# Fog at the Edge: Experiences Building an Edge Computing Platform

Nam Ky Giang*, Rodger Lea†, Michael Blackstock‡, and Victor C.M. Leung*
*Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, Canada. Email: {kyng,vleung}@ece.ubc.ca
†School of Computing and Communications
Lancaster University, Lancaster, UK. Email: rodger@comp.lanc.ac.uk
‡Sense Tecnic Systems Inc, Vancouver, Canada. Email: mblackstock@sensetecnic.com

***Abstract*—Technology advancement has pushed computation to the network edge, paving the way for a class of IoT applications that leverage CPU, storage and communications in edge devices. Building these new IoT applications is not an easy task however. Two key challenges are supporting the dynamic nature of the edge network and the context-dependent characteristics of application logic. In this paper we report our experience in building an edge computing platform called Distributed Node-RED (DNR) that uses a distributed data flow programming model based on the popular open source Node-RED tool. We describe some of the challenges we faced as well as some novel solutions that were implemented in our platform. A new approach in applying the concept of exogenous coordination is also presented and shown to be necessary in building large-scale IoT applications across the edge, fog and cloud.**

***Keywords*-edge; fog; computing; exogenous; dataflow**

## I. Introduction

As the number of connected Internet of Things (IoT) devices increases, the amount of data generated by these devices is putting pressure on the traditional centralized cloud computing infrastructure. Fog computing, where resources are distributed closer to the edge network, has emerged as a system model to support many data-intensive or delay-sensitive IoT applications [1]. Thanks to its widely distributed nature, fog computing infrastructures are often able to process the large amounts of data produced by IoT applications at the edge network rather than transporting raw data streams to the distant cloud.

While "edge-ward" computing infrastructure offers a more efficient and timely system model for IoT applications, combining and leveraging sparsely distributed computing resources across the edge network, fog and cloud, is not an easy task. The large-scale characteristics of these systems and the complexity of the applications suggest that we rethink both the application model and development process. The requirements are essentially twofold: how do we decompose the application so that it can be easily distributed from the cloud to the edge network; and how do we support the characteristics that are inherently relevant in the edge network, such as the large-scale, the dynamic nature and the context-dependent nature of computation?

To gain experience and begin evaluating IoT applications in fog systems, we have leveraged the open source project Node-RED [1], which provides a visual dataflow programming paradigm for building IoT applications. We developed extensions to Node-RED and built an open-source platform, *Distributed Node-RED (DNR)*.

Our first version of DNR, originally introduced in [2], allowed developers to build distributed IoT applications that spanned multiple devices from the edge network to the cloud. While this system could distribute Node-RED applications or *application flows* across multiple devices, it did not address the need to deploy multiple instances of software components to a range of devices; nor did it adequately support the dynamic nature of edge devices and the context-dependent nature of application logic. In a subsequent iteration of the platform, DNR v2, we addressed these requirements and solved a number of challenges that arose as we improved our programming model. We developed several fog applications using DNR v2 which we report on in this paper and use to offer a number of lessons that we feel are valuable for the community as a whole. To support further experimentation within the community, we have made our system, DNR v3, fully open source and available online[2]. Throughout the paper, we use the term *fog applications* to denote the IoT applications that involve computing resources across the edge network toward the cloud, with a focus on edge devices that are IP reachable and have sufficient resources to support a minimal run-time.

## II. Fog/Edge Computing

Today's Internet infrastructure could be seen as a three-tier distributed system [3] that includes a cloud layer, a communication network and a client layer. Internet applications have been scattered around these computing elements for many years with backend, frontend and content delivery/caching logic. For the most part, the development of such Internet applications still follows a manual process where developers build individual pieces of the application and the communications in between.

[1] https://nodered.org/
[2] https://github.com/namgk/dnr-editor

978-1-5386-7238-9/18/$31.00 ©2018 IEEE
DOI 10.1109/EDGE.2018.00009

Building IoT applications in this distributed manner is even more difficult as they often involve moving components and have a close bond to the physical world. While Distributed Systems have been extensively studied, with a plethora of distributed application models, it is not always clear how to apply them to these new requirements. There are two distinguishing characteristics that challenge the application development process: large scale geographic distribution of the computing infrastructure and the dynamic nature of edge devices.

### *A. Large Scale Geographic Distribution of Computing Infrastructure*

Large-scale fog computing systems may range from industrial IoT spanning factories, to smart city or regions spanning metro areas of larger geographical groupings. This large-scale geographic distribution has several consequences that influence the way applications are developed.

First, the fog computing resources are generally communicating over a heterogeneous network that involves 1) different communication mediums (e.g Wi-Fi, LTE, Wired, etc) and 2) a mix of static and dynamic endpoints with different reachability (e.g direct IP addresses vs behind a NAT). This heterogeneity makes inter-device communication more difficult. In large-scale Fog applications, these communication details should not hinder the development of the application in general. Therefore, the developers should be provided with enough programming tools and primitives that allow them to focus only on the application logic.

Secondly, due to the large-scale distribution of computing resources, their physical location, or more generally their physical context, becomes an important factor in the fog computing application model. For instance, if there are many smoke detectors and robot fire extinguishers distributed over a large area, a particular smoke detector instance might signal only *nearby* robot extinguishers. That is, in addition to specifying the location where an application component should be deployed, a developer might want to restrict the interaction between two components based on their relative locations.

### *B. Dynamic Nature of Edge Devices*

Due to the close bonding with the physical world, edge devices often exhibit a highly dynamic nature, in terms of both load fluctuation and context changes (e.g. location changes when edge devices are mobile). While load balancing and dynamic scaling is commonly used in cloud computing to cope with variations in application load, it is more difficult to do the same at the edge. This is partly because edge resources are not as centralized and readily available as cloud computing resources. The other reason is that the heterogeneous networking environment at the edge makes it difficult to locate resources for load balancing.

In addition to varying load, changes in physical context such as location plays an important role in IoT applications, requiring a certain level of context monitoring and situational re-evaluation. The involvement of dynamic physical context also leads to the question of how to expose the context information to programming primitives or constructs to application developers. Returning to our previous example of smoke detectors and robot extinguishers, the application data in this case is the sensing data generated by the smoke detectors, such as the level of $CO_2$ in the air. In order to communicate or send this data to the appropriate robot extinguisher component, the contextual data, their locations, are used to coordinate the communication, i.e to route sensing streams from the detectors to the appropriate robot(s).

## III. Early Experiments with the Dataflow Programming Model

In our first attempt to build an application platform for fog applications we only focused on solving the problem of distributing the application across the edge devices, the fog and the cloud. Dataflow-based programming languages appeared to be a natural fit for the need for application decomposition and distribution, and seemed very suitable for a large number of IoT applications[4].

Node-RED is a popular dataflow-based visual programming tool and language for IoT applications. Applications are developed by dragging and dropping processing nodes onto a canvas and 'wiring' the nodes together. The wires represent communication paths between nodes. As can be seen (fig. 2), this is a direct visual analogy for the DAG mentioned above. The resulting application, referred to as a "flow", is then 'deployed' to run as single process on a single device, i.e. Node-RED has no support for distribution or large-scale programming.

Our initial goal was to leverage Node-RED to experiment with the partitioning of fog applications so that they can be easily deployed into the distributed computing infrastructure across the edge network, fog and the cloud. We created the first iteration of our Distributed Node-RED project where we extended Node-RED in a number of ways.

We introduced the notion of *device* to the dataflow language. Accordingly, every node in a dataflow program is augmented with a new *device Id* constraint that specifies on which device the node should be deployed and run. For example, a node can be constrained to be deployed on an edge device, a mobile host, a cloud server or on any intermediary device across the edge to the cloud.

The second augmentation made to Node-RED was the notion of "remote wires" or "remote arcs". Since the nodes may run on separate devices, the existing Node-RED wires have to support inter-device communication to handle the situation where a flow is broken up and its nodes are distributed to several devices. This is implemented using

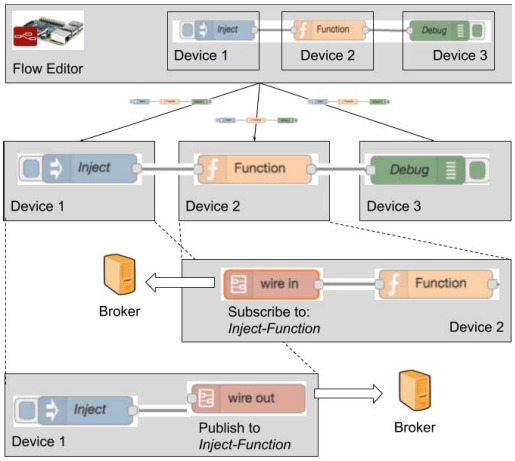


Figure 1. Initial Distributed Node-RED

a publish/subscribe communication mechanism that binds the nodes together. The key idea is to leverage the node identifiers on each end of a wire as the publish-subscribe topic. To implement this, a flow transformation process is applied so that the nodes that do not run on the current device host are replaced with a *wire in* or a *wire out* node. The *wire in* node subscribes to a communication broker so that it can receive data from the external node running on a different device. A *wire out* node receives data from the local node and publishes it to a communication broker so that the *wire in* nodes on other devices will receive it. Figure 1 illustrates this process of supporting the distributed deployment of a Node-RED flow across multiple devices.

Based on early experiments with DNR, we found that the Distributed Dataflow programming model offers a natural basis to decompose or partition an application so that it can run across multiple devices from the edge network toward the cloud. By hiding the details of remote communication, developers did not have to worry about the communication details, thus making the development process of distributed fog-based applications more seamless.

However, in order to build more sophisticated and larger scale applications, we found a need to support more complex deployment requirements and a need to be able to deploy flows and nodes to multiple devices in parallel. For example, when deploying to multiple devices, we found that the notion of *device Id* was too limiting. If we want to configure a node to run on a specific location, we have to incorporate that information into the *device Id*, e.g *"office-laptop", "home-laptop"*. That is, we found the need to deploy nodes based on more complex requirements, and to have multiple instances of nodes running on multiple devices at the same time. These requirements are the basis for our work on our second iteration DNR v2.

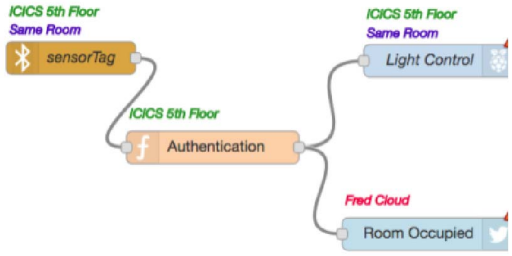


Figure 2. Annotating nodes with constraints

## IV. DNR v2: Supporting More Complex, Larger Scale Applications

DNR v2 was built based on the lessons learnt from using DNR v1 to develop fog applications. The main motivation was to support applications at a larger scale that has a vast number of participated devices, and with a larger deployment scope, such as across a city. In DNR v2, we addressed two major issues: support for more complex deployment constraints of the *application flow*; and the deployment of multiple instances of devices running the same sub-flow.

### A. Constraint-based Application Flow Distribution

As described above, one key limitation of our DNR v1 was the simplistic nature of the mapping from nodes to devices. Our experience with DNR v1 highlighted that the mapping needed to be much more sophisticated, for example:

- A sensor node mounted on a moving vehicle could be restricted to operate in a certain location.
- A vision processing node might be restricted to operate in a more capable computing device.

To address these needs, we introduced the *constraint* primitive as a broader abstraction that specifies how a node is deployed and run in a distributed computing setting. Accordingly, every node in a dataflow program is augmented with a *constraint* property that defines how the deployment is carried out. In our project, a *constraint* involves the requirements on device identification, computing resources such as CPU and memory and physical location.

The goal is to make the application model more suitable for a class of fog-based applications that are heavily dependent on the context associated with the edge devices they operate on. As a result, the developer can not only specify which type of device a node should run on (e.g mobile, server or laptop, etc) but can further constrain where the node should run based on a variety of aspects such as memory size, processing capability, location etc. To address this need, we added support to allow application developers to specify these node constraints via the programming user interface. (fig. 2).

Table I
TOPICS CONSTRUCTION

| Wire Cardinality | Communication Topic |
|---|---|
| 1-1 | <An instance of node A>_<An instance of node B> |
| N-M | <node A>_<node B> |
| 1-N | <An instance of node A>_<node B> |
| N-1 | <node A>_<An instance of node B> |

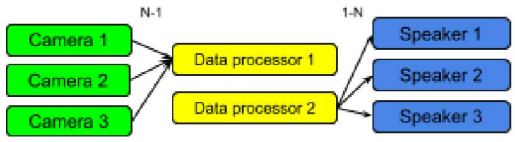


Figure 3. Wire Fragmentation

### B. Replicated Deployment of Nodes

With the new *constraint* primitive, we were able to experiment with the replication of nodes on multiple devices. For example, in the simplified application flow described in the last section, the Function node can now have a constraint such as *"run within the location X (e.g. in Vancouver)"* instead of being constrained to only *device Id*. This is significant because, while a *device Id* can only specify a simple deployment constraint, mostly based on a particular device identification, a *constraint* primitive can capture more sophisticated requirements. This means that DNR v2 is now able to support large-scale deployment of fog-based applications that involve a large number of devices and a programming pattern where applications flows and their associated nodes are replicated across many heterogeneous devices. As identified earlier, this is a key requirement for large-scale IoT applications. Interestingly, there are several inherent challenges in supporting this new deployment capability.

*1) Wire cardinality:* First we have to add the notion of *cardinality* to the wires. That is, since a node can run on multiple devices, there may be multiple instances of a given node at runtime, each hosted by a different devices. An initial implication of this capability is that there has to be a cardinality relationship between nodes that mandates how their runtime instances communicate. These include 1-1, 1-N, N-1 and N-M.

Table I summarizes these different cardinalities and their associated communication topics. Specifically:

*N-M wire cardinality:* , the communication happens freely between Node A and Node B and that the participating devices do not have to know about one another. The topic for inter-device communication is as simply as *<node A>-<node B>*. For the sake of readability, we now omit the node Id in a node's representation (e.g instead of writing *node A's Id*, we only write *node A*). Topic construction for this special case is the same as in our first version of DNR.

*1-N or N-1 wire cardinality:* , one instance of Node A can multicast to a number of instances of Node B. However, each instance of Node B can only accept data from one particular instance of Node A. In this case, the topic is constructed as *<an instance of node A>-<node B>*, where *an instance of node A* is a combination of node A's Id and the Id of the device it is running on. A similar reasoning is applied to the N-1 *wire cardinality* case, in which the constructed topic will be *<Node A>-<an instance of Node B>*.

*1-1 wire cardinality:* , both the devices running Node A and Node B have to know the particular instance of the other peer in order to construct the topic. In this case the constructed topic will be*<an instance of node A>-<an instance of node B>*.

To support these instance-specific topic construction, when a device participates in the DNR application, it periodically provides information about its' physical context and the nodes it can run, to the coordinator. The coordinator in turn keeps a repository of node instances with corresponding constraint information. When an instance-specific topic needs to be constructed, the device asks the coordinator which node instance to use and the coordinator answers to those requests based on its node instance repository. We refer to this as a *node instance request*.

*2) Wire Fragmentation:* With the new *constraints* abstraction and the associated notion of *wire cardinality*, we are faced with a *wire fragmentation* problem. The *wire fragmentation* problem occurs when there is more than one instance of a data processing node available across the runtimes and the system inadvertently configures itself to have all data sources communicate with one instance, but all data sinks communicate with a different instance, i.e. the natural flow (or wiring) from source, via processing, to sink is fragmented.

Consider an intrusion detection and alert system which involves security cameras, data processor systems and speakers for alert purposes. Accordingly, multiple instances of the camera sources send video streams to an instance of data processors for intrusion detection, and the potential alerts are broadcast to the speakers.

Fig. 3 illustrates the process of forming a distributed deployment of this application. When the devices running Camera nodes want to construct the topic for inter-device communication, since this is a N-1 *wire cardinality*, they request the coordinator for an instance of Data processor node by sending *node instance requests*. The coordinator, based on its node instance repository, picks an instance of the Data processor node that satisfies the deployment constraints and sends it back. A similar process happens when the devices running a Speaker node want to form a N-1 inter-device communication with a particular Data Processor node.

The wire fragmentation problem occurs when there is more than one instance of the Data Processor node in the

system. That is, even though the coordinator's assignment of these node instances can satisfy all the constraints, the instances being assigned could be different. This results in a situation where all Camera node instances send their video streams to one instance of Data Processor node and all Speaker node instances get the alert notification from a different instance of Data Processor node, i.e. the wire is broken or fragmented. This symptom also occurs when devices fail or move to different locations.

To overcome this problem, we developed *link score*, a quantitative measure that indicates the connectedness of a specific node. With this new primitive, every node has a forward and a backward *link score*. Whenever the centralized coordinator assigns a node instance to a *node instance request*, it updates the *link scores* for the node instance being assigned in both forward and backward direction. In subsequent *node instance requests*, it chooses the node instance that has the highest *link score* to minimize the *wire fragmentation* problem.

### C. Experience with DNR v2

DNR v2 exhibited several novel solutions to some of the challenges in building fog applications, such as constraint-based deployment primitives, multitude deployment of nodes and link score. However, to fully support edge devices in fog applications, the platform has to be able to cope with the highly dynamic characteristics of many typical edge devices. For example, there are some major use cases in fog applications that involve the mobility of the host device, such as in vehicular applications. In such cases, the deployment of nodes becomes more dynamic as the edge device moves from place to place, which may at times enable or disable the execution of certain nodes. Based on this location change, the communication pattern among nodes from different devices also needs to be coordinated properly.

To better understand the needs of large-scale fog applications, we developed a series of prototype deployments for smart city transport applications that focused on intelligent vehicles. These are reported in more detail in [3].

One application that we explored is to leverage video streams from dashcams mounted on cars to deliver real-time traffic status. The video streams can be processed by local road-side units mounted on lamp posts and the results are distributed back to nearby cars. It could also be used to actuate some smart transportation elements, such as traffic lights, to be recorded for data analysis purposes, or to interact with social media. This type of application involves several independent data processing components that are linked together in a dataflow-type of architecture. It also involves the need to produce, in a timely manner, an outcome from the video inputs in order to actuate some physical processes, e.g. smart traffic lights.

This example application illustrates two new challenges, which DNR v2 was not able to support fully. Firstly, the need to synchronize with the changing context in a highly dynamic and large-scale system. As described above, vehicles in a transportation network are highly dynamic and their context, such as location, but also resources and capabilities, are constantly changing as different groups of vehicles pass through specific locations in the city. This points to a need for a management (or co-ordination) system that is able to evaluate, in real time, the current status of the city-wide system and make decisions about where nodes should run.

Secondly, a related problem is the coordination of the communications among components. Once we are able to support highly dynamic systems where processing components are constantly reconfigured to meet changing context constraints, then we need to address the co-ordination of the communication amongst them. We refer to this as *inter-component constraints*.

These two requirements influenced the evolution of our DNR platform as we began to consider how the communication among components is coordinated and how that might require an external coordination layer.

## V. DNR v3: Toward Exogenous Coordination for Fog-based IoT Applications

### A. Towards an Exogenous Coordination Platform

Coordination models and languages have long been used to support the communication constraints of Distributed Systems applications and to coordinate the interplay among distributed computation activities [5]. For larger scale systems, where "programming in the large" comes to the fore, the separation of concern between computation and communication has led to a need for exogenous coordination models and languages [6].

In exogenous coordination, applications consists of participating computation activities (software components) which cooperate with one another to fulfill the application's logic. These software components can be independently developed by different developers and have no knowledge of their environment. Thus, they should not have external dependencies other than having to conform with the coordination protocol. To incorporate these different software components together in a single application, a coordination layer is needed to coordinate the communication among components. To do this, the software components usually declare their communication ports such as input and output ports. They then take data from the inputs, do the computation and place the result on the outputs, where it will be taken by the coordination layer and routed to other appropriate components. Everything related to communication such as buffering, queueing or rerouting is taken care of by the coordination layer.

We suggest that an exogenous coordination model is necessary in building an application model for large-scale, geographically distributed fog computing systems. When the computation and communication aspects of the application

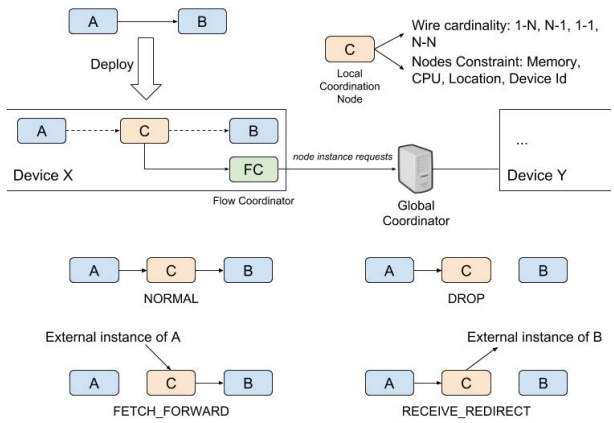


Figure 4. Exogenous Coordination in DNR

is explicitly separated, it is possible for the applications to leverage off-the-shelf software components and dynamically adjust to the constantly changing physical context of the underlying computing infrastructure. The application development process is then reduced to simply designing the components architecture and specifying coordination requirements, without the need to change code.

### B. Exogenous Coordination in DNR v3

As we introduce the notion of exogenous communication to our DNR platform, some major design decisions have been made that replace aspects of the old design.

One particular aspect is how we control the execution of each node among devices. When an *application flow* is deployed to the participating devices, the nodes could be replaced with *wire in* or *wire out* nodes depending on the deployment constraints. When supporting the dynamic nature of the system, and in particular, by building an exogenous coordination platform, this design choice no longer works. This is because the decision of whether or not to run a particular node is not permanent and at times is made by an external coordinator, not the device itself.

Our solution is to intercept the communication links among nodes and regulate these communications by injecting to every wire a special node that we developed, the coordination node. This could be seen as a different design of remote links in comparison to DNR v1 and v2. These coordination nodes receive coordination controls from a flow coordinator that runs locally. The job of this flow coordinator is to synchronize the device context with the centralized coordinator and receive coordination control messages from the centralized coordinator. Upon receiving coordination control messages from the centralized coordinator, it updates the coordination nodes. Consequently, the coordination nodes are the one who directly do the coordination. There are four coordination states that these coordination nodes can be in. Fig 4 illustrates this process.

In the NORMAL state, these nodes just pass the data through its outputs, thus acting as a normal wire between two nodes. This is the state where the local device satisfies the deployment constraints of both the Node A and B.

There is a DROP state where nodes drop all the data they receive. This is useful when a local coordinator believes the data will have already been consumed elsewhere. For example, Node A does not have any constraint so all devices can run it, however the local device cannot run Node B. Since all devices can run Node A, the coordination node can assume Node B is already in another device that has data input, so it just drops the data.

Another state is the FETCH_FORWARD state, where it acts as a *wire in* node to get data from an external instance of Node A and forwards the data to Node B. To recall, this happens when the device can run Node B but not Node A.

Lastly a similar state with FETCH_FORWARD is the RECEIVE_REDIRECT state, where it acts as a *wire out* node to send data from Node A to an external instance of Node B. Again, this happens when the device can run Node A but not Node B.

In the last two states, there is an extra piece of information that has to be determined in order to construct the state: the destination to which the data is to be sent to, or the origin of the data to be received. The coordination nodes use the *wire cardinality* of the communication link they are currently coordinating to decide the data destination. If the *wire cardinality* is N-M, they do not need to specify a particular node instance for the communication, thus the target is just the node's identity (see Section III). If the *wire cardinality* is instead 1-1, 1-N or N-1, they need to specify a particular instance of external node for the data. The process of obtaining this information is described in Section III. Once the destination is obtained, it is included in the state construction.

From time to time, the state of each coordination node might change, allowing them to dynamically coordinate the communication among nodes in our DNR platform. Generally, there is a 50% chance the participating devices have to send out a *node instance request* to the centralized coordinator. For example, if the *wire cardinality* is 1-N, but the node state is RECEIVE_REDIRECT, the coordination node does not need to send this request because it should redirect the data to all other external devices. Similarly if the *wire cardinality* is N-1 but the node state is FETCH_FORWARD, it should fetch data from any external device, thus need not send a *node instance request*. In the N-N state, it never has to send out these requests. However, in 1-1 state these requests are always needed.

Another important aspect relates to how the coordination layer resolves the node constraints and its response to *node instance requests*. Based on the context synchronization mechanism, the coordination layer should have a global view of the whole system. However, the search space could

potentially be very large. Currently we are experimenting with using a constraint solver to resolve the node constraints in our platform. There are several inherent challenges with regard the adoption of an off-the-shelf constraint solver. Firstly, the solver has to be executed periodically to yield a correct coordination execution as the system is dynamic. Second, the solver has to be able to keep track of previous solutions and minimize the change in responses to *node instance request*. This is because if previous solutions are not reused, existing communications, while still meeting all the constraints, might be rewired unnecessarily leading to a high communication overhead and inefficiency as the system reconfigures itself.

Interestingly, the constraint solver also has to derive as many non-overlapped solutions as possible due to the large geographic distribution of the system. This is because when there are a large number of node instances that are scattered over a large area such as a city, there are multiple combinations of these node instances that are useful in different places. For example, there could be multiple combinations of *[dashcam, image processing and visualizer]* serving in multiple locations of the city. The constraint solver has to derive as many of these combinations as possible without any overlapping instances.

## VI. Lessons learnt

Through the development of DNR and our effort in making DNR a comprehensive platform for building fog applications, the challenges we have faced have highlighted a number of interesting lessons.

### A. Programming-in-the-large Mindset Required

From our experience to date, we have found that there is a separation of concerns when building large-scale fog applications which are not present when building similar applications deployed in a centralized cloud. The first concern is how to implement the application logic, and the second is how the system will be deployed *in-the-large*. The system will not only run on a single device or server, but across cloud servers, gateways and edge devices. This involves writing components that encapsulate functionality, can be easily distributed, and can communicate with other required components in various ways as discussed in this paper. The second and more important concern is the need for the fog developer to specify how the system as a whole decides where groups of components, that is, sub flows should be split, run and how they communicate with each other. In essence, the developer needs to adopt a programming-in-the-large mind-set, one that involves specifying constraints such as location, computing and network requirements, replication and cardinality requirements. The system can then use dynamic information from the physical environment such as the quality of network communications, current location, current power levels and other factors to decide what sub-flows are deployed where. The combination of these two concerns, expressed in two different programming models that work together, is key to the development of fog applications.

### B. Exogenous Coordination necessary for Dynamic Fog Systems

At a general level, a dataflow language can be seen as having a coordination model with a clean separation of computation and communication activities [7]. At the simplest, it acts as a configuration-oriented coordination model [8] with static assignment of nodes and links. It also exhibits a simple form of exogenous coordination as the computation and communication is explicitly separated. However the dynamic nature and the complexity of the fog system demand a more comprehensive coordination platform to coordinate the communications among software components. One example is to support *inter-component constraints*, which requires a global view of the whole system with constant situation re-evaluation instead.

### C. A Stateful Coordination Layer is Necessary

In order to properly coordinate the communications among nodes in a large-scale, dynamic distributed system, the coordination layer has to be able to keep track of the system as it evolves over time. That is, the constraint solver has to derive the minimum difference based on the change of the system context in order to reduce the coordination overhead as much as possible. This is particularly important in large-scale, more complex systems which incorporate many heterogeneous edge devices.

### D. Dataflow Programming Maps Well to Fog Applications

The dataflow programming model offers a natural way to decompose and partition an application into independently developed, off-the-shelf components, which can be distributed to a large-scale computing infrastructure. The dataflow programming model also provides a means for abstracting the complexity of inter-device communication, an important characteristic when it comes to building large-scale, complex applications. While it provides many advantages, applying dataflow programming in large-scale fog applications has several challenges such as *deployment constraints*, *wire cardinality* and *wire fragmentation*. We solved these problems by modeling the *constraints* with computing resources and location of a device; introducing *link score* and a special publish/subscribe topic construction method. This approach lends itself to a wide range of use-cases with differing requirements.

## VII. Related work

As a new computing system, fog computing with its own characteristics, requires an appropriate application model. Some recent attempts to address these problems have been

demonstrated by Hong et al. [9], Olena et al. [10] and Bin et al. [4]. Hong et al. proposed a high-level programming model that allows the developers to specify how the application can be deployed to many fog computing devices and supports dynamic scaling of the computing resources. This work demonstrated that the application model itself should be aware of the underlying computing platform on which the application is running. On the down side, the proposed model does not explicitly break the application into sub components, which make it impossible to have different deployment strategies without changing the application logic. Olena et al. proposed that fog applications consisting of many services, that have their own computation demands, are placed into the fog computing system in a way that satisfies the deadline requirement of the applications. While this work pushed the componentization of the application code and studied the placement of those components, it does not take into account the contextual information of data. Bin et al. proposed a containerized dataflow-oriented application models with external configurations that specify how the data flows through the fog computing nodes. However, the coordination mechanism is still simple, based on only deployment scope and the inter-node communication is also limited to either broadcast or unicast fashion.

Several coordination models and languages exist to support the development of edge-ward distributed application such as [11], [12]. Although they provide a means for application decomposition and distribution, they lack support for context-dependent logic or *inter-component constraints*.

In our work, contextual information is incorporated into the application model, allowing the developer to express application-level constraints based on the physical context of the computation activities (i.e. software components, dataflow nodes). DNR also has support for the dynamic nature of the system, allowing mobile edge devices such as in vehicular applications to actively participate in the application.

## VIII. Conclusion

In this paper, we describe our experiences in building an edge computing platform: Distributed Node-RED. Through the three iterations of our project, we explored a number of challenges associated with developing fog applications that span across the edge network to the cloud. Several novel solutions have been introduced that were incorporated into the platform. We also show that in developing fog applications, exogenous coordination provides a reusable and scalable application model thanks to the explicit separation of communication and computation activities.

## IX. Acknowledgements

This work has been partially funded by NSERC (IPS application ID 486401) and the EU H2020 BigClout project (Grant Agreement N723139)

## References

[1] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, "A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1–1, 2017.

[2] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung, "Developing IoT Applications in the Fog : a Distributed Dataflow Approach," in *Internet of Things (IOT), 2015 5th International Conference on the*. IEEE, 2015, pp. 155–162.

[3] N. Giang, V. Leung, and R. Lea, "On developing smart transportation applications in fog computing paradigm," in *DIVANet 2016 - Proceedings of the 6th ACM Symposium on Development and Analysis of Intelligent Vehicular Networks and Applications, co-located with MSWiM 2016*, 2016.

[4] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities," *IEEE Internet of Things Journal*, vol. 4662, no. c, 2017.

[5] G. A. Papadopoulos and F. Arbab, "Coordination Models and Languages," *Advances in Computers*, vol. 46, no. C, pp. 329–400, 1998.

[6] F. Arbab, "Composition of interacting computations," *Interactive Computation: The New Paradigm*, pp. 277–321, 2006.

[7] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, 2004.

[8] G. A. Papadopoulos and F. Arbab, "Configuration and dynamic reconfiguration of components using the coordination paradigm," *Future Generation Computer Systems*, vol. 17, no. 8, pp. 1023–1038, 2001.

[9] K. Hong, D. Lillethun, B. Ottenwälder, and B. Koldehofe, "Mobile Fog : A Programming Model for Large Scale Applications on the Internet of Things," in *The second ACM SIGCOMM f (MCC '13)*, 2013, pp. 15–20.

[10] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards QoS-Aware Fog Service Placement," *Proceedings - 2017 IEEE 1st International Conference on Fog and Edge Computing, ICFEC 2017*, pp. 89–96, 2017.

[11] R. Sen, G. C. Roman, and C. Gill, "CiAN: A workflow engine for MANETs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5052 LNCS, pp. 280–295, 2008.

[12] A. Lombide Carreton and T. D'Hondt, "A Hybrid Visual Dataflow Language for Coordination in Mobile Ad Hoc Networks," in *Coordination Models and Languages: 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, vol. 6116 LNCS, Berlin, Heidelberg, 2010, pp. 76–91.