPdAVrslmHxcFPfQNWykVOIiIZ2hdiy0FSQOhB' \
-H 'routingKey: #'
```

- **Response**:

```
Bind Queue OK
```

- **Possible error scenarios**:
  - `Invalid authentication credentials`: Make sure you have provided the right API key
  - `You do not have access to bind this queue`: Make sure that the follow request made to a device has been approved before attempting to bind.

2.3.7 Catalogue

- **Summary**: The catalogue API is used to browse through the list of devices registered with the middleware.
- **Endpoint**: `https://localhost:8443/api/1.0.0/cat`

- **Method**: `GET`

2.3.8 Deregister

- **Summary**: The deregister API is used to completely remove all entries belonging to a device from the middleware. This deletes from the consumers list in Kong, deletes all 6 exchanges and 4 queues from RabbitMQ, removes entries in LDAP and finally deletes the document from catalogue. The owner of the specific device has to initiate this action, the device cannot deregister itself.
- **Endpoint**: `https://localhost:8443/api/1.0.0/register`
- **Method**: `DELETE`
- **Required Headers**:

Header Name	Description
apikey	API key of the owner of the device

- **Body**:

```
{"id":"<id_of_the_device>"}
```

- **Example Request**:

```
curl -X DELETE \
https://localhost:8443/api/1.0.0/register \
-H 'Content-Type: application/json' \
-H 'apikey: guest' \
-d '{"id":"streetlight"}'
```

- **Example Response**:

```
{
  "De-Registration": "success",
  "entityID": "streetlight3"
}
```

- **Possible error scenarios**:
  - `Invalid authentication credentials`: Make sure you have provided the right API key of the owner of the device
  - When the device does not belong to the owner:

    ```
    {
      "De-Registration": "failure",
      "Reason": "Device does not belong to Owner"
    }
    ```

    This means that the device being deregistered does not belong to the owner whose API key has been provided.
  - `ID not provided`: The device ID which needs to be deregistered has not been supplied

2.3.9 Unshare

- **Summary**: The unshare API is used to revoke the permissions given to a device. This deletes the `share` entry from LDAP and unbinds the queue if bound. This API can be used by the owner's device when the data lease time of the requesting device has expired, or the owner for some reason deems that the requestor is no longer eligible for their data.
- **Endpoint**: `https://localhost:8443/api/1.0.0/share`
- **Method**: `DELETE`
- **Required Headers**:

Header Name	Description
apikey	API key of the device

- **Body**: The body must contain certain specific fields as mentioned below:

Field	Description
entityID	Name of the owner's entity
permission	**Can be any of:** – `read` – `write` – `read-write`
validity	Of the form <Integer><Metric> **Metric can be any of:** – `Y`: Year – `M`: Month – `D`: Day Example: 10D for 10 days, 1Y for 1 year and so on
requestorID	Name of the requesting device

- **Example Request**:

```
curl -X DELETE \
https://localhost:8443/api/1.0.0/share \
-H 'Content-Type: application/json' \
-H 'apikey: ko6A9npXespXwyK85mtfCfmHGVLFYJZMOxScjk9iUJy' \
-d
'{
     "entityID": "device1",
     "permission":"read",
     "validity": "10D",
     "requestorID":"device2"
 }'
```

- **Example Response**:

```
Successfully unshared from device2
```

- **Possible error scenarios**:
  - `Invalid authentication credentials` : Make sure you have provided the right API key

  - `Possible missing fields`: The fields required in the body, as given above, are not all present according to the format.

2.3.10 Unfollow

- **Summary**: The unfollow API is a complement of the `unshare` API for the requestor. If the requestor at any point feels that he/she no longer needs the permissions granted by a device, they can call the `unfollow` API. This will delete entries from LDAP and unbind the queue if bound.
- **Endpoint**: `https://localhost:8443/api/1.0.0/follow`
- **Method**: `DELETE`
- **Required Headers**:

Header Name	Description
apikey	API key of the device

- **Body**: The body must contain certain specific fields as mentioned below:

Field	Description
entityID	Name of the entity which the requestor is interested in
permission	**Can be any of:** – `read` – `write` – `read-write`
validity	Of the form <Integer><Metric> **Metric can be any of:** – `Y`: Year – `M`: Month – `D`: Day Example: 10D for 10 days, 1Y for 1 Year and so on
requestorID	Name of the requesting device

- **Example Request**:

```
curl -X DELETE \
https://localhost:8443/api/1.0.0/follow \
-H 'Content-Type: application/json' \
-H 'apikey: ko6A9npXespXwyK85mtfCfmHSDLFYJZMOxScjk9iUJy' \
-d
'{
     "entityID": "device1",
     "permission":"read",
     "validity": "10D",
     "requestorID":"device2"
 }'
```

- **Example Response**:

```
Successfully unfollowed device1
```

- **Possible error scenarios**:
  - `Invalid authentication credentials` : Make sure you have provided the right API key
  - `Possible missing fields`: The fields required in the body, as given above, are not all present according to the format.

2.3.11 Unbind

- **Summary**: The unbind API is used to enable unbinding queues and exchanges. Unbinding can be done when the default bindings between the exchanges and queues does not suit the user. It can also be done when the device temporarily does not need to receive data from another device but wishes to keep the share entry intact.
- **Endpoint**: `https://localhost:8443/api/1.0.0/bind/<queue>/<exchange>`

Field	Description
<queue>	Name of the queue which needs to be bound to an exchange.
<exchange>	Exchange name. If this does not belong to the owner, the process of `follow` and `share` must happen.

- **Method**: `DELETE`
- **Required Headers**:

Header Name	Description
apikey	API key of the device

- **Optional Headers**:

Header Name	Description
routingKey	Topic with which the exchange must be bound to the queue

- **Example Request**:

```
curl -X DELETE \
https://localhost:8443/api/1.0.0/bind/app/streetlight.protected \
-H 'apikey: SLJTRxPdAVrslmHxcFPfQNWykVOIiIZ2hdiy0FSQOhB' \
-H 'routingKey: #'
```

- **Response**:

```
Unbind Queue OK
```

- **Possible error scenarios**:
  - `Invalid authentication credentials`: Make sure you have provided the right API key
  - `You do not have access to unbind this queue`: Make sure that you have the permission to unbind the queue. A queue can be unbound from an exchange in two scenarios only
    - If the exchange and queue belong to the same owner
    - If the exchange belongs to a different owner but the owner has `share` d with the current device

2.4 Usage

The following workflow will attempt to illustrate the typical usage of the smart city stack.

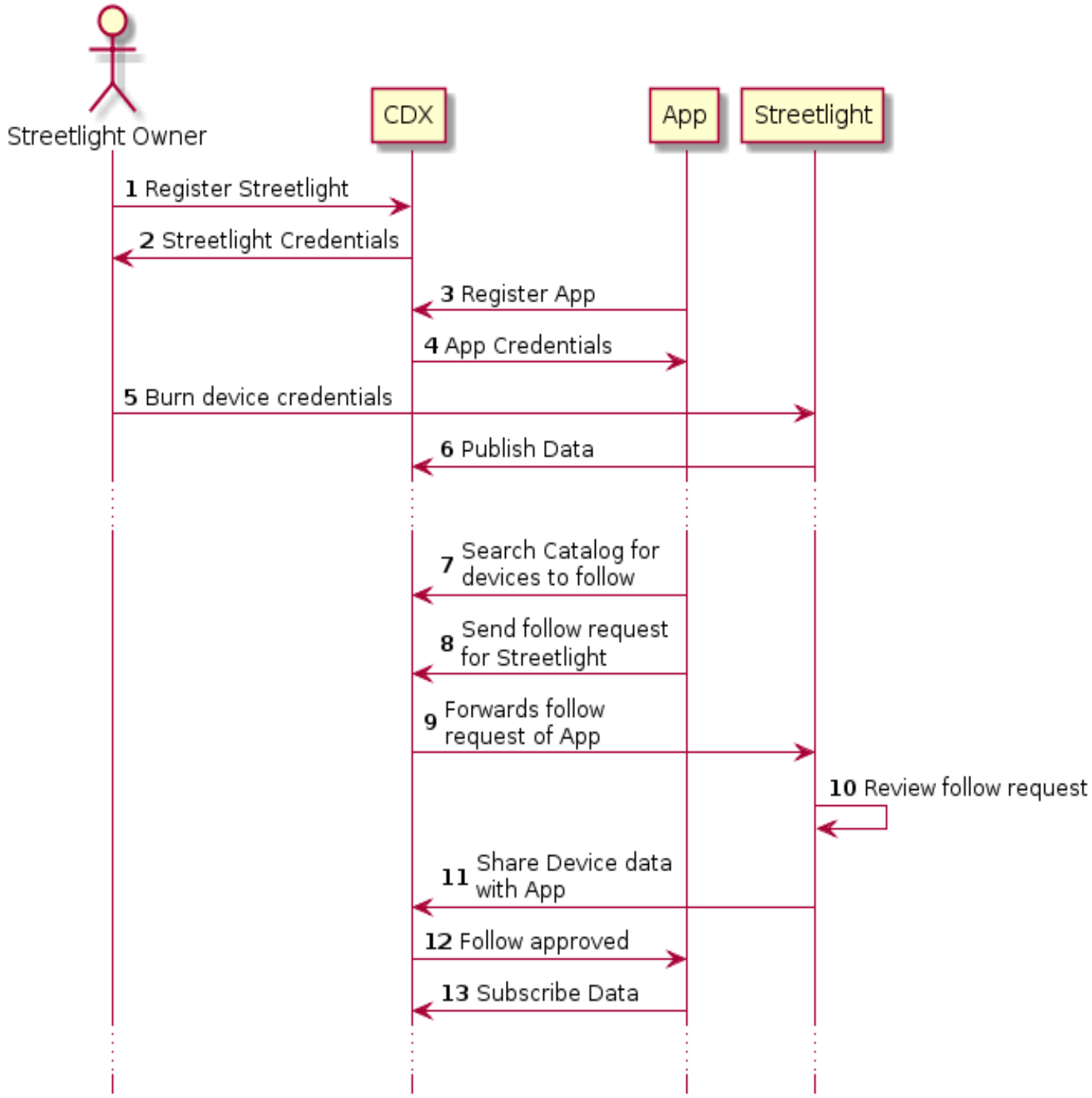


Figure 1 : Usage overview of CDX

1. Consider that two devices have registered with the middleware viz. “streetlight” and “app”. The API keys obtained are as follows:

   For the streetlight

```
{
  "Registration": "success",
  "entityID": "streetlight",
  "apiKey": "ReiHIzTA6hQRbv3JO2gRHP9iEh4NzK-bBSl-J0GKJZo",
  "subscriptionEndPoint": "https://smartcity.rbccps.org/api/{version}/follow?
↪id=streetlight",
  "accessEndPoint": "https://smartcity.rbccps.org/api/{version}/db?id=streetlight
↪",
  "publicationEndPoint": "https://smartcity.rbccps.org/api/{version}/publish?
↪id=streetlight",
  "resourceAPIInfo": "https://rbccps-iisc.github.io"
}
```


For the app


```
{
  "Registration": "success",
  "entityID": "app",
  "apiKey": "7KfqcDOPvz3JECUwwGsQYuTdKtyGTQTpLDaZjIbJpMt",
  "subscriptionEndPoint": "https://smartcity.rbccps.org/api/{version}/follow?
↪id=app",
  "accessEndPoint": "https://smartcity.rbccps.org/api/{version}/db?id=app",
  "publicationEndPoint": "https://smartcity.rbccps.org/api/{version}/publish?
↪id=app",
  "resourceAPIInfo": "https://rbccps-iisc.github.io"
}
```


2. Now let's say that the device `app` is interested in `streetlight` 's data. It should make a `follow` request to `streetlight` as below:

```
curl -X POST \
https://localhost:8443/api/1.0.0/follow \
-H 'Content-Type: application/json' \
-H 'apikey: 7KfqcDOPvz3JECUwwGsQYuTdKtyGTQTpLDaZjIbJpMt' \
-d '{
      "entityID": "streetlight",
      "permission":"read",
      "validity": "10D",
      "requestorID":"app"
    }'
```

Note that the API key used to make the follow request is that of `app` 's. Once the request goes through, we get the success response as


```
{"status":"Follow request has been made to streetlight.follow with permission␣
↪read at 2018-09-10T17:15:23.330Z"}
```


3. The device `streetlight` would have got a notification in its `streetlight.follow` queue. It will subscribe using:

```
curl -X GET \
https://localhost:8443/api/1.0.0/subscribe/streetlight.follow/1 \
-H 'apikey: ReiHIzTA6hQRbv3JO2gRHP9iEh4NzK-bBSl-J0GKJZo'
```

Note that the API key used is that of `streetlight`'s. This gives back a response as:

```
[
  {
    "data": {
        "permission": "read",
        "validity": "10D",
        "requestor": "app",
        "timestamp": "2018-09-10T17:15:23.311Z"
        },
    "topic": "#",
    "from": "streetlight.follow"
  }
]
```

4. Now that the device `streetlight` knows that the device `app` is interested in its data, it can approve the `follow` by calling the `share`:

```
curl -X POST \
https://localhost:8443/api/1.0.0/share \
-H 'Content-Type: application/json' \
-H 'apikey: ReiHIzTA6hQRbv3JO2gRHP9iEh4NzK-bBSl-J0GKJZo' \
-d '{
       "entityID": "streetlight",
       "permission":"read",
       "validity": "10D",
       "requestorID":"app"
    }'
```

Note that the API key used to call the share is that of `streetlight`'s. The above request would give back a response as

```
{"status":"Share request approved for app with permission read at 2018-09-
↪10T17:26:50.908Z"}
```

5. The issuance of `share` by the `streetlight` sends out a notification to the `app.notify` queue. The `app` device can retrieve the status using:

```
curl -X GET \
https://localhost:8443/api/1.0.0/subscribe/app.notify/1 \
-H 'apikey: 7KfqcDOPvz3JECUwwGsQYuTdKtyGTQTpLDaZjIbJpMt'
```

The API key used is that of `app`'s. This would give out a response as

```
[
  {
    "data":
        {
          "Status update for follow request sent to streetlight": "Approved. You␣
↪can now bind to streetlight.protected"
        },
    "topic": "#",
    "from": "app.notify"
  }
]
```

6. Now that the device `app` has undertsood that `stretlight` has approved the request for read, it can now bind its queue to `streetlight.protected` using:

```
curl -X GET \
https://localhost:8443/api/1.0.0/bind/app/streetlight.protected \
-H 'apikey: 7KfqcDOPvz3JECUwwGsQYuTdKtyGTQTpLDaZjIbJpMt' \
-H 'routingKey: #'
```

Note that the API used is that of app's. This above request would give out a success message as:

```
Bind Queue OK
```

7. Now if streetlight publishes any data using its API key:

```
curl -X POST \
https://localhost:8443/api/1.0.0/publish/streetlight.protected \
-H 'Content-Type: application/json' \
-H 'apikey: ReiHIzTA6hQRbv3JO2gRHP9iEh4NzK-bBSl-J0GKJZo' \
-H 'routingKey: #' \
-d '{
      "ambientLux": "10",
      "caseTemp": 34.5
    }'
```

The device app can also get a copy of the data in its queue by using:

```
curl -X GET \
https://localhost:8443/api/1.0.0/subscribe/app/1 \
-H 'apikey: 7KfqcDOPvz3JECUwwGsQYuTdKtyGTQTpLDaZjIbJpMt'
```

Note the use of app's API key to subscribe. This would give a response as

```
[
 {
   "data":
     {
       "ambientLux": "10",
       "caseTemp": 34.5
     },
   "topic": "#",
   "from": "streetlight.protected"
 }
]
```

2.5 CDX Subsystems

CDX consist of the following systems:

1. **Authenticator and API gateway** : A system which is the entry point for users of the CDX. It authenticates users and if successful connects users to the appropriate microservice inside the CDX.
2. **Data broker** : The main system of CDX which allows users to publish and subscribe data in a secure fashion. This provides the interface to access live data being exchanged with the CDX.
3. **Catalog** : The service which provides the user with a list of devices/apps currently connected to the CDX. It also provides other meta-information such as the data format in which messages are subscribed/published by the device/app.
4. **Database** : The system which stores the historic data.

5. **AAA server** : The system which authenticates and authorizes users.
6. **Video server** : A system which acts as a broker for videos in a smart-city.
7. **Identity manager** : A system which is responsible for providing credentials and maintaining identity of CDX users.
8. **Certificate authority** : A system which provides certificates for CDX users.
9. **Log server** : A system which monitors logs from various system in the CDX. These logs are collected to detect or prevent any intrusions.
10. **IDPS** : The intrusion detection and prevention system, which analyzes the logs produced by the log server and takes appropriate actions to detect/prevent potential intrusions.

More details about the sub-systems can be found in the following sections

2.5.1 Catalog

The information resource catalog is one of the key components of the CDX stack. It contains a list of data resources and their descriptions in a machine readable format. The resource descriptions may include API endpoints, and other meta-data like access hints, ownership, providers, structure of data, e.g., list of parameters and their descriptions etc. A useful analogy is the online shopping catalog, where a consumer can browse through available products, their features, reviews etc. and then decide to purchase a subscription or a copy of them. The resource catalog plays a similar role for the clients wishing to use the smart city resources and, as additional benefit, it can help nucleate the development of a new data/digital economy.

Key services offered by resource catalog are:

1. Secure, authenticated, open API based
    - Discovery of information sources
    - Addition/Modification of meta-data for new/existing resources
    - Semantic search over resource meta-data
2. API framework supporting
    - Filtering of query results (For example, sorting, access modifier based filtering, specific field based filtering etc.)
    - Grouping

As an example, Hypercat is a recent alliance which has developed a standard for a catalogue under the British Standards Institution (PAS212). It is a lightweight, JSON based hypermedia catalogue format for exposing a collection of Uniform Resource Identifiers (URI) for IoT assets. The interactions with the catalogue use REST APIs over HTTP and the catalogue items are stored as RDF-like triples. RDF triples are essentially of the form "Subject, Predicate, Object" and can be used to encode relationships and hence provide a way to link the data/device parameters to external concepts. This will enable creating appropriate metadata for the devices that can then provide the right context and semantics for analysing the data from the IoT device. The framework allows a lot of flexibility in defining new annotations for new devices or information sources. In addition, Hypercat also advocates a few different search APIs, with a key one based on geographic indices. Another important feature in Hypercat is the possibility of having links to other catalogues. This is to enable federated/distributed growth of IoT resources which will be a very desirable property.

Data Schemas

From the above, it is clear that the resource meta-data is the key element that enables data interoperability. However, this is possible only if the meta-data is described with a common vocabulary which is unambiguous and free from

semantic conflicts.

Open data schemas from all data sources (and actuators) needs to be developed to enable data interoperability. Key desirable features of data schemas are:

1. Flexibility to accommodate diverse nature of underlying data sources
2. Describable in both human and machine-readable formats
3. Should be accompanied by a rich set of tools to create, modify and validate schema data
4. Should be extensible, allowing for easy enhancement

There are many online repositories for schemas like json-schema.org (for schema templates in JSON), www.oneiota.org for some data schemas for some IoT devices. In addition, projects like FiWare have also proposed schemas. We need to adapt and extend existing schemas (and develop new ones if they do not exist), to meet the needs for the IoT devices (and other data resources) for the Indian Smart Cities. As an example, we have discussed (and made publicly available) examples of schemas for street light and power meter.

Recently, Open Connectivity Foundation, which is an international consortium of companies and some academic institutions, has developed data schemas for many IoT devices. The schemas are based on "JSON Schemas". We find JSON-Schemas to be a better way to describe the metadata than that proposed in Hypercat as there are open-source tools readily available that allow schema creation, integrity checking of the schemas etc. (Essentially, JSON-schema allows a simple grammar to be defined and hence it is easy to check schemas against that). This will be particularly important when one wants "user-contributed" schema development and deployment.

We recommend combining the best features of Hypercat (API based access to catalogue, with powerful search capabilities), and OCF (JSON-Schema based information description to enable integrity checking). We believe that the structures proposed in OCF can be further extended as described next.

As in OCF, we recommend to use JSON schemas to define the structure of each entry in the resource catalog. The schemas define specific fields to be included in the metadata for a given class of devices. Apart from mandating the inclusion of certain fields, there exist provisions to include (optional) device/vendor specific fields, e.g., links to reference ontologies, additional device information, usage hints etc., which enhances the usability of data by third party applications. The schemas are further used to validate the catalog entries at the time of on-boarding the device using the easily available json-schema validation tools.

Towards defining the structure of catalog items, we propose to organize the metadata into following categories:

- Static information: General information about the device/information resource, e.g., geo-location, type of device, ID, tags etc.
- Message sub-schemas: Meta-information for the various types of messages that the device/resource cand send or receive. Access to these messages can be provisioned based on individual message types. Examples include, but are not limited to, the following message types:
  - Observation messages: Sensed parameters etc.
  - Control messages: Actuation parameters accepted by the device, e.g., control inputs etc.
  - Configuration messages: Configuration parameters of the device, e.g., sampling rates, sleep times etc.
  - Management messages: Messages related to Device management.
  - JSON-RPC messages and responses.

Of course the above message types could be further extended in the future.

A key idea is to store dynamic meta-information within a catalog entry as a JSON-schema itself. Thus, the metadata becomes an actionable specification that can be used to dynamically validate various messages being sent from/to the device (see Figure 1).

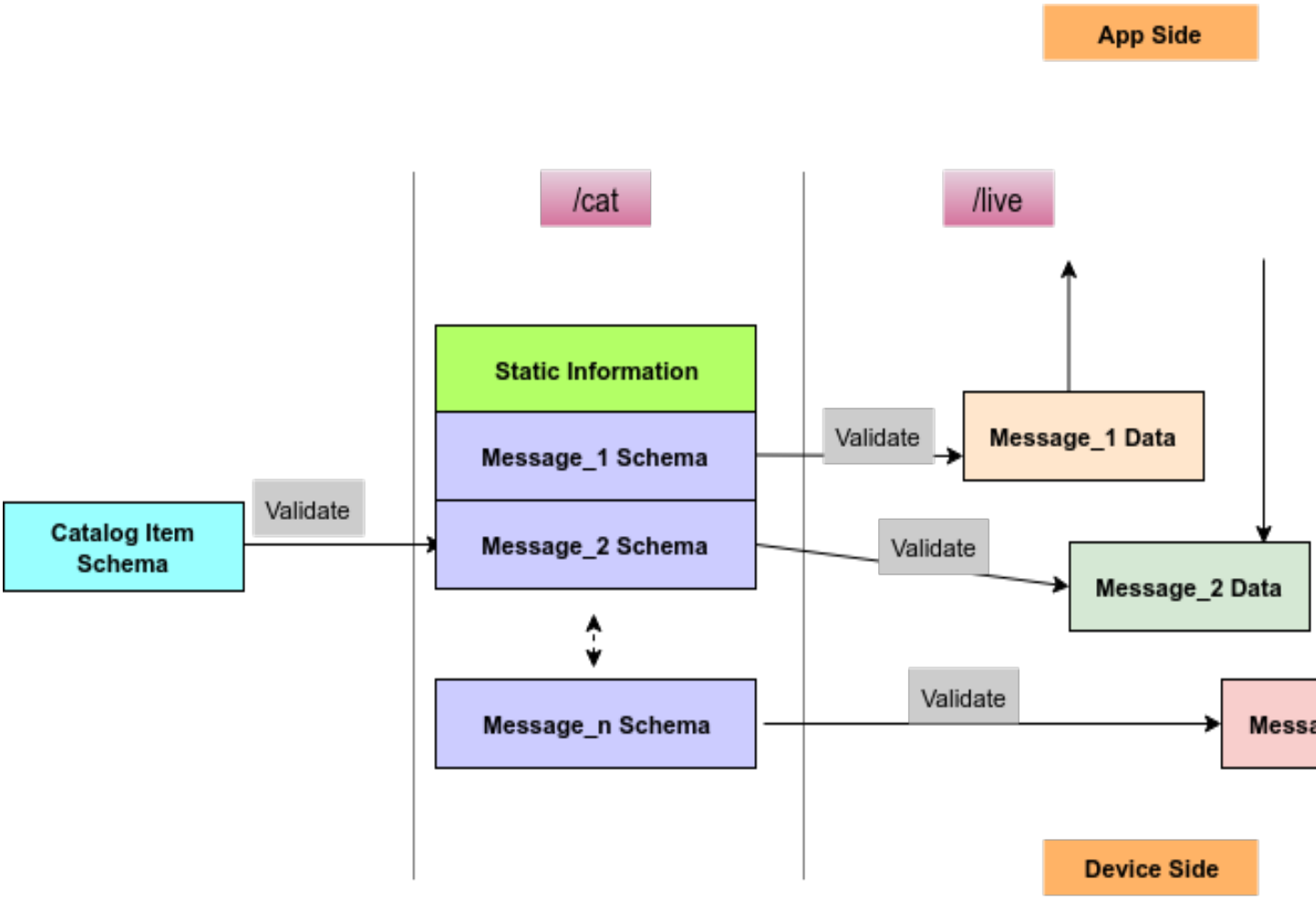


Figure 1: A device schema in the catalog is validated against a global-meta-schema. Each data item coming from/to the device is further validated by the embedded schemas in the device schema. /cat is the API endpoint for the Information Resource Catalog in the CDX stack.

We give an example of a schema entry in the catalog for a smart streetlight

```
{
  "refCatalogueSchema": "generic_iotdevice_schema.json",
  "id": "70b3d58ff0031de5",
  "tags": [
    "Onstreet",
    "streetlight",
    "Energy",
    "still under development!"
  ],
  "refCatalogueSchemaRelease": "0.1.0",
  "latitude": {
    "value": 13.0143335,
    "ontologyRef": "http://www.w3.org/2003/01/geo/wgs84_pos#"
  },
  "longitude": {
    "value": 77.5678424,
    "ontologyRef": "http://www.w3.org/2003/01/geo/wgs84_pos#"
```

(continues on next page)

(continued from previous page)

```
  },
  "owner": {
    "name": "Indian Institute of Science",
    "Id": "IISC",
    "website": "http://www.iisc.ac.in"
  },
  "provider": {
    "name": "Robert Bosch Centre for Cyber Physical Systems, IISc",
    "Id": "RBCCPS",
    "website": "http://rbccps.org"
  },
  "geoLocation": {
    "address": "80 ft Road, Bangalore, 560012"
  },
 "data_schema": {
"type": "object",
"properties": {
    "observation_msg": {
        "type": "object",
        "direction": "from-device",
        "accessModifier": "public",
        "tags": [
          "on street",
           "energy"
        ],
        "properties": {
           "dataSamplingInstant": {
             "type": "string",
             "description": "Sampling Time in UTC format",
             "units": "seconds"
           },
           "caseTemperature": {
             "type": "number",
             "description": "Temperature of the device casing",
             "units": "degreeCelsius"
           },
           "powerConsumption": {
             "type": "number",
             "description": "Power consumption of the device",
             "units": "watts"
           },
           "luxOutput": {
             "type": "number",
             "description": "lux output of LED measured at LED",
             "units": "lux"
           },
           "ambientLux": {
             "type": "number",
             "description": "lux value of ambient",
             "units": "lux"
           }
        },
        "additionalProperties": false
    },
    "setpoints_msg": {
        "type": "object",
        "direction": "from-device",
```

(continues on next page)

(continued from previous page)

```
        "accessModifier": "protected",
        "tags": [
           "on street",
           "energy",
           "control"
        ],
        "properties": {
            "targetPowerState": {
              "type": "string",
              "enum": [
                "ON",
                "OFF"
              ]
            },
            "targetBrightnessLevel": {
              "type": "number",
              "description": "Number between 0 to 100 to indicate the percentage␣
↪brightness level.",
              "units": "percent"
            },
            "targetControlPolicy": {
              "enum": [
                "AUTO_TIMER",
                "AUTO_LUX",
                "MANUAL"
              ],
              "description": "Indicates which of the behaviours the device is␣
↪currently set to. AUTO_TIMER is timer based, AUTO_LUX uses ambient light and MANUAL␣
↪is controlled by app. Writeable only allowed for authorized apps"
            },
            "targetAutoTimerParams": {
              "type": "object",
              "properties": {
                "targetOnTime": {
                  "type": "number",
                  "description": "Indicates time of day in seconds from 12 midnight␣
↪when device turns ON in AUTO_TIMER.",
                  "units": "seconds"
                },
                "targetOffTime": {
                  "type": "number",
                  "description": "Indicates time of day in seconds from 12 midnight␣
↪when device turns OFF in AUTO_TIMER.",
                  "units": "seconds"
                }
              }
            },
            "targetAutoLuxParams": {
              "type": "object",
              "properties": {
                "targetOnLux": {
                  "type": "number",
                  "description": "Indicates ambient lux when device turns ON in AUTO_
↪LUX.",
                  "units": "lux"
                },
                "targetOffLux": {
```

(continues on next page)

(continued from previous page)

```
                    "type": "number",
                    "description": "Indicates ambient lux when device turns OFF in AUTO_
↪LUX.",
                    "units": "lux"
                  }
                }
              }
        },
        "additionalProperties": false
    }
  },
    "control_msg": {
        "type": "object",
        "direction": "to-device",
        "accessModifier": "protected",
        "tags": [
           "on street",
           "energy",
           "control"
        ],
        "properties": {
            "targetPowerState": {
              "type": "string",
              "enum": [
                "ON",
                "OFF"
              ],
              "description": "If set to ON, turns ON the device. If OFF turns OFF the␣
↪device. Writeable parameter. Writeable only allowed for authorized apps"
            },
            "targetBrightnessLevel": {
              "type": "number",
              "description": "Number between 0 to 100 to indicate the percentage␣
↪brightness level. Writeable only allowed for authorized apps",
              "units": "percent"
            },
            "targetControlPolicy": {
              "enum": [
                "AUTO_TIMER",
                "AUTO_LUX",
                "MANUAL"
              ],
              "description": "Indicates which of the behaviours the device should␣
↪implement. AUTO_TIMER is timer based, AUTO_LUX uses ambient light and MANUAL is␣
↪controlled by app. Writeable only allowed for authorized apps"
            },
            "targetAutoTimerParams": {
              "type": "object",
              "properties": {
                "targetOnTime": {
                  "type": "number",
                  "description": "Indicates time of day in seconds from 12 midnight␣
↪when device turns ON in AUTO_TIMER. Writeable only allowed for authorized apps",
                  "units": "seconds"
                },
                "targetOffTime": {
                  "type": "number",
```

(continues on next page)

(continued from previous page)

```
                    "description": "Indicates time of day in seconds from 12 midnight␣
↪when device turns OFF in AUTO_TIMER. Writeable only allowed for authorized apps",
                    "units": "seconds"
                  }
                }
              },
              "targetAutoLuxParams": {
                "type": "object",
                "properties": {
                  "targetOnLux": {
                    "type": "number",
                    "description": "Indicates ambient lux when device turns ON in AUTO_
↪LUX. Writeable only allowed for authorized apps",
                    "units": "lux"
                  },
                  "targetOffLux": {
                    "type": "number",
                    "description": "Indicates ambient lux when device turns OFF in AUTO_
↪LUX. Writeable only allowed for authorized apps",
                    "units": "lux"
                  }
                }
              }
            },
            "additionalProperties": false
        }
    },
    "oneOf": [
      {"required": [ "observation_msg" ] },{"required": [ "control_msg" ]},{"required":␣
↪[ "setPoint_msg" ] }
    ]
}
}
```

Figure 2: Example catalogue schema for a smart streetlight

There are two portions to the above schema, highlighted by the two shades. The first (in green shading) pertains to static information. The second part (light blue shaded) provides meta-information about various message types the device receives/sends. In the above example, the streetlight device has three messages: observation_msg: Observations made by the streetlight (caseTemperature, ambientLux etc.). This message flows "from-device" as is indicated by the keyword "direction". For example, the streetlight measures "caseTemperature" which is a number and it describes the temperature of the light casing and its units are "degreeCelcius".

Control parameters for this device (targetPowerState, targetBrightnessLevel etc.). This message flows "to-device" as is indicated by the keyword "direction". For example, the parameter "targetPowerState" can be used to switch ON (and OFF) the streetlight. This parameter is a dimensionless quantity. setPoint parameters for this device (targetPowerState, targetBrightnessLevel etc.). This message flows "from-device" as is indicated by the keyword "direction". It indicates the values of writeable parameters which may have been set to certain values using control messages.

The structure for streetlight catalog item is defined by the base schema "generic_iotdevice_schema.json". This information is contained within the static information part of the item using the json fields, namely "refCatalogueSchema" and "refCatalogueSchemaRelease". To enable format validation of an uploaded catalog item, it should mandatorily mention the name of reference schema files and the schema release numbers.

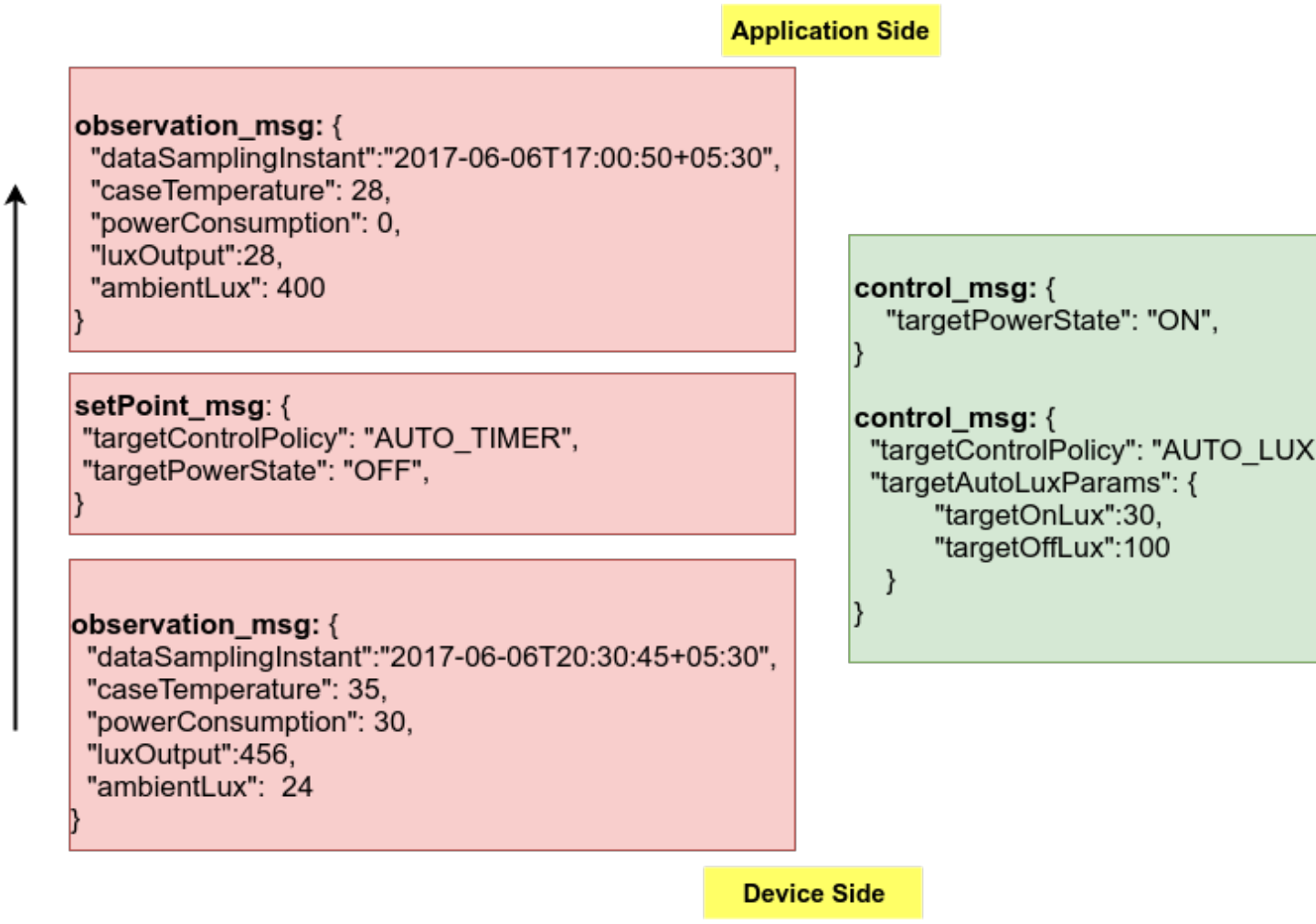


Figure 3: Streetlight observation and control data

To highlight the validation features of the catalog, we present an example of the observation, setPoint (up-arrow) and control (down arrow) data packets for the streetlight device in Figure 3. A key point is that, the observation schema which is stored in catalog (e.g., see the example streetlight item above), can be directly used to validate the incoming observation packets. The same applies to the control packets in the reverse direction.

Another important field in the above example is the "ontologyRef" field included within "latitude" and "longitude" fields. This field provides a reference to a site where further explanation as to the meaning/semantics of the "key" can be found. For example, a website explicitly created to hold the ontology (dictionary) for that IoT device parameters in a standard ontology language (for example RDF).

We recommend that an Indian City specific dictionary/ontology be created and which contains semantic descriptions of things which are unique to Indian cities and hence not available in other existing ontologies. The dictionary as well as the catalogue should be ideally made available in all Indian languages. The catalogue can be accessed and searched via HTTP(S) commands and hence will be accessible from anywhere in the web. We advocate a resource catalogue be a part of every smart city middleware instance and it adheres to a standard schema as well as an API format for querying.

2.5.2 IoT Data Exchange

IoT devices are the key behind building a Smart City which enables efficient management of services offered by a City. These devices can range from being simple ones such as a temperature sensor to a more complex and sophisticated

ones such as video cameras. Data produced by these devices will lead to building smart applications for managing the city efficiently. For example, Smart Waste Management, Smart Street Light Management, Smart Water Management, Smart Energy Management, Smart traffic Management etc.

A problem which we see currently is the siloed solution provided by application or solution vendors to build smartness into the city infrastructure. For example, a Smart Energy Monitoring and Control application will not be able to consume the data produced by a Smart traffic Management system or have the possibility to control the Street Lights using the Smart Street Light Management system in order to optimise the power usage. This makes us realize that building a common platform for bringing in the data from multiple silos a necessity for interoperability and efficient management of data.

To enable multiple such new applications in a multiplicity where different solution providers are in contention to provide smart solutions, it is critical for smart cities to adopt an API based, platform approach towards exchanging, consuming, metering and monetising data.

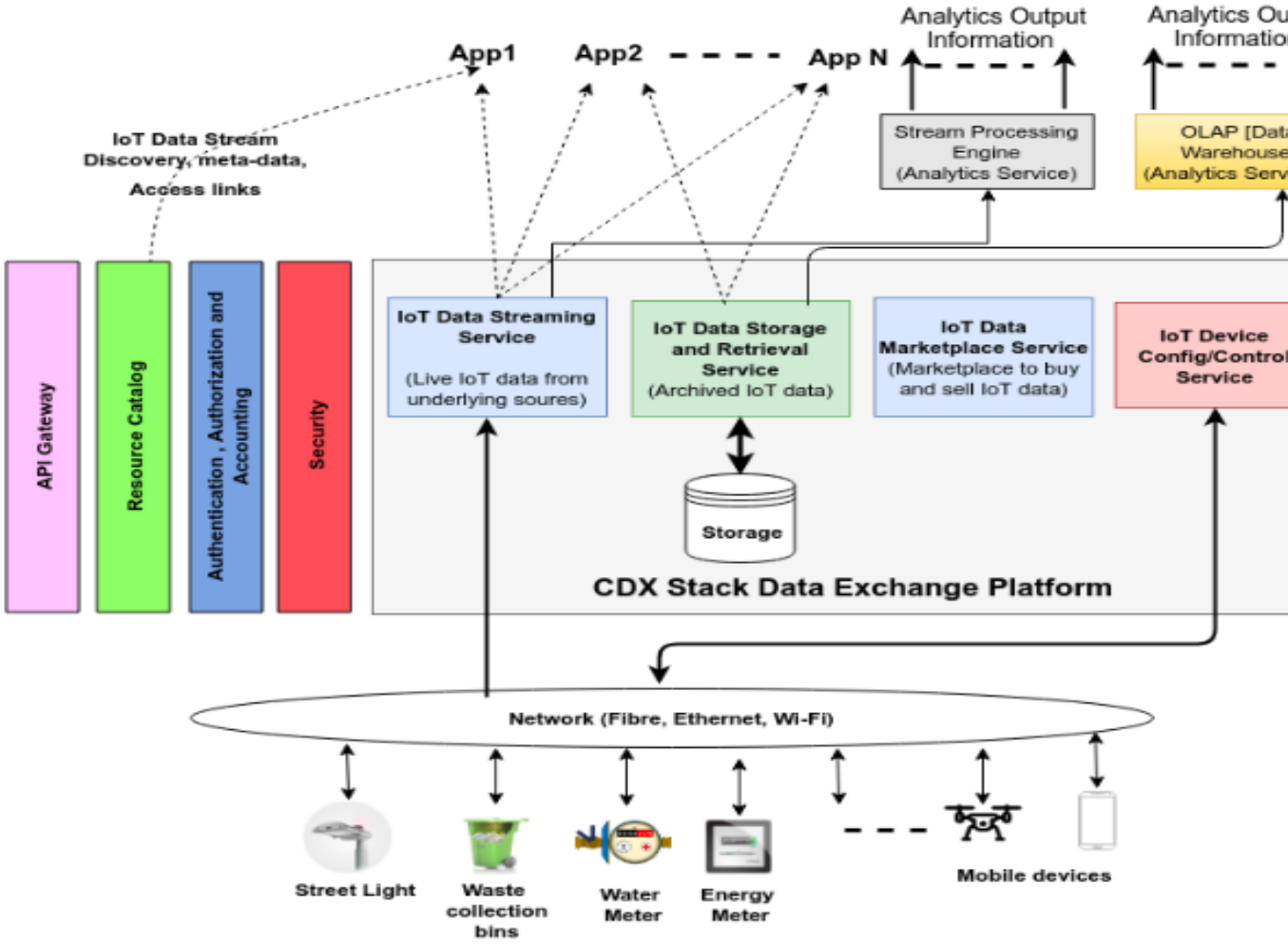


Figure 1 : Service overview of IoT Data Exchange

To cater the requirements for an interoperable smartcity stack, we need sophisticated services in the Data Exchange as follows:

- IoT Data Streaming Service
- IoT Data Storage and Retrieval Service
- IoT Data Monetization Service

- IoT Device Configuration and Control Service
- Secure, authenticated, authorised and access controlled ways to access data exchange via APIs

IoT Data Streaming Service

IoT Data Streaming Service is an IoT Message Bus for exchange of real-time time-series IoT data. Services offered by an IoT Data Streaming Service are as follows:

- **Dynamic Data Ingestion**
  - Secure, authenticated, authorised framework to enable ingestion of data
- **Data Distribution**
  - Secure, authenticated, authorised framework to enable development of new services and applications which consumes the data produced by the various data sources
  - Framework to support decoupling and asynchrony between the producers and consumers
- **Data Metering**
  - Secure, authenticated, authorised framework to enable metering of data for both time-based and data-size based access
- **Data Streaming APIs**
  - Secure, authenticated, authorised, access controlled API framework for interacting with the data streaming service
- **Multi-Protocol support**
  - Support for diverse set of application protocols which can be used to implement the APIs and include: AMQP, STOMP, XMPP, WebSockets and HTTP

IoT Data Streaming APIs

IoT data exchange shall have endpoints to access services in a secure, authenticated, authorised, access controlled way.

- **Publication**
  - Publication to CDX stack is done by Devices (sensors, virtual sensors) and Applications (device controllers, device managers etc.). The IoT data streaming service should provide necessary interface to devices (for sending sensed information) and applications (for sending control commands and configuration changes) through specific endpoints.
- **Subscription**
  - Similar to publication, a subscription to CDX stack is done by Devices (for receiving control commands and configuration changes) and Applications (for receiving sensed information). The IoT data streaming service should provide necessary interface to devices and applications through specific endpoints.

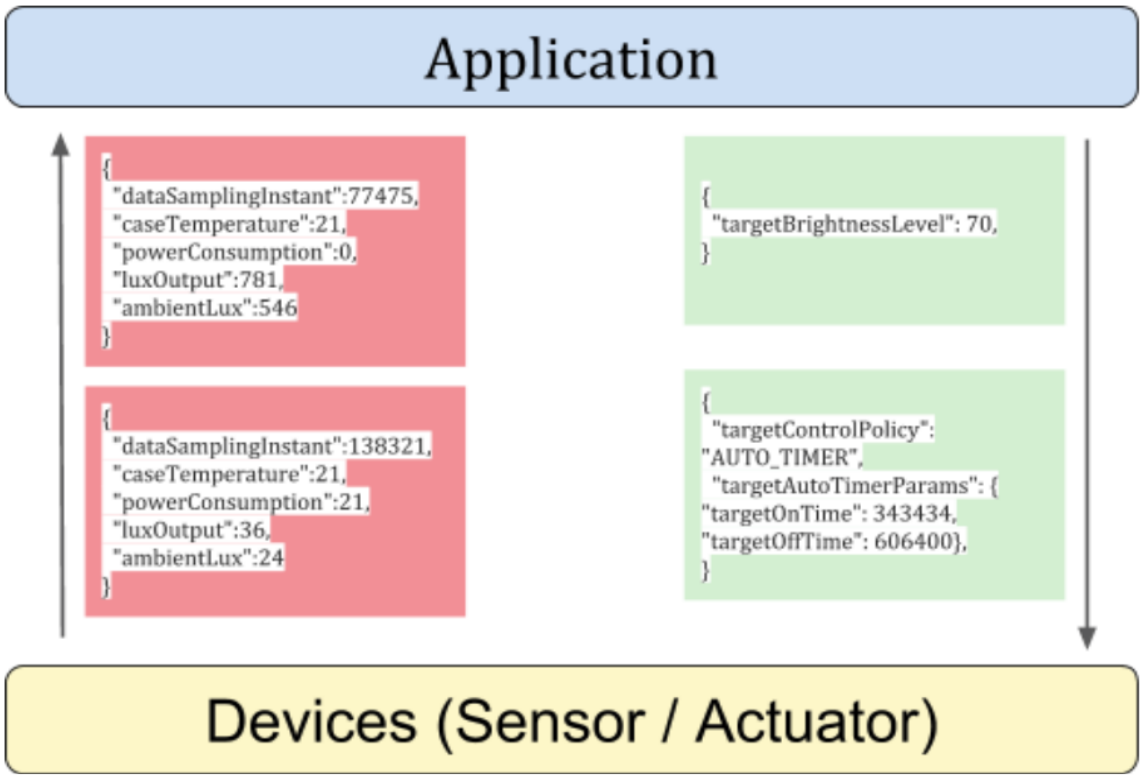


Figure 2 : Streetlight observation and control data

The above example is that of a streetlight publishing sensed data (red-block with up-arrow) and subscribing commands or configuration parameters (green-block with down-arrow) as per the schema provided during on-boarding. As a security policy, the system should make necessary validation and security checks to prevent devices from publishing data that doesn't adhere to the schema defined during on-boarding.

Some architectural requirements that needs to be looked into for implementation of data exchange platform are as follows:

- Reliable delivery - Support for providing a reliable message delivery using a protocols which are well tested, reliable, and optimized for high-data traffic
- Secure transactions - Support Authentication, Authorization and mechanisms to include Network Security
- Operational Maintenance - Support Monitoring, Auditing and Troubleshooting of the messaging system
- Uninterrupted Scaling - Support scalability through distributed and clustered modes of operation
- Real-Time IoT Message Bus - for enabling low latency actuation of mission critical applications

IoT Data Store Service

An IoT Data Store shall be considered as an Online transaction processing (OLTP) system which shall contain a repository (source data) from which thorough inferences can be made about the data as well as the functioning of the smart city. For example, these endpoints can be used for requesting the last known state of a streetlight or the values of the illuminance level over the past week, from a particular streetlight etc.

The IoT Data Store can leverage the existence of an Online analytical processing (OLAP) layer within the CDX stack. OLAP shall be considered as a layer on top of time-series IoT Data store to perform analytics. It can perform Extract Transform Load (ETL) operations with complex aggregate queries to provide multi-dimensioned data. This OLAP

system could be an analytics application providing more insights over the time-series IoT Data store. This analyzed multidimensional data can then be pushed back into the stack.

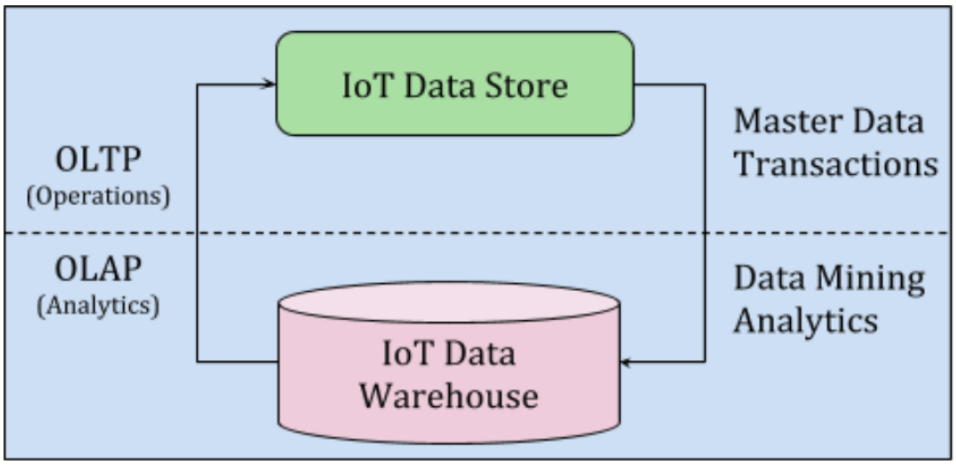


Figure 3 : CDX view of a Data Store and Warehouse

Services offered by an IoT Data Store Service are as follows:

- **Data Querying**
  - Secure, authenticated, authorised framework to support multi-dimensional queries
- **Data Analytics**
  - Secure, authenticated, authorised framework to submit analytic queries
- **Multi-Data Indexing**
  - Support for providing multi-dimensioned or multi index data for ease of querying
- **Data Store APIs**
  - Well defined APIs to interact with the datastore for ease of operation

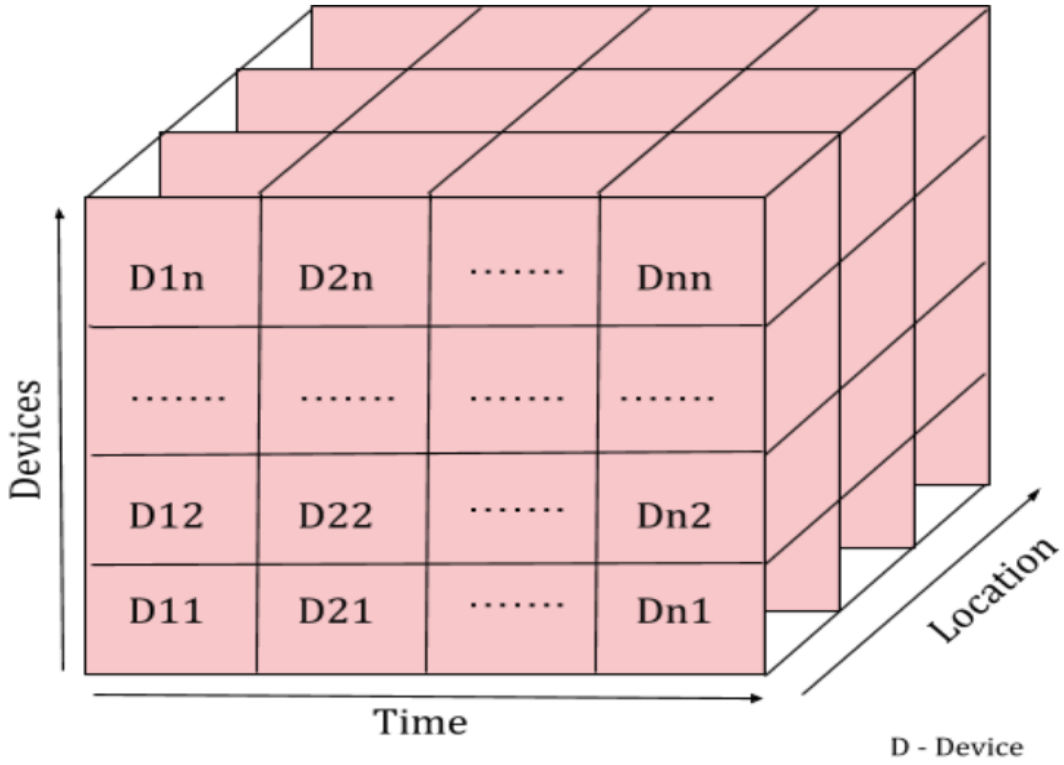


Figure 4 : Multidimensional indexed view of a Data Store

Some architectural requirements that needs to be looked into for implementation of Data Store is given below:

- Swift write performance: There will be numerous sensors sending multiple data points per second. Hence the write performance of the database should be swift.
- Flexible query performance: The specifications of data required for analytics by a data warehouse range from being narrow to wide. This requirement should be handled by the database. Querying should be flexible as well fast.
- Ease of scale-out: There should be an option to linearly scale out when required. Hence, without too much intervention or configuration by developers, the database should support linear scalability and high availability.

IoT Data Marketplace Service

IoT Data Marketplace Service will enable data monetization where users can provide Data as a service (DaaS). Marketplace wil

- Data economy
- Interoperability
- Improve Data Quality
- Crowdsourcing

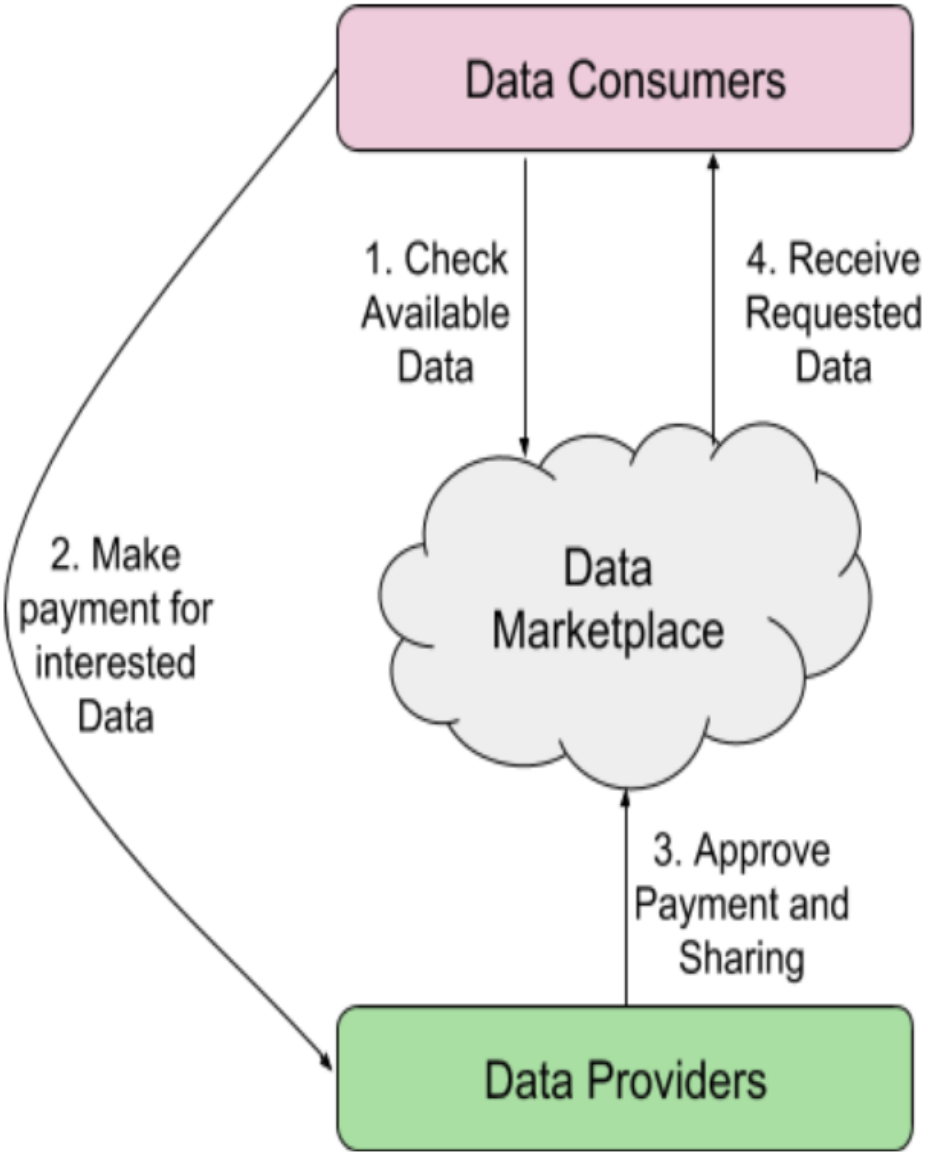


Figure 5 : Data Marketplace workflow

Services offered by an IoT Data Marketplace Service are as follows:

- **Data Discovery**
  - Secure, authenticated, authorised discovery framework for searching and discovering available data sources. This shall be enabled by the Catalog service
- **Data Following**
  - Secure, authenticated, authorised data following framework for requesting access for interested data sources
- **Data Payment**
  - Secure, authenticated, authorised payment framework for accessing interested data sources
- **Data Sharing**

  - Secure, authenticated, authorised data sharing framework for allowing interested users to obtain data

IoT Data Marketplace APIs

IoT data marketplace shall have endpoints to access services in a secure, authenticated, authorised, access controlled way. The r

- **Follow**
  - Follow endpoint in IoT data marketplace should provide necessary interface for showing interest to data sources through specific endpoints.
- **Payment**
  - Payment endpoint in IoT data marketplace should provide necessary interface for initiating digital payment between consumers and producers of data through specific endpoints. The interface shall be based on Unified Payment Interface (UPI)
- **Share**
  - Share endpoint in IoT data marketplace should provide necessary interface for allowing provider of data to approve interested parties to access data sources through specific endpoints.

2.5.3 Media Data Exchange

Cameras are amongst the key data resources for smart cities. They have the potential to enable smart applications that touch almost all aspects of daily urban life; automated surveillance, intelligent transport management systems, traffic management systems, smart parking systems, infrastructure monitoring to name a few. For example, networked IP cameras can be used to collect real-time data about vehicle counts, road congestions, accidents, vehicle breakdowns etc., which can help city officials manage the traffic better. The same video streams can be used to collect live road-side parking data that can provide better parking management systems. Further, the same camera feeds can aid security aspects by providing data about crowd build up etc. and help officials react faster to a growing situation in real time.

Currently, various smart city applications are operating as vertical silos with very little cooperation amongst each other. This problem is especially acute when it comes to sharing of video data. For example, the police department may have thousands of cameras across the city for surveillance, the traffic police may have hundreds of cameras at traffic junctions for traffic management and the transport department may have several cameras in bus-fleets, depots etc. with no way of sharing data with each other and/or with other public or private smart city ecosystem partners. Further, there may be many private producers of data that may be of interest to smart city operations, e.g., privately owned campus cameras, car fleet videos, public shopping arena videos etc. Sharing video data and the corresponding meta-data across various public/private entities will help break these silos and enable efficient and cost-effective operations.

One can view the combination of raw video/audio stream and corresponding analytics software as an intelligent sensor conveying specific information summarized from the input media content. Such meta-data streams containing data summaries are highly desirable as it allows downstream applications to avoid looking at raw media streams. In addition, one can use the meta-media streams to "correlate" data across multiple such and other IoT streams available from various entities present in a given spatial or temporal space. Sharing media data and the associated meta-media streams will accelerate ushering in an era of futuristic AI applications for smart cities.

To enable multiple such new applications, and especially by a multiplicity of different solution providers, it is critical that smart cities adopt an API based, platform approach towards exchanging and consuming video and media meta-data, similar to those for IoT data.

The following are the key requirements to create such a platform:

- [Media Meta-Data] A framework that supports associating a meta-data stream to raw media streams, that contain additional information of relevance to the semantic content of the raw media stream, and that enables easy discovery and additional value creation by downstream analytics applications.
- [Media Data Exchange] An API based, scalable, media data exchange platform, that allows for:
  - Easy discovery of media resources
  - Secure, authenticated, privacy preserving and accountable access for consuming live and archived media streams
  - Secure, authenticated and accountable access for supplying live and archived media streams

Meta Information for Video (and Media) streams

Value of data grows in proportion to the number of people who can use it. One critical necessary condition for successful usage of data, is availability of meta-data - i.e. data about the data. Example meta information include: time when data was captured, location from where it was captured, pose of the camera, etc. Such meta information adds a lot of value to the data itself and makes it easier to analyze. While the exact meta-data is application dependent and cannot be dictated in general, one can create a standard framework within which to capture and report the meta-data of media streams like video, audio etc. While many encoding formats already capture some meta-information, they are mostly about the encoding aspects. In addition to those, from the perspective of analytics especially, it is very useful to also have meta-data that is about the semantic content of the streams. This meta-data stream is adjoint (in parallel to) to the raw media stream as depicted in Figure 1. Some examples of meta-data content streams include:

- GPS/Pose time series data attached to media data for mobile media sources (e.g. for video streams from drones)
- Time series data of specific objects and their bounding boxes (e.g. times and number of diesel vehicles at a particular junction)
- Time series data about certain events (e.g. cows crossing the road)
- Semantic summary of the entire stream (e.g. live stream of an accident at south end '*'circle)

These meta-data streams can be generated by video-analytic softwares or by an agent that has access to other sensors. A key idea is to treat these meta-data streams also as any other IoT sensor stream. That is, these can be consumed via the platform and have an associated data-schema (that describes what information is contained in the stream) and this schema is stored in the resource catalog.

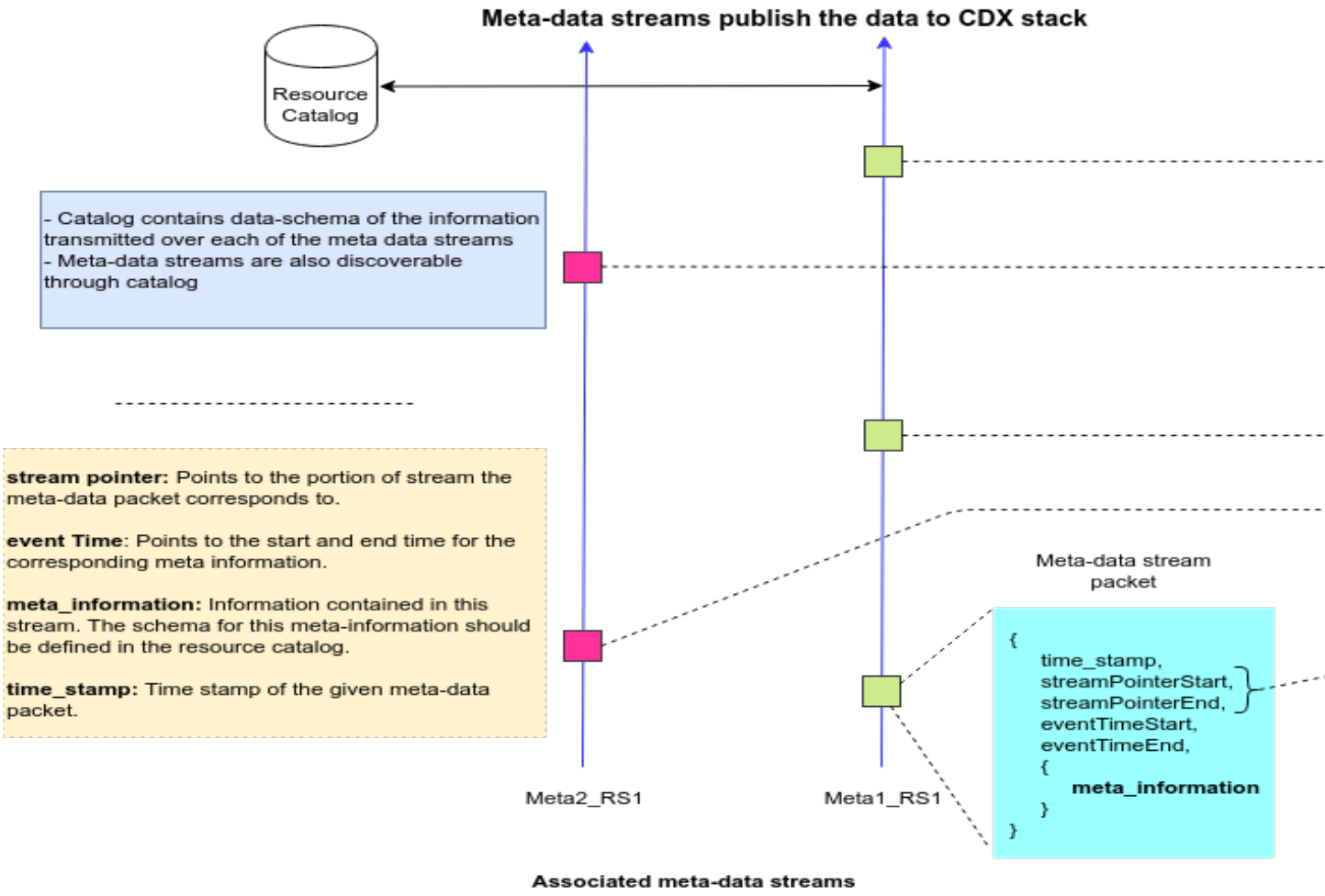


Figure 1 : IoT view of video data

An example schema for meta-information for a mobile camera is given below

```
{
"refCatalogueSchema": "meta-media-schema.json",
"id": "Meta1_RS1",//Unique Identifier for this Meta-Media stream
"resourceType": "meta-media stream",
"sourceId": "RS1", //Unique Identifier for source Media stream for which this meta
↪stream is for”
"tags": [
  "drone",
  "Camera",
  "Position"
],
"refCatalogueSchemaRelease": "0.1.0",
"Description": "Meta information for camera feed from Drone 80b3d58ff003AAe5"
"owner": {
   "name": "IISC",
   "website": "http://www.iisc.ac.in"
},
"provider": {
   "name": "Robert Bosch Centre for Cyber Physical Systems, IISc",
```

(continues on next page)

(continued from previous page)

```
   "website": "http://rbccps.org"
},
"data_schema": {
    "type": "object",
    "properties": {
    "onboard_analytics_msg": {
    "type": "object",
    "direction": "from-device",
    "access-modifier": "public",
    "tags": ["smoke detection", "location"],
    "properties": {
                "timestamp": {
                "type": "number",
                "description": "Time in EPOCH format",
                "units": "milliseconds"
                },
                "streamPointerStart": {
                "type": "number",
                "description": "ptr into the start of the media stream portion for this␣
↪meta info"
                },
          "streamPointerEnd": {
                "type": "number",
                "description": "pointer into the end of the media stream portion for␣
↪this meta info"
                },
                "eventTimeStart": {
                  "type": "number",
                  "description": "Start time for this meta info in EPOCH format"
                },
                "eventTimeEnd": {
                   "type": "number",
                   "description": "End time for this meta info in EPOCH format"
                },
                "latitude": {
               "type": "number",
               "description": "latitude as per WGS84",
               "ontologyRef": "http://www.w3.org/2003/01/geo/wgs84_pos#",
               "units": "degrees"
          },
                "longitude": {
               "type": "number",
               "description": "latitude as per WGS84",
               "ontologyRef": "http://www.w3.org/2003/01/geo/wgs84_pos#",
               "units": "degrees"
                },
                "smoke_detected": {
               "type": "boolean",
               "description": "Indicates if the smoke is detected in the video stream␣
↪or not "
                }
    }
     }
   }
}
}
```

Table 1: Schema to describe a Meta Video Stream which gives additional information about the geolocation of the camera at various time instances.

As shown in Table 1, the meta-media stream packets may contain broadly the following fields (in the dataobject section)

- timestamp: The time stamp at the time of packet generation in a pre-specified epoch
- streamPointerStart, streamPointerEnd: These fields provide a linkage to the source video stream. It specifies the parts of video stream this particular meta-data packet pertains to. These fields provide synchronization across various meta-information streams that are linked to the same video and will also help overlay meta-information with the display contents of the video. Further, it can help validation of meta-information that is generated by analytics software.
- eventStartTime, eventEndTime: These fields provide time indication for the start and end of the event/information described in this data packet.
- Meta-information fields: These contain the actual meta-information.

For the example of a flying UAV sending raw video data to the platform, since, the location of UAV is continuously changing it will be hard to ascertain the position of camera just by looking at the raw stream. However, the UAV can use its onboard GPS sensor and generate a meta-data stream which provides information about the position of UAV as time series data (Table 2)

```
"Onboard_analytics_msg": { "timestamp”:987565432768,"streamPointerStart":234615,
↪"streamPointerEnd":234616,"latitude":13.013846,"longitude":77.570311, "smoke_
↪detected": TRUE
}
"Onboard_analytics_msg": { "timestamp":987565432769,"streamPointerStart":234617,
↪"streamPointerEnd":234618,"latitude":13.023846,"longitude":77.571311, "smoke_
↪detected": TRUE
}
```

Table 2: Example snippets from a meta-data stream for a mobile camera source like a drone

Media Data Streaming Service

A conceptual diagram of the media data exchange platform is given in Figure 2.

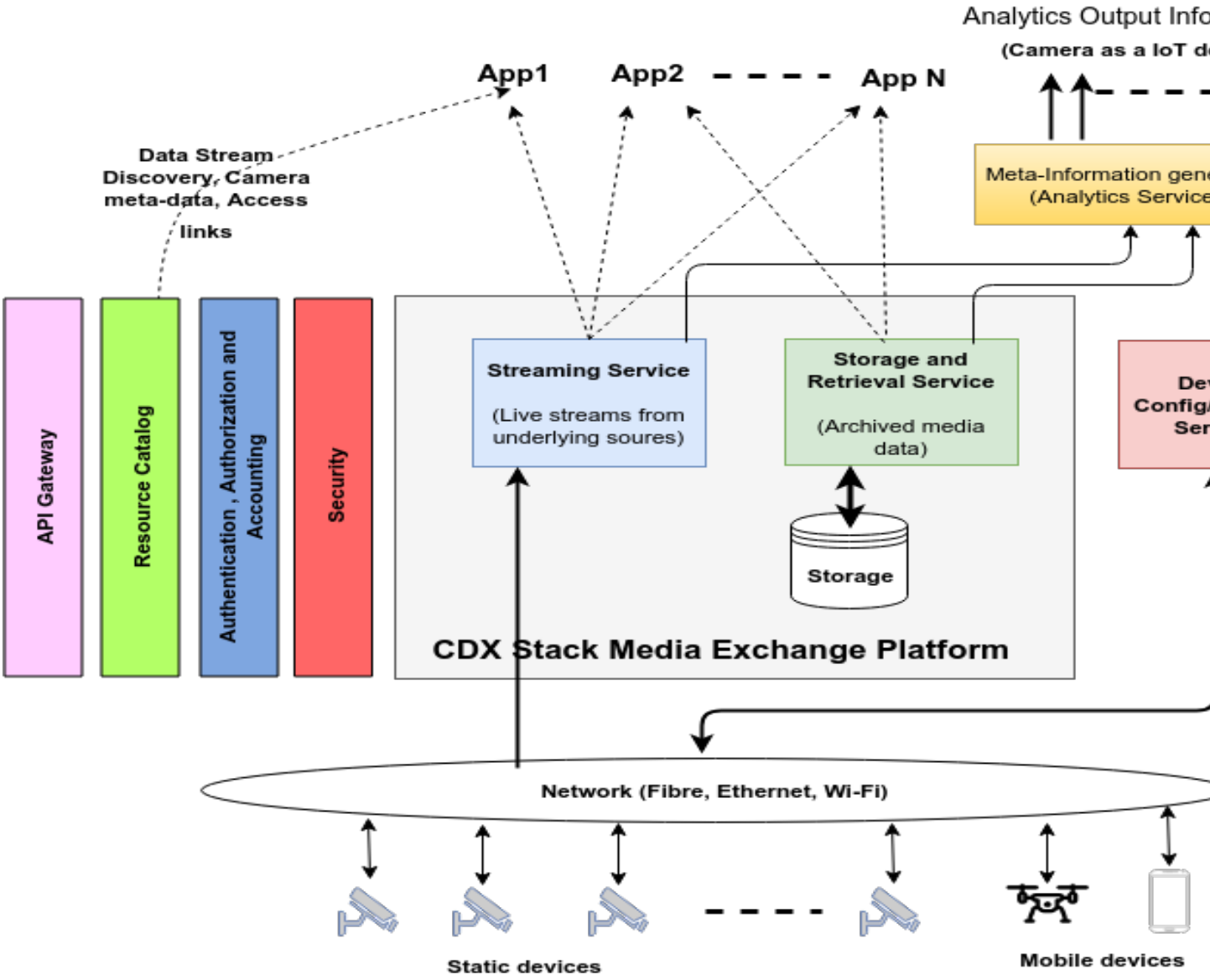


Figure 2: Media Data Exchange platform: Services view

Several registered, authenticated and authorized clients/applications should be able to publish and access live and archived media (video and/or audio) streams via standard media transport protocols and standard stream formats (e.g., RTMP, HLS etc. for live streams and MP4, AVI etc. for archived video). An emerging useful scenario is the ingestion of video streams and/or events associated with video data from the surveillance UAVs, where the data publishing is triggered by onboard analytics.

The available streams should be discoverable via resource catalog which may contain other useful information about the data source (camera) itself. Similar to IoT data streams other services can also be provided through the data exchange stack: flexibility to applications to pick and choose the streams they want to subscribe to, flexibility to the media device owners to share only appropriate feeds with others, for metering of video data consumed by a given application etc. For example, the police department, that owns several surveillance cameras, may want to share only some of the media streams (security non-critical feeds) with other entities.

At the core of the media exchange platform is the media streaming engine. The streaming engine ingests video streams from the installed cameras (and even mobile devices, e.g., drones, in future) and serves it to various applications. To

cater to diverse set of clients, a streaming engine may provide transcoding and re-formatting capabilities.

Note that the above figure is the conceptual view of media exchange platform. Today, most commercially available VMS platforms support device config/control services (via the well adopted ONVIF standard), storage/archiving and retrieval services. To some extent even streaming services may be supported. Thus, in actual implementations, a scalable, open API based VMS platform may be easily adapted to offer the streaming and archiving services envisioned by the media exchange platform.

Services offered by Media Exchange Platform

The media data exchange platform facilitates easy exchange of media-data across various smart city entities. The main services offered by the media exchange platform are as follows:

- **Distribution service: It provides**
  - Secure, authenticated, authorised and accountable (metered) access to the available media streams
  - Framework and mechanisms for defining sharing policies to facilitate media sharing across various entities
  - Framework for lifetime management of streaming sessions: For example, time based access, data limit based access etc.
  - Framework for privacy preserving presentation of media streams
- **Dynamic Ingestion service: It provides**
  - **Framework for creation and management of dynamic media channels by authorized entities**
    - Setting up stream parameters (data rates, data limits etc.)
  - Secure, authenticated and metered publishing of media streams over such channels
- **Archiving service: It provides**
  - **Authenticated recording and secure archiving of live media streams**
    - Framework to support various recording policies
  - Secure, authenticated search and on-demand retrieval and playback of archived media segments
  - Sufficient storage and backups provisioning for storing archived media data
- **Catalog service: It provides support for**
  - **Search and discovery of available media streams**
    - Meta-information corresponding to the associated media devices
  - Search and discovery of media-iot streams corresponding to a given media stream
- **Onboarding services: It provides**
  - Secure device onboarding and zero touch provisioning to configure security credentials, stream end-points and parameters (data limits etc.)
  - Secure onboarding and provisioning of meta-media stream sources

The media exchange platform security and AAA (Authentication, Authorization and Access control) requirements, that are imp

- Provide support for data encryption for both inbound (ingestion) and outbound (distribution) media data stream

- Provide framework for authenticated access and access lifecycle management to avail services offered by the media exchange platform
- Provide framework for authenticated and authorized sharing of media data amongst various entities

Further, some architectural requirements, which may guide specific implementations of the media exchange platforms and are desirable from the performance point of view in the context of smart city applications, are as follows:

- **Scalability**
  - Flexibility to add more streaming engines
  - Load balancers to offload access and publish client connections
- Support for variety of stream protocols and formats
- Support low-latency protocols for media streaming
- Support for transcoding, multiple bit rates and re-formatting

Detailed APIs for each of the above services will be documented. In particular APIs for:

- Secure device onboarding and secure onboarding and provisioning of meta-media stream sources
- Accessing live Media streams and meta-media streams for authenticated and authorized consumers, with metering support
- Publishing live media streams and offline media content for authenticated and authorized content producers, with metering support
- Searching of media stream sources, and Meta-Media sources and streams
- Data sharing APIs: Request for following and grant of sharing live & archived media streams and meta-media streams with authentication & authorization support.
- Recording and archiving media streams. Accessing archived media streams and meta-media streams for authenticated and authorized consumers with metering support

In Figure 2 above, we have included a 'Device configuration and management service' which, although an integral part of video sub-system, is not strictly a part of the media data layer. It provides secure and authenticated access to configure and manage underlying media devices (e.g., cameras) and is, thus, aligned to the operational aspects which are going to be handled by the operational-telemetry component of the stack. Further, since various public/private entities may deploy their own devices, these entities may not want to share the configuration and management functions with other entities. Thus, currently, these functions have been excluded from the media data exchange platform.

2.6 CDX Users

The CDX mainly consist of 3 types of users:

1. **Human users (or owners)** : who own devices and apps that are to be connected with the CDX.
2. **Subscribers** : The devices or apps whose main purpose to connect with the CDX is to subscribe for data.
3. **Publishers** : The devices or apps whose main purpose to connect with the CDX is to publish data.

Each owner after successful registration with the CDX is provided with:

1. **Owner's follow bus created at the CDX** : The purpose of this bus is to allow other users to request for accessing owner's devices/apps. Thus, this bus is writable by all valid users.
2. **Owner's follow queue created at the CDX** : This queue allows the owner to get the follow requests mentioned above. This queue can only be controlled by the owner.

Each subscriber after successful registration with CDX is provided with:

1. **Subscriber's queue** : This queue provides the subscriber with the messages from devices/apps which the subscriber has subscribed to. This queue can only be controlled by the owner of the subscriber.
2. **Subscriber's priority queue** : This queue provides messages of high priority and is typically used for low latency messages. This queue can only be controlled by the owner of the subscriber.
3. **Subscriber's control bus** : This bus allows other who have been given explicit write access by the owner of the subscriber to the bus to send control messages to the subscriber.
4. **Subscriber's control queue** : This queue allows the subscriber to receive messages sent to the Subscriber's control bus. This queue can only be controlled by the owner of the subscriber.

Each publisher after successful registration with CDX is provided with:

1. **Publisher's public bus** : A bus where the publisher can send messages which any valid user can read.
2. **Publisher's protected bus** : A bus where the publisher can send messages which can be read by a user to whom explicit permission has been given by the owner of the publisher.
3. **Publisher's private bus** : A bus where the publisher can send messages which can only be read by the owner or the publisher itself.
4. **Publisher's priority bus** : A bus where the publisher can send important messages, and can be read by an authorized user. This bus is used to publish low latency messages, and is typically subscribed using a priority queue of a subscriber.
5. **Publisher's control bus** : A bus where any authorized user can send control messages for the publisher
6. **Publisher's control queue** : A queue where control messages sent to the publisher can be read from.

2.7 License

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative

Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

2.8 Help and Support

- If you find any bug, have any trouble using CDX, or want to request for a feature, feel free to file an issue on Github at https://github.com/rbccps-iisc/ideam/issues
- You can also drop an email at smartcity@rbccps.org