Note: this workshop paper has been superseded by a publication at the IEEE IoT conference:
Developing IoT Applications in the Fog: a Distributed Dataflow Approach
Nam Ky Giang, Michael Blackstock, Rodger Lea, Victor C.M. Leung. Procs. of the Internet of Things (IOT), 2015 International Conference on the, Seoul, Korea, Oct 26-28, 2015

Toward a Distributed Data Flow Platform for the Web of Things

Michael Blackstock
Human Communication Laboratory
University of British Columbia
Vancouver, Canada
mblackst@magic.ubc.ca

Rodger Lea
School of Computing and Communications
Lancaster University
Lancaster, UK
rodger@comp.lancs.ac.uk

ABSTRACT
Several web-based platforms have emerged to ease the development of interactive or near real-time IoT applications by providing a way to connect things and services together and process the data they emit using a *data flow* paradigm. While these platforms have been found to be useful on their own, many IoT scenarios require the coordination of computing resources across the network: on servers, gateways and devices themselves. To address this, we explore how to extend existing IoT data flow platforms to create a system suitable for execution on a range of run time environments, toward supporting distributed IoT programs that can be partitioned between servers, gateways and devices. Eventually we aim to automate the distribution of data flows using appropriate distribution mechanism, and optimization heuristics based on participating resource capabilities and constraints imposed by the developer.

Categories and Subject Descriptors
D.2.6 [**Programming Techniques**]: *Graphical Environments, Integrated Environments,* C.2.4 [**Distributed Systems**]: *Distributed applications,* D.1.3 [**Concurrent Programming**] *Distributed Programming.*

General Terms
Performance, Design, Languages

Keywords
Visual data flow languages, toolkits, Web of things, Internet of things

1. INTRODUCTION
Many common IoT scenarios in areas such as home and industrial automation, real time business intelligence, and interactive spaces require integration with online services and (near) real time sensing and actuation. While it is possible to create real time interactive IoT applications using traditional programming tools, it can be difficult, requiring developers to learn new protocols and APIs, create data processing components, and link them together. Today practitioners are creating new tools and platforms to ease the development of this class of IoT applications. Some of these systems focus on connectivity and routing [1, 13], while other use a 'condition action' paradigm [3, 8] to connect services and things. To provide more flexibility while maintaining ease of use, several platforms [4, 5, 7, 11, 14] provide a data flow programming paradigm where computer programs are modeled as directed graphs connecting networks of 'black box' nodes that exchange data along connected arcs. This simple paradigm, while not necessarily expressed visually, lies at the heart of several visual programming languages. Visual data flow programming languages (VDFPLs) [9] have been used in many other domains such as high performance parallel computing [2] leveraging multi-core processors, music [10], toys [15] and industrial applications [4]. Two web-based data flow systems in particular; the WoTKit Processor [5], and Node-RED [11], begin to address interactive IoT scenarios like these. The Processor is a multi-user system for running data flow programs in the cloud while Node-RED is toolkit for developing data flows on devices and servers.

While data flow platforms have proven useful on their own, many IoT scenarios require the coordination of computing resources hosted on things and servers across the Internet: hosted in the cloud, gateways and at the edge on smart devices. Devices with attached sensors can change raw sensor readings to send "presence" events for example. Gateways can aggregate data from several sensors and perform some simple data processing; cloud-based systems can connect to online services for real time access to social network feeds and alert channels such as email and SMS. We believe that by providing a distributed data flow run time for the IoT where data flows can be hosted on a variety of platforms, developers can more easily make use of these computing resources across the network.

In this paper we provide an overview of data flow architectures and describe the WoTKit and Node-RED systems in some detail, comparing and contrasting these systems toward the design of a distributed data flow system called WoT Flow. We then outline our proposed distributed architecture, and describe an early prototype system we've developed based on Node-RED.

We then provide context for this work by describing existing interactive web-based platforms that can be used in IoT applications, highlighting their key characteristics. We conclude

Pre-print version V3: Final version available from the ACM: http://dl.acm.org/citation.cfm?id=2684432

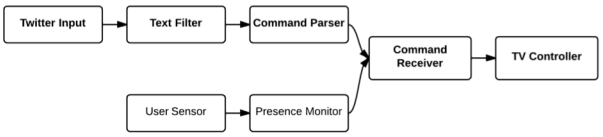

Figure 1. Simple data flow connecting Twitter and a presence sensor to control a television.

with some open questions for participants at the workshop to inform the design of WoT Flow.

Our contribution is an exploration of how to extend existing IoT data flow systems to create a platform suitable for execution on a range of run time environments, and support data flows that can be partitioned manually. Eventually, using an appropriate flow distribution facility and optimization heuristics, we aim to support the automatic partitioning and distribution of data flows based on participating resource capabilities and constraints imposed by the developer around cost, performance and security.

2. IoT DATA FLOW ARCHITECTURES

Research in data flow architectures dates back to the mid-70's when researchers wanted to exploit the massive parallelism offered by parallel processors [9]. While traditional programs are typically represented as a series of instructions written using text, and focusing on the operations that are to be executed, data flow programs lend themselves to a visual representation where nodes representing data inputs, outputs and functions are connected with arcs that define the data flow between components as illustrated in Figure 1. Since these operations are stand alone, they can be easily re-used and composed in other programs. The dataflow semantics of VDFPLs can be more intuitive for non-programmers

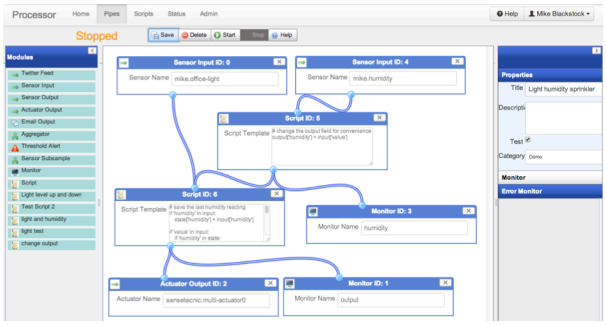

Figure 2. WoTKit Processor browser-based pipe editor.

A pipe management page lists the pipes that the user has created, indicating their execution status (error, executing, stopped). Users can start, stop and edit their own pipes from this page; they do not have access to other users pipes executing on the platform. To develop a new pipe, users drag and drop modules to the main canvas, and connect them as illustrated in Figure 2.

After saving a pipe, a user can start a pipe's execution in the editor or on the management page. The system then checks the pipe for errors, and "compiles" the pipe by instantiating pipe modules in the server, and adding wires to a global 'routing' table that links module endpoints together. Once all modules are instantiated, the system calls 'start' on each module. On start, input modules connect to external systems such as Twitter or the

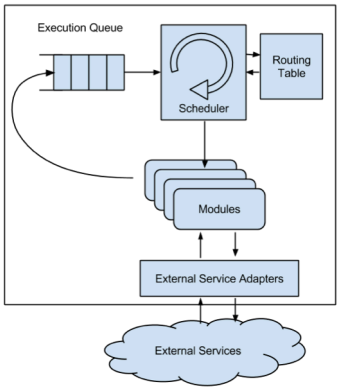

Figure 3. WoTKit Processor Architecture

to understand and lend themselves to programming environments where users can arguably more easily move between program design and implementation [9], and reduce development time [4]. The two systems described here provide such an environment for building real time IoT applications.

2.1 WoTKit Processor

The WoTKit Processor is a multi-user service bundled with the WoTKit platform [5] that allows users to process sensor data and react to real time updates from sensors and other external systems. The WoTKit processor is implemented in Java using the Spring Framework, and leverages the Java Concurrency Framework in its execution engine. Like other data flow systems, the primary interface is a browser-based visual data flow editor. Inspired by Yahoo Pipes [14], users of the Processor create data flow programs called pipes made up of *modules* connected with *wires*.

WoTKit sensor data aggregation platform [18], subscribing to appropriate real time data streams.

The Processor uses a multi-threaded execution scheduler to process data as it arrives at the system for all users as illustrated in Figure 3. Input modules connected to external services add *tasks* to the execution queue consisting of a received data message and an instance of the input module. The scheduler waits on this queue, and retrieves the next task, calling the associated `process()` method on the associated module with the new data. The module implementation may add additional tasks (messages and associated modules) to the execution queue, or send the data to an external service itself.

To stop the execution of a pipe, the system looks up the modules on the specified pipe and calls `stop()` on each module. This allows endpoint modules to unsubscribe or disconnect from external services. It then removes the wires from the routing table in the system, leaving other modules and wires in the system for other pipes running. This architecture allows pipes to be managed and controlled independently and by changing the execution queue implementation, control the priority of pipe execution between users.

The WoTKit Processor includes input and output modules to send and receive data to external systems such as WoTKit-hosted sensors, twitter feeds and email. A monitor module is used for testing and debugging pipes, allowing users to watch data as it flows through the system. Alerting is supported with an email module to send emails when interesting events occur. An actuator module, allows users to control things from a pipe. Function modules for data aggregation and thresholds are

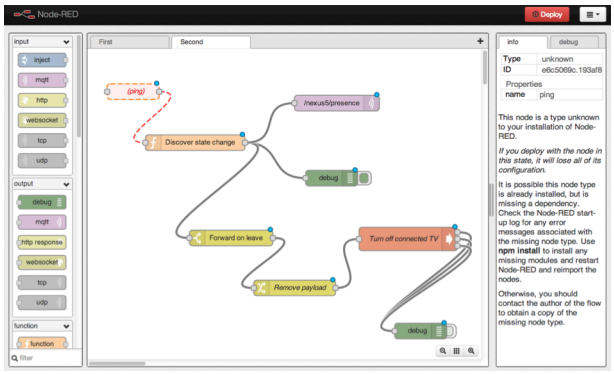

Figure 4. Node-RED browser-based flow editor.

included. Finally, the Processor includes the ability for developers or users of the system to write their own modules. Developers can add new modules in Java by implementing JavaBeans and registering them with the Processor database. End users of the system can implement their own modules in the UI using Python and save copies of these scripts for use in other pipes.

The Processor's pipes are expressed using JSON documents as shown in Figure 5. This example pipe takes input from a Twitter feed called *myeverydayearth,* does some processing on the message, then sends the processed output to a WoTKit sensor. The visual editor on the browser generates these JSON representations, which are sent to the server for storage and execution.

2.2 Node-RED

Like the Processor, Node-RED is a web-based tool for connecting hardware devices and APIs. It also provides a browser-based flow editor (Figure 4). It is implemented in JavaScript using the Node.js framework, taking advantage of Node's built in event model, and native support for JavaScript on both the client editor and the server.

Data flow programs on Node-RED are called *flows* consisting of *nodes* connected by wires. Like the WoTKit Processor, the user interface consists of a flow editor with node templates on the left that can be dragged and dropped into a flow canvas. Unlike the Processor, all users accessing Node-RED manage a single flow, which may be shown on multiple pages. Once a flow is created, or after a change, the user *deploys* the flow, both saving it to the server and (re)starting its execution on the Node-RED server.

Nodes in a flow inherit from the `Node` base class. A Node is also subclass of an `EventEmitter` in the Node.js event API that implements the observer design pattern to maintain subscriber lists defined by *wires*, and emits events to downstream nodes. On instantiation, input nodes may subscribe to external services, begin listening for data on a port, or begin processing HTTP requests. Once data is processed by a node, either from an external service, or received from an upstream node via its "input" handler, the node calls the base class Node `send()` method with a JavaScript object. The send method delegates to the `EventEmitter.emit()` method to send named events to downstream Node instances which process data and either generate additional events, or communicate with outside services or the OS.

Node-RED is supported by IBM and a large community of users that contribute new nodes and flows[1]. New nodes can be implemented in JavaScript and added to the system by adding an HTML file to implement the UI in the browser, and a JavaScript file for data processing or integration on the server. Text representations of flows can be imported and exported between instances. When a node used in one instance of Node-RED is not available on another, a placeholder node is shown in the UI to indicate the user must install a missing node implementation before the flow can be deployed.

Node-RED's flows are similar to the Processor's Pipes in that they are expressed using JSON as shown in Figure 6. One difference is that "wires" are not separate objects, but are arrays associated with each node connecting it to a downstream node. Configuration information that may be shared between nodes such as the twitter user name or MQTT topic is held in a *configuration node* without wires. Unlike the Processor, flows on different tabs or pages are not separated in Node-RED, there is only one flow for the entire system.

2.3 Analysis

Both systems use a drag and drop visual editor to generate data flows using JSON. The generated flows (pipes) consist of nodes (modules) connected via wires. On execution, modules are instantiated in memory and execute code as they receive data.

The Processor is targeted for server deployments and supports

```
[
  {
    "id": "81bbf709.7e4408",
    "type": "mqtt-broker",
    "broker": "public",
    "port": "1883",
    "clientid": ""
  },{
    "id": "23df8a24.dc2076",
    "type": "twitter in",
    "twitter": "",
    "tags": "",
    "user": "false",
    "name": "myeverydayearth",
    "topic": "tweets",
    "x": 185.09091186523,
    "y": 75.090911865234,
    "z": "db25cd7.f24da3",
    "wires": [["7b3fdb79.84c024"]]
  },{
    "id": "85ac23b0.7a53e",
    "type": "mqtt out",
    "name": "Output Sensor",
    "topic": "output-sensor",
    "broker": "81bbf709.7e4408",
    "x": 467.09091186523,
    "y": 212.09091186523,
    "z": "db25cd7.f24da3",
    "wires": []
  },{
    "id": "7b3fdb79.84c024",
    "type": "function",
    "name": "Example function",
    "deviceId": "server",
    "func": "\nreturn msg;",
    "outputs": 1,
    "x": 318.09091186523,
    "y": 142.09091186523,
    "z": "db25cd7.f24da3",
    "wires": [["85ac23b0.7a53e"]]
  }
]
```

Figure 6. Node-RED Example Flow

[1] http://flows.nodered.org/ Node Red Flow Library. Accessed 15-Aug-2014.

multiple users. As such, logged in users can start and stop, create, update and delete individual data flow programs independently of other pipes and users on the same platform. Node-RED hosts a single set of connected nodes, and can only manage one flow on the system. Unlike Node-RED, the Processor never access local sensors (e.g. local GPIO pins, or USB ports), or OS services directly since the execution engine is server-based. Node-RED can be deployed on a smart device; the lightweight nature of the node.js and the simplicity of the Node-RED execution engine allows Node-RED flows to execute with good performance on devices such as the Raspberry Pi. The different systems come bundled with different modules/nodes; Node-RED has a large set of nodes and flows contributed by the open source community. The scripting nodes on the Processor are written in Python, whereas Node-RED supports JavaScript.

Unlike Node-RED's data flow model, the Processor pipe data flow model has a separate section for wires allowing programs to iterate through nodes and wires separately, while wires in the Node-RED model are associated directly with nodes. Processor modules can have multiple inputs and outputs, while Node-RED

```
{
  "name": "Twitter to Sensor",
  "modules": [{
      "id": 2,
      "name": "Twitter Feed",
      "type": "twitter-in",
      "x": 110.99999809265,
      "y": 121.18181610107,
      "config": {
        "feed": "myeverydayearth"
      }
    },{
      "id": 1,
      "name": "Sensor Output",
      "type": "sensor-output",
      "x": 372.99999809265,
      "y": 269.36362457275,
      "config": {
        "sensor": "mike.output"
      }
    },{
      "id": 3,
      "name": "Python Script",
      "type": "py-script",
      "x": 211.99999809265,
      "y": 213.90906524658,
      "config": {
        "script":"input = output"
        }
    }],"wires":[
    {
      "source": {
        "module": 2,
        "terminal": 1
      },"target": {
        "module": 3,
        "terminal": 0
      }
    },{
      "source": {
        "module": 3,
        "terminal": 1
      },"target": {
        "module": 1,
        "terminal": 0
      }
    }
  ]
}
```

Figure 5. WoTKit Processor Example Pipe

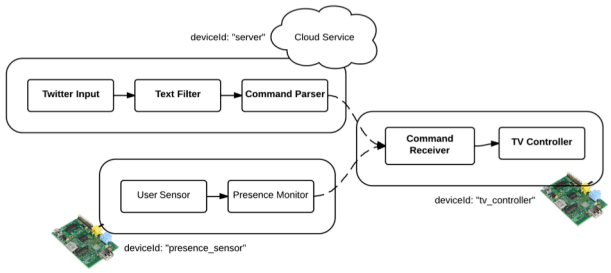

Figure 7. In a distributed flow, nodes may be hosted on servers and devices; remote wires connecting devices to servers, indicated by the dashed connections.

can only support single input nodes. All pipes are associated with users on the Processor, and have an 'owner'. Each user manages their pipes on the shared service independently. In Node-RED, a single global flow is managed and deployed by a single user.

3. TOWARD DISTRIBUTED DATA FLOW

Because operations in a data flow can run as soon as data is available, and share no state with each other, operations are inherently parallel and can run independently [2]. Like functional programs [6], there are no 'side effects' in data flow programs; computation can be moved between processors and devices so that parts of a data flow run in parallel on different devices. This approach has been used in other contexts to provide scalable real time data processing for example [17]. This ability to split and distribute a data flow allows some operations to execute in the cloud while others can execute an edge device depending on user preferences, the cost and speed of communications, and the performance of the run time host.

In this section we propose a cloud-based platform called 'WoT Flow' that takes advantage of this property of data flow programs and the open source Node-RED system to provide an execution engine suitable for both multi-user cloud environments and individual devices. Developers both create flows and identify the devices participating in these flows. On deployment and execution of a flow, a portion of the overall flow, a *subflow*, will be executed on the cloud service, while other subflows will be distributed and executed on devices. On deployment of the flow, devices will be notified, and will download the segments of the flow they are responsible for executing. Both devices and the server will begin execution of the flow. Where necessary, devices will send data to the server or to other devices, leveraging network connectivity and appropriate middleware for data transfer.

3.1 Data Flow Extensions

To support distributed flows, the data flow program model used by Node-RED or the Processor needs to be extended in a number of ways. First, flows should have an owner. This allows the system to associate access and control of a flow to a specific user and their devices, a requirement for any multi-user system.

Second, it must be possible to mark nodes as 'device' nodes, 'server' nodes, or 'mobile' nodes. Device nodes are those that rely on local device connectivity, or the specific capabilities of a device. This may include, for example, direct access to the local file system, connected sensors on sensor pins, or connectivity to a USB port on the device. A server node may require support for a

specific programming language on the platform, the processing power or data storage of a cloud-hosted server, or server connectivity outside of firewalls, etc. Nodes with unspecified device ids can be considered mobile nodes and may be hosted on either a device or a server, depending on user preferences, or a heuristic that determines the best placement of a node.

Similarly, we must extend a simple flow model to include different types of arcs or wires. In a distributed flow, the wires between nodes are not all local connections in the same execution engine, but may involve the transfer of data between servers and devices over a local or wide area network. "Local" wires are hosted on the same execution engine, while "remote" wires will require a network connection. An example distributed flow is shown in Figure 7 where a subflow related to Twitter input and message parsing are hosted on a server like the WoTKit Processor, while a presence sensor and TV control actuator sub flows are hosted on the "presence_sensor" and "tv_actuator" devices.

3.2 Work to Date

The WoT Flow system is at the early stages of development. We are currently finalizing the distributed flow model by modifying Node-RED to support distributed flows.

To date we have modified certain nodes in Node-RED by adding a *device id* property. This property specifies the device where the node should execute. It may represent a server-hosted instance of Node-RED (e.g. "server"), a sensor (e.g. "presence_sensor") or other thing. When a flow is executed, an added node analysis phase looks for wires in the flow between collaborating devices. Nodes in a flow connected on different devices are replaced by a hidden "wire in" or a "wire out" node for connectivity between distributed hosts of the flow. Unlike local Node-RED wires, "wire" nodes connect Node-RED instances on different devices using a shared MQTT message broker. Inner nodes of a subflow that are not hosted on the current device are deleted from the flow locally, since they will be executed on another device, and have no connectivity to the local device.

Other devices participating in the flow install the flow using the built in Node-RED import mechanisms. When they find a node that is not on their device, they also subscribe or send to a corresponding MQTT topic using their "wire" nodes.

4. RELATED WORK

Several web-based platforms provide support for connecting hardware and services for real time interaction. Some target IoT applications specifically, while others are more general purpose. In this section, we provide an overview of some representative systems. We categorize these systems as *Thing Connectors*, *Condition Action Systems*, and *Data Flow Systems*.

Thing Connectors connect Internet-connected things to each other. SpaceBrew [1] provides a way for devices to publish and/or subscribe to data (strings, boolean values or numbers). Using the SpaceBrew web interface, users configure the connections between publishers on the left and subscribers on the right to create interactive applications. By connecting devices such as Arduinos, home automation equipment and displays to SpaceBrew, it is possible to create useful interactive space applications by connecting the output of one thing or service to another, leaving data processing, sensing and actuation to the connected things themselves.

Paraimpu [13] is another system for creating connections between things. In the workspace, *sensors*, which are sources of data, are connected to *actuators* to receive data. These connections can perform filtering or mapping operations using JavaScript expressions. A connection between a wind speed sensor and a twitter message actuator can be configured to send a Twitter message that expresses the wind speed in English by mapping values between 0 and 50km/h to "low wind speed" and values over 50 to "high wind speed". Paraimpu supports sharing things with others to make it easy to add new things to your workspace.

Space Brew and Paraimpu focus on providing an easy way to connect things to each other, making it possible to easily reroute data from one thing to another, and leverage the integration work done by others necessary to use things with the platform.

Condition Action Systems present a web interface for creating scripts that trigger an action in a target web service when a condition is met on another service. If This Then That (IFTTT) [8] is a multi-user internet service for building interactive 'recipes' made up of an event condition called a 'trigger' and an 'action'. The triggers come from channels that you connect to the system corresponding to popular web applications and services such as social networks, document storage, email, and web-based IoT platforms. While not targeted specifically at IoT systems, several channels are connection to Internet connected devices such as light switches, and thermostats, fitness tracker, and others. Configuration of these recipes is done using a simple user interface, where the specific triggers and actions depend on the channels used. Like IFTTT, Zapier [3] also connects online web services or applications, using a 'trigger' and 'action' metaphor.

Condition Action systems focus on providing connectivity to existing cloud services 'out of the box', and remove the need for any coding, targeting non-technical users that want to perform integration tasks.

Data Flow Systems provide a data flow-programming paradigm for IoT applications. The WoTKit Processor and Node-RED tool fall into this category. Huginn [7] is another data flow system that allows developers to create 'agents' that can be connected together by users of the system, creating a flow of events between agents. Users configure contributed agents with the agent editor, adding other agents as event sources. Users can then view the overall event flow diagram visually. Once the agents are configured and connected, the system moves events through the system taking action on your behalf.

Yahoo Pipes [14] is an online service to mashup data from the web. Users create pipes visually in the browser by dragging pre-configured modules such as RSS feeds onto a workspace and wiring them together. Once you've created a pipe, you can request the output of a pipe like any other web based feed. While Pipes does not process data in real time, the output from Pipes can be requested periodically to get regular updates when data changes on the web. Yahoo Pipes provides testing and debug facilities to monitor and tune the execution of pipes.

One of the most well known tools for developing hardware-centric applications is LabVIEW[2] [5]. Using LabVIEW, users create data flow programs called "virtual instruments" (VIs) visually by connecting objects supplied by the platform in block diagrams. Objects correspond to input and output devices, functionality and user interface components. Using LabVIEW, developers can create user interfaces for control of equipment, process and view data from measurement equipment and open

[2] http://www.ni.com/newsletter/51141/en/, What is LabVIEW: 2013. Accessed: 2014-08-12.

source hardware such as Arduino. While LabVIEW does not support creating block diagrams in a browser, VIs can interact with services on the web, and can themselves be deployed as web services.

The COMPOSE project [12] aims to provide an open Marketplace for IoT applications. A key component of the system called the *data plane* and servIoTicy [16] is a cloud hosted system that includes features for data storage and processing of connected things. Service Objects (SO's) in the COMPOSE platform can include data processing logic that provides the capability to transform sensor updates and generate new data for subscribers in a fashion similar to Node-RED.

Systems in this category leverage a data flow-programming model for not only the reuse of integration endpoints, but also the code necessary to process data as it flows from source things and services to sinks. In a data flow, there are typically more than two modules, and one connection or route between them. In most of these systems, it is possible for users to introduce new modules extending the capabilities in the platform itself rather than relying on the connectors or channels provided by system integrators. While Huggin and Node-RED provide data flow programming capabilities for a single device/user, the Processor and Yahoo Pipes manage data flow programs for many users, and are designed for deployment in the cloud.

4.1 Summary

An interactive IoT system must, at a minimum, provide an easy way to 'connect' things. It is important to provide a set of common integration points such as sensor platforms, email, HTTP, MQTT connectivity out of the box. All of the systems aim to make it easier for developers to create interactive applications, minimizing the programming effort involved. By providing reusable integration and processing modules, they allow users of their systems to leverage the work done by others.

While SpaceBrew, Huggin and Node-RED provide the ability to create applications, only one set of routes, flows or agents is hosted on a deployment at a time, assuming one administrative user (or set of users) is in control of the entire instance. Cloud services like Paraimpu, the WoTKit Processor, Yahoo Pipes, and IFTTT highlight the need to support multiple users managing and executing their applications independently and without access to or interfering with other user's applications. LabVIEW, Node-RED, and the Processor allow end users to extend the basic building blocks of the platform, adding new primitives for reuse.

With the flexibility VDFPLs offer, programs can become complex; these systems require some level of debugging and monitoring capabilities. None of the systems described provide an easy way to take advantage of computing resources of both cloud based services and smart devices as we have proposed.

5. OPEN QUESTIONS

Our early experience has already provided some insight on how to best support distributed data flows for the IoT. However, even at this early stage our work has raised several questions:

Do visual data flow programming languages find the right balance between ease of use and flexibility for creating real time IoT applications? Are there ways of extending *Thing Connectors* or *Condition Action Systems* for greater flexibility and end user programmability?

Is the distributed data low model described in section 3 complete? What other attributes for flows, nodes and wires are required? How should we extend the visual programming language to indicate different devices in a distributed flow?

In a distributed system, connections between devices may fail or errors may occur on remote devices. How can we support distributed monitoring, debugging and error handling of distributed flows?

Can web protocols and the web itself be used as the connector and support infrastructure for distributed data flow, i.e. can the WoT support data flow architectures directly?

Related to security, how do we ensure devices don't accidentally download malicious code? Do we need to digitally sign subflows downloaded from a coordinator?

What are the appropriate heuristics for deciding node placement automatically, where a node may be hosted either in the cloud service or on the device? How do we indicate the cost and performance of executing nodes, or network connectivity?

6. CONCLUSIONS

In this paper we presented the detailed design of two existing data flow platforms to inform our work toward a distributed data flow platform. We then described our early work toward modifying Node-RED to support distributed flows by specifying the device for node execution. This work has raised questions for us around the programming model, error handling, security and heuristics for distribution. In future work, using appropriate optimization heuristics, we hope to support the automatic partitioning and distribution of data flows based on based on profiling execution and participating resource capabilities and constraints imposed by the end user around cost, performance and security.

7. ACKNOWLEDGMENTS

Our thanks to IBM and the authors for contributing Node-RED to the open source community. This work was funded in part by NSERC.

8. REFERENCES

[1] About Spacebrew: *http://docs.spacebrew.cc/about/*. Accessed: 2013-01-30.

[2] Ackerman, W.B. 1982. Data Flow Languages. *Computer*. 15, 2 (Feb. 1982), 15–25.

[3] Automate the Web - Zapier: *https://zapier.com/*. Accessed: 2014-08-12.

[4] Baroth, E. and Hartsough, C. 1995. Visual Object Oriented Programming: Concepts and Environments. *Visual Programming in the Real World*. Manning Publications Co. 21–42.

[5] Blackstock, M. and Lea, R. 2012. IoT mashups with the WoTKit. *Internet of Things (IOT), 2012* (Wuxi, China, Oct. 2012), 159–166.

[6] Hughes, J. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (Apr. 1989), 98–107.

[7] Huginn. Your agents are standing by.: *https://github.com/cantino/huginn*.

[8] IFTTT / Put the internet to work for you.: *https://ifttt.com/*. Accessed: 2013-01-29.

[9] Johnston, W.M. et al. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (Mar. 2004), 1–34.

[10] MAX is a visual programming language for media: *http://cycling74.com/products/max/*. Accessed: 2014-08-12.

[11] Node-RED: *http://nodered.org/*. Accessed: 2014-08-12.

[12] Pérez, J.L. et al. 2014. The COMPOSE API for the Internet of Things. *Proceedings of the Companion Publication of WWW 2014* (Republic and Canton of Geneva, Switzerland, 2014), 971–976.
[13] Pintus, A. et al. 2011. The anatomy of a large scale social web for internet enabled objects. *Web of Things Workshop (WoT 2011)* (San Francisco, CA, USA, 2011), 6:1–6:6.
[14] Pipes: Rewire the Web: *http://pipes.yahoo.com/pipes/*. Accessed: 2013-01-30.
[15] ROBOLAB for Lego Mindstorms: *http://www.legoengineering.com/program/robolab/*. Accessed: 2014-08-12.
[16] servIoCity: *http://www.serviotic y.com/*. Accessed: 2014-08-12.
[17] Storm: *http://storm.incubator.apache.org/*. Accessed: 2014-08-12.
[18] Blackstock, M. and Lea, R. 2014. IoT interoperability, a Hub based approach. *Internet of Things (IOT), 2014 4th International Conference on* the (Boston, USA, Oct. 2014).