A Governance Architecture for Self-Adaption & Control in IoT Applications

Roger Young, Sheila Fallon, Paul Jacob
Software Research Institute, Athlone Institute of Technology,
Athlone, Co Westmeath
r.young@research.ait.ie, sheilafallon@ait.ie, pjacob@ait.ie

Abstract - The "Internet of Things" has become a reality with projections of 28 billion connected devices by 2021. Much R&D is currently focused on creating methods to efficiently handle an influx of data. Flow based programming, where data is moved through a network of processes, is a model well suited to IoT. This paper proposes a dynamic, distributed data processing architecture, utilizing a flow based programming inspired approach. We illustrate a distributed configuration management protocol, which coordinates processing between edge devices and a central controller. Our proposed architecture is evaluated in a vehicle use case that predicts driver alertness. We present a scenario for reducing data on vehicular networks when the connectivity options are limited, while maintaining computational accuracy.

Keywords: Flow Based Programming, Apache NiFi, Apache Minifi, Data Prioritization, Internet of Things

I. INTRODUCTION

The "Internet of Things" (IoT) is a paradigm in which sensors, actuators, and devices will have internet connectivity. Traditionally, the majority of data processing occurred on the cloud, or a central controller. However, sending large amounts of data over limited bandwidth makes the centralized data mining process infeasible. The introduction of edge/fog computing, where edge devices come with the capability to process and analyse newly generated data, has introduced scenarios that involves the distribution of some of the data mining tasks from the cloud to the edge.

Flow Based Programming (FBP), can be viewed as a technology where an application is constructed as a network of asynchronous processes exchanging data chunks and applying transformations to them. Although first created at IBM in the late 1960s, there has been a noticeable increase in technologies inspired by the FBP paradigm recently. Projects such as NoFlo [1], NodeRed [2], and Apache Nifi [3] have begun to focus on the strengths of FBP and the processing of data flows, which is a major requirement of the modern data-driven applications, thus making it a viable programming model for this oncoming paradigm shift.

One of the prime advantages of FBP is its modularity, meaning the degree to which a system's components may be separated and recombined. Nate Edwards of IBM [4] coined the term "configurable modularity" to denote an ability to reuse independent components just by changing their interconnections. A main characteristic of a system that exhibits "configurable modularity" is that you can build them out of "black box" reusable modules. While it is necessary to connect them together, they do not have to be modified to make this happen [5].

As previously mentioned, the cloud-centric approach is still the most common approach used. However, this approach is not sufficient where time-critical processing is required. Network bottlenecks and high latency are problematic in many scenarios. For this reason, much work is underway to efficiently move as much processing as possible out to the edge. Currently, there are limitations in regards to platforms for developers to deploy and execute generic applications on IoT edge devices. This work proposes a novel architecture that supports dynamic adaption of IoT applications based on internal and external events and conditions. This is achieved through a combination of the FBP model and custom functions implemented on both the edge and central container.

Our architecture acts as an ecosystem for developers to implement and manage generic IoT applications. Functions that perform real-time actions can be seamlessly incorporated or modified within our architecture. Parameters within the functions can be dynamically changed based on user input, local environmental conditions including, but not limited to, network connectivity, CPU and RAM usage, disk storage, etc. External factors such as weather or traffic activity may also impact the computation of the functions. FBP is advantageous in an IoT scenario due to its configurable modularity, as processes can be easily reconfigured and reconnected to adapt applications to different scenarios.

This work focuses on a connected vehicle use case. We evaluate a scenario whereby the vehicle can automatically send a subset of features to the central controller during low network connectivity. The central container can dynamically switch between a number of models, dependent on the incoming feature set. The dataset used for this work was initially proposed in a Kaggle competition called "Stay Alert! The Ford Challenge" [6]. The objective was to design a classifier that detects whether the driver is alert or not, employing data acquired from over 100 participants while driving.

This paper is organized as follows. Section II discusses related work, followed by description of a reference architecture in Section III. An overview of candidate technologies is presented in Section IV, followed by our implementation architecture in Section V. A scenario evaluation is presented in Section VI. Conclusions and future work are described in Section VII.

II. RELATED WORK

A NECtar Agent is proposed in [7], a solution that automates the switching between different data handling algorithms at the network edge. The aim is to provide a solution for network-edge data reduction and achieves accuracies between 76.1 % and 93.8 % despite forwarding only 1/3 of the data items.

[8] Examines the benefits of data mining on the wireless, battery-powered, smart sensing devices at the edge points of IoT. The authors implement three specific algorithms: Linear Spanish Inquisition Protocol (L-SIP), ClassAct, and Bare Necessities (BN). These algorithms fall under GSIP, or General Spanish Inquisition Protocol (SIP). Under SIP, nodes only send unexpected information. The goal of this work was to transform data at source into valuable information, in turn reducing packet transmissions, energy use, and storage space. Results showed packet reduction of between 95% - 99.98% demonstrating the importance of edge mining in an IoT environment.

Mobile Fog is proposed in [9]. MF is a high level programming model for IoT applications that are geospatially distributed, and latency–sensitive. The goal of this work is to ease deployment of IoT applications across multiple devices from the edge of the network to the cloud. It utilizes a dynamic node discovery process to associate devices together in a parent-child relationship. Mf is a hierarchical system that parent nodes lend their computation resources to process data received from child nodes. Due to its hierarchical system, MF supports load balancing between nodes while also allowing IoT applications to process data locally along the way from the edge to the Cloud.

Krikkit [10] is an open-source solution initiated by Cisco, but has been acquired by Eclipse. It is a publish/subscribe mechanism where rules are registered on the edge gateways that communicate with sensors. It is in the process of specifying a data format and a mechanism for "telling the network-edge devices" which data to forward and how. In [11] we propose a distributed data processing architecture for edge devices in an IoT environment. Our approach focuses on a vehicular trucking use case. The traditionally centralized Apache Storm processes such as calculating average speeds and aggregating driver errors are recreated on the edge devices using a combination of Apache MiNiFi and the user's custom-built programs. However, communication was one directional in this use case. Information was not sent from the central server to the edge devices.

III. SYSTEM DESIGN

This section discusses the dataflow that defines our reference architecture. The FBP model consists of three main components:

Black Boxes: Each black box, or process, in the application is an instance of a component that essentially receives some data, processes it and forwards the output to another black box, creating a dataflow.

Bounded Buffers: These are the connections between the black boxes. Black boxes are connected to one to another through ports defined by their components. The black box receives data through an input port and transmits the result through an output port.

Information Packets: The data that travels through the network, usually in the form of structured packets or streams of packets. They can be owned by only one black box at a time, which will either pass it along to the next process in the network or drop it.

Figure 1 illustrates the dataflow that connects the central container to the edge container. Information packets, in the form of control data, represented by a dotted line, are received from the service UI, external interfaces, or the main processing unit, and passed to the edge container. This information can influence the local computation.

Based on incoming information, the edge container may apply algorithmic calculations to incoming sensor data. This is performed by incorporating the user's stored functions or functions downloaded from the central container. The parameters within these functions can be dynamically changed by internal environmental conditions or external factors, including user input. Dependent on the scenario, different functions can be applied to the dataflow. It is also possible to run multiple functions asynchronously if necessary. The edge container returns the output to the central container, as represented by solid lines in figure 1.

On the central container, data is ingested through a specified communication port, before being routed to the main processing unit, comprised of the users more advanced programs, for further analysis. A service UI relays user requests into the dataflow, and is a means to view the output of the analysis. The configurable nature of FBP makes this architecture suitable for many use cases involving a large number of distributed connected devices, such as Points of Sales, Weather Detection Systems, Fleets of Vehicles and Network Systems.

The architecture also supports many other tasks including the following:

1) Separating time relevant data that needs to be processed instantly from data that may be batched and analysed at a later stage.
2) Structuring and transforming data while in motion.
3) Data encryption and compression.

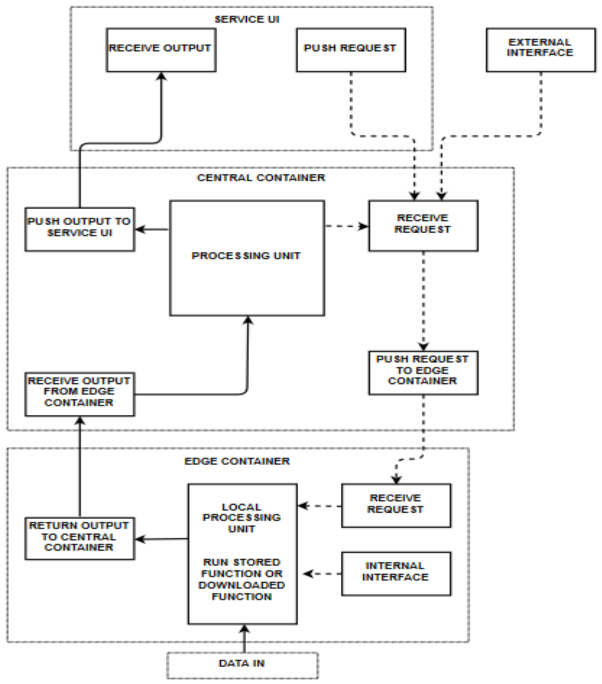

Figure 1: Reference Architecture with description of dataflow between Central & Edge Container

IV. TECHNOLOGY OVERVIEW

Apache NiFi [3], is a data in motion technology that primarily uses flow based processing. NiFi provides a user friendly GUI and contains over 200 processors. Each processor performs an action on the passing data. NiFi processors are likened to FBP black boxes. The user can create a real time dataflow by dragging Processors onto the canvas. Each processor is individually configured before connecting them to the following processor. The built in NiFi processors can perform a multitude of actions such as ingesting, transforming, merging, compressing, and routing data. There is a collection of processors available for ingesting data from a multitude of sources including URLs, ports, databases, local file systems, and external sources such as edge devices.

NiFi was created by the National Security Agency (NSA), and acquired by Hortonworks, a data analytics software company. NiFi addresses many of the technical challenges associated with IoT. NiFi adds extra security to the transportation of data with built-in support for SSL, SSH, HTTPS, encrypted content and role-based authentication/authorization and handles a diversity of datatypes as described above.

Apache MiNiFi [12] is a sub project of NiFi that can perform the majority of NiFi's actions. It is much more lightweight, just 40MB, and is optimized to perform on edge devices. MiNiFi does not have a UI, dataflows are created on the central NiFi server and downloaded onto the MiNiFi edge devices. Anaconda [13], a Python based Data Science distribution is used the build and load the machine learning models. Python codes are used to score the incoming data off the models, and perform computations on the edge containers.

V. SYSTEM SETUP

We evaluate a scenario in which data is continuously streamed from a vehicle to a central Nifi server. Figure 2 represents an instance of our reference architecture for this use case. Apache Nifi, installed on a central container, ingests data from the edge container (with Apache Minifi installed) and routes the incoming data to the Anaconda platform where it is scored against a trained model, predicting driver alertness. This prediction can trigger an alert to the driver if drowsiness is detected. In case of network connectivity dropping, another model is available to successfully score an incoming subset of features. Nifi dynamically switches between models, dependant on the incoming features from the edge container. This architecture also provides a method for data to be batched on the edge device and sent in bursts over known Wi-Fi locations. This is an effective solution as bandwidth over LTE is expensive.

Minifi was installed on a Raspberry Pi representing the connected vehicle. A dataflow consisting of multiple NiFi processors were installed via MiNiFi. A SplitText processor followed by a ControlRate processor can be configured by the user to ingest the data from the test dataset and transmitted set intervals, emulating the vehicle transmitting data in real time. An UpdateAttribute processor is configured to assign each feature an attribute name, which allows the data to be split and routed separately in the next step.

The ExecuteStreamCommand processor is a powerful and versatile processor that can run a custom program within the Dataflow. In this scenario, we created an algorithm that detects a change in network connectivity, and transmits data dependant on network strength. The algorithm is implemented through a Python code. If network connectivity is very high or connected to Wi-Fi, all features are transmitted to the NiFi Server. If network connectivity is below 50%, a priority group of features is transmitted. These features were chosen based on the work of [14].

The Nifi server ingests the data from the vehicles, where a RouteText processor, configured with regular expressions, forwards the data to the relevant model. An ExecuteStreamCommand processor calls another custom python program to score the incoming data against a model. The python code implements Scikit-learn and Panda libraries [15] to perform prediction against the model. The results of this prediction can be viewed through the service UI. If prediction is negative, an alert is sent to the driver or the fleet manager's phone via a PutEmail processor.

I. SYSTEM EVALUATION & RESULTS

A training dataset was used to build the model and a separate test dataset used to test the model. Different models were tested, with an ExtraTreesClassifier model giving us the highest prediction accuracy. For this evaluation, priority features were determined using results from [14], which performed statistical analysis on the dataset. The dataset used for this work consists of 30 features. Eight of these features are Physiological and are represented with a P, (P1, P2, P3 etc). 11 are Environmental, represented with E. 11 are Vehicular

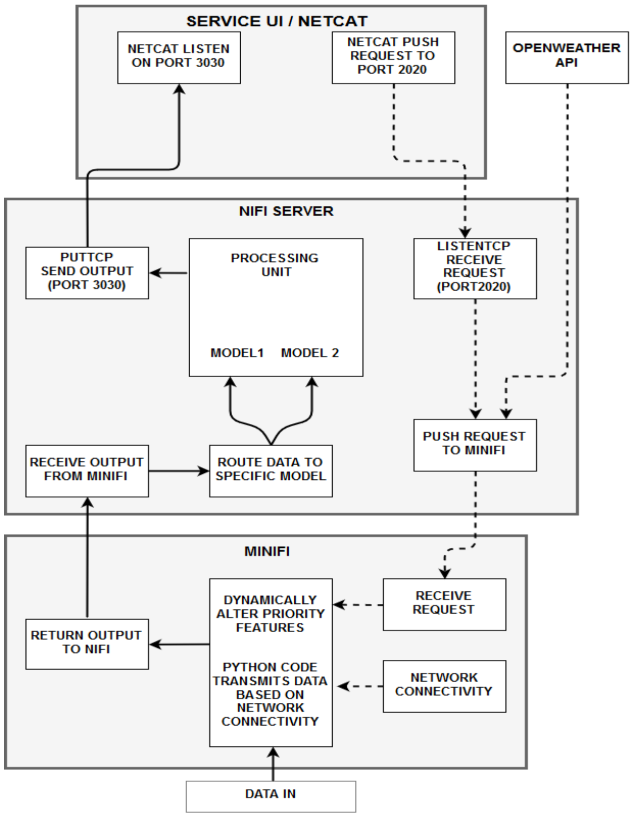
SERVICE UI / NETCAT
NETCAT LISTEN ON PORT 3030
NETCAT PUSH REQUEST TO PORT 2020
OPENWEATHER API
NIFI SERVER
PUTTCP SEND OUTPUT (PORT 3030)
PROCESSING UNIT
MODEL1 MODEL 2
LISTENTCP RECEIVE REQUEST (PORT2020)
RECEIVE OUTPUT FROM MINIFI
ROUTE DATA TO SPECIFIC MODEL
PUSH REQUEST TO MINIFI
MINIFI
DYNAMICALLY ALTER PRIORITY FEATURES
PYTHON CODE TRANSMITS DATA BASED ON NETWORK CONNECTIVITY
RECEIVE REQUEST
NETWORK CONNECTIVITY
RETURN OUTPUT TO NIFI
DATA IN

Figure 2: Implementation Architecture showing scenario 1 and 2 as described in Evaluation Section

features, and represented with V. For each observation, an output "IsAlert" is labelled with 1 indicating that the driver is alert or 0 if not alert.

To test our system, data transmission was recorded in two scenarios. This was achieved by increasing the control rate at which data passed through the edge device. The quantity of data produced was controlled by setting the granularity of data production to 100 milliseconds and 500 Ms. The table below shows a comparison between the cloud-centric approach in which all data is transmitted, and our approach during low network connectivity in which a subset of features are transmitted. The table represents data transmission over a five minute period.

Table 1: Comparison against cloud centric approach over a five minute period

Data Intervals	**Cloud Centric Approach**	**Dynamic Approach**	**Total Data Reduction**
100 ms	402 kb	99.6 kb	75.33%
500ms	80.6kb	19.9kb	75.69%

In our scenarios when network connectivity dropped, the priority features were transmitted, resulting in 75% reduction in data transmission, while still providing accurate predictions. In many cases, it may be optimal to only send the priority features at all times. However, many companies may want to receive and store all data when possible for future analytics. On a wider scale, data reduction on edge devices will play a pivotal role in the success story of IoT.

Currently, there are a number other IoT development platforms available such as NodeRed and Apache Edgent. However, NodeRed does not have a general way for configuring applications dynamically [16], and Edgent currently doesn't provide any "deployment" mechanisms. However, it does recommend to FTP the application to the device and modify the device to start the application upon start-up [17].

This next section will focus on the dynamic nature of our platform and how new functions and parameters can be seamlessly passed to the edge device, without breaking the flow of data processing.

Two scenarios were executed to evaluate the performance of the service UI and external interfaces dynamically changing the output from the edge container. In both scenarios, the list of priority features to be transmitted in low network connectivity were changed.

Scenario 1: The service UI
NiFi comes with two useful processors, ListenTCP and PutTCP. These can be configured to ingest data from a specified port and send data to a specified port respectively. By Using Netcat [18], a computer networking utility tool for reading from and writing to network connections using TCP and UDP, allows us to pass requests directly into the dataflow. In this example, a new list of priority features is transmitted to the edge container via the Netcat terminal. This was achieved by replacing the existing subset of priority features on the edge container dynamically with the list received from the users input. Figure 3 illustrates Netcat acing as a service UI, passing information through port 2020 into the dataflow, and listening on port 3030 for the output. This system has opened up many options for future work.

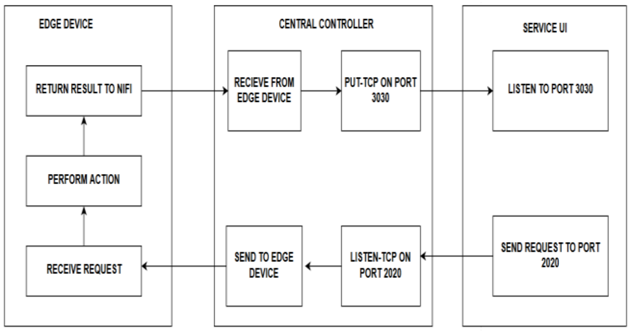

Figure 3: Illustration of Netcat acting as Service UI.

Scenario 2: External Interface.
As discussed, the functions on the edge container may be dynamically changed by external factors. To test this, a hypothetical scenario was set where the priority features are changed based on weather updates. A HTTP processor is configured on the central container to perform regular requests to an OpenWeather.API.
In this scenario, we implement a custom python code on the central container that generates a new subset of priority features whenever rain is predicted by the OpenWeather.API. These new features are sent to the edge container, updating the priority list. Figure 4 shows the protocol involved for this scenario.

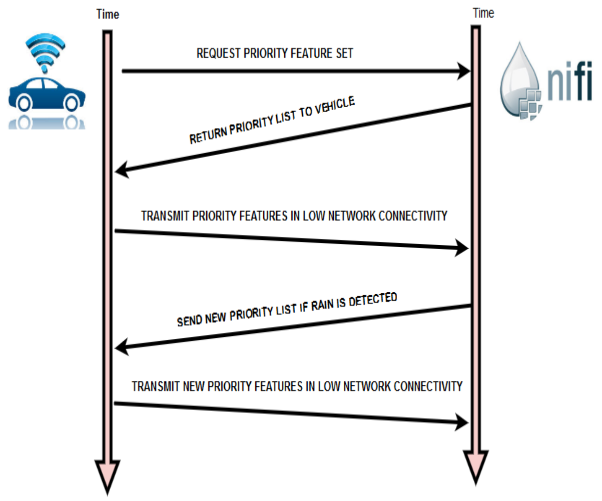
Time
Time
REQUEST PRIORITY FEATURE SET
RETURN PRIORITY LIST TO VEHICLE
TRANSMIT PRIORITY FEATURES IN LOW NETWORK CONNECTIVITY
SEND NEW PRIORITY LIST IF RAIN IS DETECTED
TRANSMIT NEW PRIORITY FEATURES IN LOW NETWORK CONNECTIVITY
nifi

Figure 4: Protocol showing dynamic change based on external factor (weather in this scenario)

II. CONCLUSION & FUTURE WORK

This work proposed a dynamic, distributed data processing architecture for an IoT environment. We evaluate a vehicle use case that predicts driver alertness. A dynamic protocol for adapting to network conditions is discussed. Two scenarios were evaluated that dynamically changed the priority feature set based on user input and external conditions.

Future work includes combining MiNiFi, NiFi and the user's custom programs to allow edge containers to perform more advanced data mining tasks such as real time prediction locally, without the necessity of transmitting to a central server. This can be achieved by implementing pythons Scikit-Learn packages on the edge containers. In a similar scenario to the use case provided in this paper, we can then build the models on the central container and distribute it to the edge containers. A dataflow can be created on the edge container that can independently score the incoming sensor data against the model. More advanced data mining tasks may also be performed on the edge, further reducing data transmission.

REFERENCES

[1] H. Burgius, "Noflo–flow-based programming for javascript," 2015. [Online]. Available: http://noflojs. org. [Accessed 1 october 2017].

[2] "Nodered.org," Node-RED, [Online]. Available: https://nodered.org/. [Accessed 1 October 2017].

[3] 451 Research , "Everything Flows: The value of stream processing and streaming integration," 451 Research, 2016.

[4] N. Edwards, "The Effect of Certain Modular Design Principles on Testability," IBM Research Report, NY, 1974.

[5] J. P. Morrison, "Flow-Based Programming: A New Approach to Application Development," Van Nostrand Reinhold, NY, 1994.

[6] "www.kaggle.com," Kaggle, June 2011. [Online]. Available: https://www.kaggle.com/c/stayalert#description. [Accessed 1 May 2017].

[7] A. Papageorgiou, B. Cheng and E. Kovacs, "Real-Time Data Reduction at the Network Edge of Internet-of-Things Systems," CNSM, heidelberg, 2015.

[8] Gaura et al, "Edge mining the Internet of Things," IEEE Sensors,volume 13, Coventry, 2013.

[9] Hong et al "Mobile fog: A programming model for large-scale applications on the internet of things," Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing. ACM, Stuttgart, 2013.

[10] "eclipse.org/krikkit/," Eclipse, [Online]. Available: https://eclipse.org/krikkit/. [Accessed 03 February 2017].

[11] R. Young, S. Fallon and P. Jacob, "An Architecture for Intelligent Data Processing on IoT Devices," IEEE, Athlone, 2017.

[12] nifi.apache.org, "nifi.apache.org/minifi," 18 December 2016. [Online]. Available: https://nifi.apache.org/minifi/. [Accessed 2016 october 2].

[13] "https://www.continuum.io/Anaconda-Overview," Continuum Analytics, 2017. [Online]. Available: https://www.continuum.io/Anaconda-Overview. [Accessed 30 June 2017].

[14] A. Sarkar, J. Kawawa-Beauda and P. Q, "Stay Alert!: Creating a Classifier to Predict Driver Alertness in Real-time," Stanford.edu, 2014.

[15] "scikit-learn.org," Python, [Online]. Available: http://scikit-learn.org/stable/. [Accessed 10 May 2017].

[16] Node-RED, "nodered.org," nodered.org, October 2016. [Online]. Available: https://flows.nodered.org/flow/6fe183c197b3464a1fe4d89744e068ff. [Accessed 3 September 2017].

[17] The Apache Software Foundation, "edgent.apache.org," Apache, 2017. [Online]. Available: https://edgent.apache.org/docs/application-development#option-1-create-an-uber-jar-for-your-application. [Accessed 11 September 2017].

[18] "sourceforge.net," sourceforge, [Online]. Available: http://nc110.sourceforge.net/. [Accessed 10 june 2016].