ARL-TR-8870 ● Dec 2019

Fog Computing Platform Microservices Framework Description

by Barry Secrest, Brian Rapp, and Robert Amrein

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8870 ● DEC 2019

Fog Computing Platform Microservices Framework

Barry Secrest and Brian Rapp
Computational and Information Sciences Directorate,
CCDC Army Research Laboratory

Robert Amrein
Technica Corporation

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
December 2019	Technical Report	October 2018–September 2019

4. TITLE AND SUBTITLE
Fog Computing Platform Microservices Framework

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Barry Secrest, Brian Rapp, and Robert Amrein

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CCDC Army Research Laboratory
ATTN: FCDD-RLC-NC
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8870

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Internet of Things relies on centralized Cloud computing and highly reliable networking. When applied to the battlefield, the anticipated benefits to the warrior include increased battlefield awareness and Anticipatory Analytics that exploit the explosion of available data. The battlefield environment, however, is contested by the adversary through electronic and cyber-warfare and does not support a highly reliable network or Cloud computing. Through Fog Computing we are able to work in a contested network and provide the benefits of Cloud computing without a Cloud, using devices that are much lower SWaP (size, weight, and power) than traditional architectures required for big data, artificial intelligence, and other computation-heavy tasks. Fog Computing combines all battlefield compute resources to form a robust, resilient, distributed computational capability providing Cloud benefits closer to the edge. This report documents the framework for microservices in Fog Computing.

15. SUBJECT TERMS
distributed computing, Cloud services, edge computing, Fog Computing, computer architecture

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Barry Secrest
Unclassified	Unclassified	Unclassified	UU	20	19b. TELEPHONE NUMBER (Include area code) 410-306-1313

Contents

List of Figures iv

List of Tables iv

1. Introduction 1

2. FCP Background 1

2.1 Fog Computing 1

2.2 FCP Implementation and Overall Architecture 2

2.3 Microservices 3

3. Implementation Details 4

4. Sample Client 6

4.1 example_client.py 6

4.2 example_config.ini 9

5. Case Study 9

6. Future 10

7. Conclusion 11

8. References 12

List of Symbols, Abbreviations, and Acronyms 13

Distribution List 14

List of Figures

Fig. 1 FCP architecture..... 2

List of Tables

Table 1 Speech-to-Text microservice development phases..... 10

1. Introduction

Fog Computing Platform Microservices Framework (FCP-MF) is a high-level application programming interface (API) that allows developers to easily and quickly build Fog Computing Platform (FCP)-compatible microservices. It allows the developers to concentrate on the microservice functionality and not worry about lower-level details like sending and receiving messages to process, configure, encrypt, and log formatting that are essential for a well-running microservice but distract from the core implementation of the microservice.

Currently, the FCP-MF is developed as an abstract Python implementation for Python-based microservices but this can be expanded to other languages such as Java and C/C++.

2. FCP Background

FCP is designed to process Internet of Things (IoT) events in near real-time. Fog nodes are deployed close to the data sources to offload some of the analytics burden from the cloud. Faster results are obtained, and with less security risk, than transmitting all data to central servers for processing.

An intuitive web-based user interface provides a full suite of services for administrators to configure a microservice, deploy it to a fog node, monitor its status, start and stop the microservice, and send an updated configuration to a running microservice. These functions are initiated on individual nodes or on clusters of nodes.

The FCP architecture is powered by microservices: discrete software components deployed at the fog and edge layers to perform specific functions. They include support services such as databases and message brokers; also, analytic services such as neural networks and machine-learning algorithms.

2.1 Fog Computing

Due to increasing demands/deployments of IoT, the Fog Computing construct has gained considerable ground. In March 2018, National Institute of Standards and Technology (NIST) released Special Publication 500-325, the NIST Fog Computing Conceptual Model.[1] NIST describes Fog Computing as a mechanism to decentralize applications, management, and data analytics into the network itself using a distributed and federated compute model. Fog Computing can be contrasted with cloud computing in that, while the cloud is high in the air with pristine network connectivity and unlimited compute, the fog is closer to the ground—where IoT

sensors/devices live. Fog networks generally have intermittent and limited connectivity and also have reduced compute available. Fog Computing is especially relevant for tactical environments because compute, analytics, storage, and so on can be brought closer to edge devices/sensors operating in networks facing Denied-Disconnected Intermittent Limited bandwidth challenges.

It is important to note the Fog Layer is abstract. There is no one-size-fits-all model. Different use cases will require different configurations of Fog Nodes. Fog Nodes can be organized hierarchically, such that a lower-level node sends a condensed/subset stream of data to a higher-level node. For example, a Soldier's Fog Node could only communicate with the higher-level Fog Node on the Humvee. Or, the Soldier may not have a Fog Node at all—his/her sensors may talk directly to the Humvee's Fog Node. Additionally, it may be the case that the Soldier's Fog Node (due to reduced power, space, processing, etc.) may not be considered a tactical high-performance computing (HPC) device, while the Humvee Fog Node would be a tactical HPC device.

2.2 FCP Implementation and Overall Architecture

Figure 1 depicts FCP notional architecture and the publishing and subscription tasks that occur between the microservices and the broker.

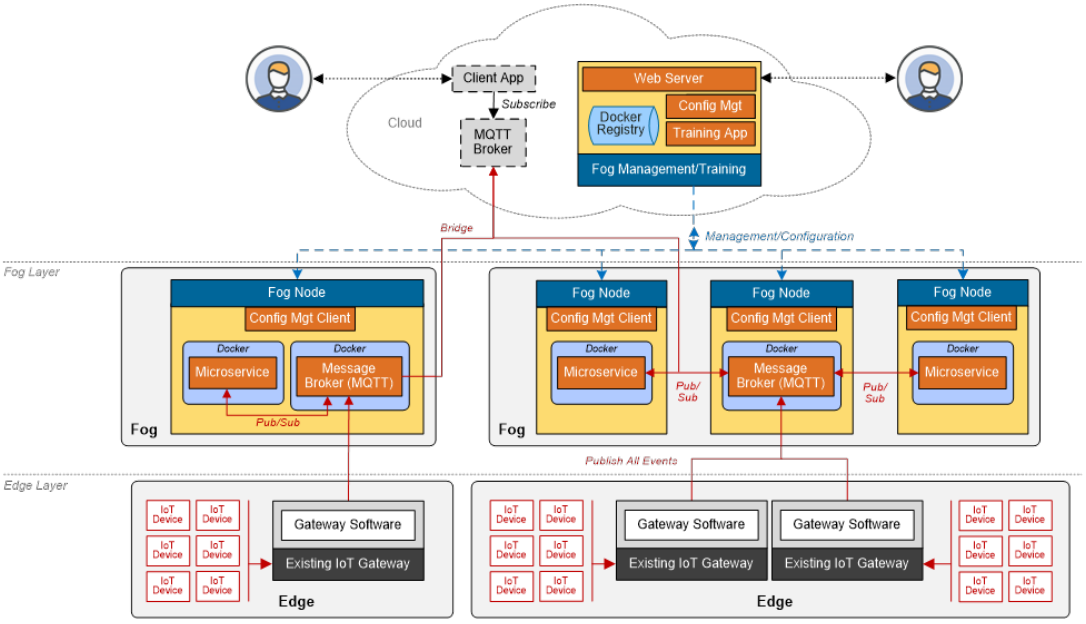

Fig. 1 FCP architecture

There are two types of end users shown in the diagram. The end user on the left represents users of a client application that makes use of the data coming from the edge and fog. The application listens for the data, processes them in some way, and

formats the data for users to see. The end user on the right side is a SmartFog administrator, who deploys and manages the microservices from the cloud.

Message Queue Telemetry Transport (MQTT) is the message-broker technology that enables the microservices to communicate with each other and between the edge and cloud layers. However, similar brokers, such as Advanced Message Queuing Protocol, can be used as well. While each device does not need its own MQTT broker, it must have access to a broker; therefore, each FCP deployment must include at least one message-broker server.

2.3 Microservices

Microservice Architecture (MSA) is a specific type of software development that concentrates on building single-purpose modules with well-defined interfaces and operations. The MSA paradigm has grown in popularity in recent years as the enterprise seeks to become more agile and move toward a continuous integration/testing pattern found in DevOps solutions. Additionally, many open-source projects such as Docker, Singularity, and SaltStack have facilitated MSA adoption. MSA can help create scalable, testable software that can be delivered daily/weekly.

MSA structures an application as a set of services based on business functionality.

In general, microservices have the following features:

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities

In addition, FCP microservices

- are easily configured, deployed, and maintained by an administrator,
- process input and output through a message broker,
- are containerized,
- continue operating while being disconnected from the cloud,
- allow resiliency with slow or unstable network connections,
- accept dynamic configuration updates, either manually from administrators or dynamically from higher-level systems, and

- communicate with each other over TLS-encrypted connections.

FCP-MF provides structure and convenience functions for building FCP-compatible microservices.

3. Implementation Details

The FCP-MF packages provide an abstract implementation of core microservice functionality to use when implementing microservices for the FCP. Microservices consist of a data MQTT client, an optional configuration MQTT client, data serializers and other various administrative functions that enables a unified interface for creation, logging and configuration of new microservices operating over the MQTT protocol.

The *fogms* package is for MQTT-based microservices that will retrieve their data from MQTT, do some processing on the data, and send their output messages to MQTT.

The *fogms* package provides the following five features to clients.

1) Defines a standard MQTT configuration object:

   This object details all parameters needed to create a secure connection to an MQTT broker. Developers can then ensure that these parameters are present in the microservice configuration. The configuration object also validates the parameters to safeguard against invalid values being passed into the microservice.

2) Connects to MQTT brokers and automatically registers *on_message* callback for received messages:

   After calling *connect_clients()* the connection to the MQTT broker will be established, and all topics defined in the *MQTTConfiguration* object will be subscribed to and linked to the *on_message()* method of the implementing object. A shutdown class is also provided to cleanly disconnect from brokers when the microservice is being terminated.

3) Registers the update callbacks if defined:

   Microservices can support dynamic configuration changes. They do this by listening on a specified update topic. This can be a separate MQTT broker that the microservice connects to for normal data processing. The framework facilitates managing this separate broker configuration, subscribing to the update topic, handling the new configuration, and managing reloading for the new configuration to take effect.

4) Standardizes log messages:

   Microservices use the standard Python logging mechanism but the framework standardizes the format of the log messages so they are consistent across microservices.

5) Provides a pluggable serializer for incoming and outgoing messages:

   Messages coming into and out of a microservice can be in a variety of formats. The pluggable serializer functionality allows clients to create a handler to convert to and from external data formats and the internal data structures required in the microservice. There is JavaScript Object Notation (JSON) Serializer available for use, but developers can write their own as well.

When inheriting the microservice, abstract base class developers must do the following:

1) Call *super() init* in the class initialization passing the parameters
   a. *data_mqtt_config* (*MQTTConnectionConfig*): configuration object for the data MQTT client
   b. *update_mqtt_config* (*MQTTConnectionConfig*, OPTIONAL): configuration object for the MQTT client to receive real-time config updates
   c. serializer (*fogms.payload_serializers*, OPTIONAL): serializer object used to conveniently serialize/deserialize payloads
   d. *logger_name* (*str*, OPTIONAL): name of logger level
2) Implement abstract methods
   a. *_on_message():* define default callback for topic subscriptions
   b. *_validate():* define validation method for microservice configurations
3) Call *connect_clients()* to make the connection to the MQTT broker and subscribe to topics for incoming messages.

These interfaces are written as a Python module, which makes it very easy to use in developing new microservice code by simply importing the appropriate module components.

4. Sample Client

The following passage is a simple Fog Microservice that shows how the FCP-MF should be used. This is a very simple service that connects to an MQTT broker, receives a message, and outputs a message on the configured topics. Section 4.1 shows the microservice code implemented in Python, and Section 4.2 shows the configuration file for the service.

4.1 example_client.py

```
from fogms import configurations
from fogms.microservice import microservice
from fogms.func._general_func import logging_format
from fogms.payload_serializers.JSONSerializer import JSONSerializer
from datetime import datetime
import logging
import argparse
import configparser
import sys
import time
import json

class SimpleService(microservice):
    '''
        Put any initialization code for the service in this method
    '''
    def __init__(self, data_mqtt_config, update_mqtt_config, serializer):
        super().__init__(data_mqtt_config=data_mqtt_config,
                         update_mqtt_config=update_mqtt_config,
                         serializer=serializer)

    '''
        Used to validate the configuration if necessary
    '''
    def _validate(self):
        #validate method - way to validate data members by type/value
        self.logger.debug("Validating....")
```

```
    '''
        Called when a message is received on the MQTT topic defined in the
        configuration
    '''
    def _on_message(self, client, userdata, msg):
        self.logger.info("Received message for processing from MQTT")
        self.logger.info("\tOn Topic: %s", msg.topic)
        self.logger.info("\tMsg: %s", msg.payload)

        #Service isn't really doing anything.  But this is where the
        #interesting pieces would go.
        #Do something interesting with the incoming message then create the output
        self.logger.info("Doing some important work here......")

        raw_msg = { "msg": "Hello World",
                    "success": "True" }
        json_msg = self.serializer.serialize(raw_msg, "data")

        #Send result to MQTT
        self.dataClient.publish_msg(json_msg)
        self.logger.info("Successfully sent results to MQTT")

def main():
    logger = logging.getLogger(__name__)

    #Read the config file location from the command line
    cl = argparse.ArgumentParser()
    cl.add_argument('--config_path', '-c', type=str,
                    default='./exmple_client.ini',
                    help = 'Full path to config file')
    cl.add_argument('--log_path', type=str, default='./',
                    help = 'Path to write log files')

    cl_args = cl.parse_args()
    log_file_name = datetime.now().strftime('example_ms_%d_%m_%Y')
    logging.basicConfig(format=logging_format(), level=logging.DEBUG,
                        handlers=[
```

```
                        logging.FileHandler("{0}/{1}.log".format(
                            cl_args.log_path, log_file_name)),
                        logging.StreamHandler()])

#get the config file and parse
logger.info("Parsing config from {}".format(cl_args.config_path))
config = configparser.ConfigParser()
with open(cl_args.config_path) as data_file:
    config.read_file(data_file)
    config_errors = False

#load MQTT configuration items from file into MQTT Configuration Object
try:
    mqtt_dict = {}
    mqtt_dict['host'] = config['mqtt']['host']
    mqtt_dict['port'] = config.getint('mqtt','port')
    mqtt_dict['pub_topic'] = config['mqtt']['pub_topic']
    mqtt_dict['sub_topics'] = config.get('mqtt','sub_topics').split(',')
    mqtt_dict['ssl_protocol'] = config['mqtt']['ssl_protocol']
    mqtt_dict['keepalive'] = config.getint('mqtt','keepalive')
    mqtt_dict['sub_qos'] = config.getint('mqtt','sub_qos')
    mqtt_dict['pub_qos'] = config.getint('mqtt','pub_qos')
    mqtt_dict['mqtt_protocol'] = config.getint('mqtt','mqtt_protocol')
    mqtt_dict['ssl_cert_path'] = config['mqtt']['ssl_cert_path']

    mqtt_c = configurations.MQTTConnectionConfig(c=mqtt_dict)
    mqtt_c.output_config()
except AssertionError:
    config_errors = True

#Instantiate and initialize the service class
jsonSerializer = JSONSerializer()
simpleMS = SimpleService(data_mqtt_config=mqtt_c,
                         update_mqtt_config=None,
                         serializer=jsonSerializer)

#Connect to MQTT and start listening for messages
```

```
    simpleMS.connect_clients()
    try:
        while True:
            time.sleep(1)
    except:
        simpleMS.shutdown()
        sys.exit(1)

if __name__ == '__main__':
    main()
```

4.2 example_config.ini

```
[mqtt]
host=x.x.x.x
port=8883
pub_topic=example/out
sub_topics=example/in
ssl_protocol=tls_12
keepalive=180
sub_qos=2
pub_qos=2
mqtt_protocol=311
ssl_cert_path=./certs/ca-chain.crt
```

5. Case Study

To get an idea of efficiency speed-up of using the FCP-MF in creating new microservices, this section will review the Speech-to-Text application that was recently converted to an FCP microservice. This was part of a larger effort to convert a speech-to-speech translation application and run it in the FCP, but will focus just on the speech-to-text portion of the workflow.

This developmental effort was done without the aid of the FCP-MF. Table 1 details the microservice-development process with the high-level tasks and associated effort involved.

Table 1 Speech-to-Text microservice development phases

Task	Effort
Research	72 h
Design	24 h
Development	48 h
Testing	48 h

Most of the time used in converting this functionality to a microservice was for learning and understanding the speech-to-text algorithms. For this effort we used Kaldi, which is a speech-recognition toolkit that uses machine learning. This had a large learning curve that accounted for most of the time used in the research phase.

However, we estimate the use of the FCP-MF would cut the development and testing time by 25%–50%. By using this toolkit, the developer no longer must worry about many of the details required to connect to and process messages from the FCP. Also, this cuts down testing time by reusing code that has already been thoroughly tested and used in other microservices.

6. Future

The FCP-MF has enormous potential for growth. Possibilities to increase its usefulness include the following:

- Expanded set of common APIs

  As the microservice catalog expands, additional common services could be extracted into the Microservices Framework (MF) and made available to all microservices. For example, microservice developers would find these APIs helpful:

  - Health Check Endpoint
  - Telemetry and Metrics
  - Automated Test Integration

- Additional language support

  Currently, the MF extensions are written in Python. However, microservices can be written in any language. As more microservices are developed in other languages, the creation of similar abstract

implementations in languages such as Java and C/C++ is a natural evolution.

- Other common functionality

  Additional modules can be created to handle other common tasks in microservices. One that might be especially useful would be support for machine-learning toolkits to do inference quickly and easily in a microservice.

7. Conclusion

The FCP-MF comprises APIs that allow microservice developers to focus on the business functionality of the microservice and not be concerned about common tasks like securely connecting to message brokers and receiving and processing messages.

The FCP-MF today provides a strong base set of functionality for microservice developers, but the expanded support of common APIs, languages, and machine learning envisioned for subsequent revisions would increase the usefulness and time savings for developers in the future.

8. References

1. Iorga M, Felman L, Barton R, Martin MJ, Goren N, Mahmoudi C. Fog computing conceptual model. Gaithersburg (MD): National Institute of Standards and Technology; 2018 Mar. NIST Special Publication No.: 500-325.

List of Symbols, Abbreviations, and Acronyms

API	application programming interface
FCP	Fog Computing Platform
FCP-MF	Fog Computing Platform-Microservices Framework
HPC	high-performance computing
IoT	Internet of Things
JSON	JavaScript Object Notation
MF	Microservices Framework
MQTT	Message Queue Telemetry Transport
MSA	Microservice Architecture
NIST	National Institute of Standards and Technology
TLS	Transport Layer Security

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	CCDC ARL FCDD RLD CL TECH LIB
2 (PDF)	CCDC ARL FCDD RLC NC B RAPP B SECREST