DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks

Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, Mani Srivastava
University of California, Los Angeles
Los Angeles, California
{jnoor,tsenghy,garcialuis,mbs}@ucla.edu

ABSTRACT

Programming distributed applications in the IoT-edge environment is a cumbersome challenge. Developers are expected to seamlessly handle issues in dynamic reconfiguration, routing, state management, fault tolerance, and heterogeneous device capabilities. We introduce DDFlow, a macroprogramming abstraction and accompanying runtime that provides an efficient means to program high-quality distributed applications that span a diverse and dynamic IoT network. We describe the programming model and primitives used to isolate application semantics from arbitrary deployment environments. Using DDFlow leads to portable, visualizable, and intuitive applications. The accompanying system runtime enables dynamic scaling and adaptation, leading to improved end-to-end latency while preserving application behavior despite device failures.

CCS CONCEPTS

• **Information systems → Spatial-temporal systems**; • **Computer systems organization → Sensors and actuators**; **Dependable and fault-tolerant systems and networks**; • **Software and its engineering → Software development methods**;

KEYWORDS

Macroprogramming, Declarative programming, Distributed systems, IoT networks, Visualized programming, Adaptation, Dynamic reconfiguration, Fault tolerance

ACM Reference Format:
Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, Mani Srivastava. 2019. DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks. In *International Conference on Internet-of-Things Design and Implementation (IoTDI '19), April 15–18, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3302505.3310079

1 INTRODUCTION

The booming expansion of the Internet-of-Things has led to a world where advanced sensors, actuators, and specialized hardware can interact with the environment in unprecedented ways, enabling emerging applications ranging from agriculture to city planning and military surveillance [1, 19, 24]. A challenge arises in how to effectively specify and manage coordinated activity across a wide variety of heterogeneous hardware. Enterprise frameworks, such as Samsung SmartThings [25] and Apple HomeKit [10], offer simplistic frameworks that are hardware dependent. For complex solutions derived from open-source tooling, developers are expected to build their own distributed systems. This often leads to a lack of portability, programmability, and efficient system management.

In recent years, new capabilities have further exacerbated the problem of system management in the IoT-space. First, new enterprise and research devices have created even more heterogeneity and hardware isolation [6, 11]. Second, new accelerators and custom hardware for applications such as machine learning create a broad variation in device efficiency [5, 17]. Finally, new IoT devices introduce a more dynamic range of actuation, including actions such as opening/closing valves in industrial systems, camera PTZ, 2D movement of ground robots, and 3D movement of drones [7]. Capturing these new capabilities have made the challenge of application development and management in the IoT world a burdensome task.

This paper introduces DDFlow, a macroprogramming abstraction and accompanying runtime that provides an efficient means to program high-quality distributed applications that span a diverse and dynamic IoT network. Application specification is accomplished through a declarative user interface implemented as an extension of the Node-RED IoT system [3]. Developers visually specify what the application should accomplish without explicit regards to the how. This allows for effective visualization, programmability and reusability.

The accompanying DDFlow system runtime dynamically deploys applications to the available network. The distributed coordinator maintains the current state of the network and intelligently maps services to available devices while minimizing end-to-end latency. In the case of significant network changes, or device failure, DDFlow will reconfigure to preserve application semantics by re-mapping the computation onto the network and/or switching to alternative networking protocols.

Contributions. Our contributions are summarized as follows:

(1) DDFlow: a visual and declarative programming abstraction for heterogeneous IoT networks.
(2) A system runtime that supports DDFlow programs by dynamically scaling and adapting application deployments.
(3) A preliminary evaluation of the system in adapting to node failures and an unstable network.

2 BACKGROUND & RELATED WORK

Macroprogramming refers to a field of research that aims to provide centralized specification for applications spanning distributed

IoTDI '19, April 15–18, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6283-2/19/04…$15.00
https://doi.org/10.1145/3302505.3310079

sensor networks. The goal is to enable complex coordinated activity without forcing developers to configure individual devices or account for a particular network. The typical approach is to define a language (often with an accompanying runtime framework) that enables efficient description and execution of global application behavior. We briefly touch upon a few notable systems.

DFuse [13] is a macroprogramming framework focused on aggregating sensor data into higher level evaluations. Applications are defined as dataflow graphs, and the runtime determines where to place data fusion points in a heterogeneous network. While effectively capturing data fusion applications, DFuse lacks support for external actuation, mobility, and scalability; tasks such as sensing should often be subdivided across a set of available devices.

Kairos [8] is a programming language to define global behavior of distributed computation by expressing pairwise interactions between neighboring nodes. This places a constraint on application portability and expressivity, and can potentially lead to unexpected behavior in the mobile IoT environment.

Regiment [18] offers a functional macroprogramming language that groups data streams into regions based on spatial locality, allowing an expressive language that captures many sense-compute applications in sensor networks. Due to the high-level of the language abstraction, it is generally ineffective at expressing heterogeneous network capabilities. With no language support for actuation, it falls short in control applications.

Mobile Fog [9] presents a programming model for IoT applications spanning the fog network hierarchy. Applications are constructed using event-driven message-passing callbacks. With support for dynamic scaling, complex event processing, and distributed key-value storage [15], Mobile Fog offers a flexible programming interface that enables centralized specification. However, for complex applications spanning heterogeneous edge devices, grouping all devices' application logic into a unified callback function can become unwieldy and difficult to understand.

The most directly analogous to DDFlow is the *D-NR* system described in [4]. Their work provided an initial implementation of a distributed extension to Node-RED. Developers define a master flow composed of sub-flows which deploy to different devices in the network. Heterogeneity is accomplished by explicitly categorizing devices into *edge*, *IO*, and *compute*, with inter-device communication via MQTT brokers. Ultimately, D-NR provides an interesting prototype for visual programming. However, with a lack of declarative specification, fault tolerance, and dynamic adaptation, it falls short in delivering a robust system framework for IoT applications extending outside a controlled home environment.

3 DDFLOW ABSTRACTION

Our goal is to provide a flexible programming framework at the appropriate level of abstraction in order to formulate complex applications without forcing a developer to concern themselves with low-level network, hardware, and coordination details. The challenge lies in how to efficiently and succinctly express application semantics in a manner that allows for dynamic deployment across diverse environments. We move away from binding a computation to a device, or defining an explicit scale to a computation. Instead, we focus on expressing collective behavior by relying on a high-level declarative interface for application formulation and specification, thereby allowing an underlying runtime to dynamically scale and map to available resources.

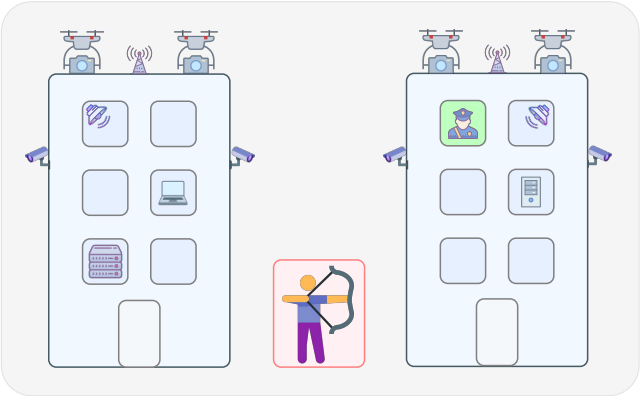

Figure 1: DDFlow motivating application. Cameras identify an active school shooter, and drones are deployed to follow the shooter and provide a live video feed.

3.1 Motivating Application

In order to ground the discussion, we will focus on an example that reflects a latency-constrained application on an edge network. The motivating scenario is as follows: a school shooter is actively roaming a college campus. With a sensor network that spans campus *cameras*, *drones*, a *speaker* and cloudlet *servers*, the objective is to achieve the following:

(1) Recruit cameras in the region of interest to identify a target or object-of-interest.
(2) Classify captured image frames to detect the target.
(3) Upon detection, a speaker at a command center plays a sound to notify those nearby that the target has been identified.
(4) Available drones pursue the target and stream a live feed.

This application, illustrated in Figure 1, showcases many pain points involved in creating applications for heterogeneous IoT networks. First, different devices have vastly different mobility and computational abilities. Taking advantage of this heterogeneity is a non-trivial task. Second, coordinating the efforts of these devices in an ad-hoc network presents optimization challenges. Finally, reconfiguring the application to a new campus or environment requires significant adaptation, placing a substantial burden on a developer.

To ease this burden, DDFlow takes a declarative approach. Instead of specifying low-level implementation details, a developer states the high-level objectives, effectively translating an application *description* into an application *specification*. The system runtime translates this declarative specification into an explicit computation graph that dynamically deploys across devices in the network.

3.2 Declarative Dataflow

The motivating application's dataflow graph, depicted in Figure 2, represents a full DDFlow specification by containing the information necessary to describe the application. It begins by generating image frames in the area of interest. These frames pass through an

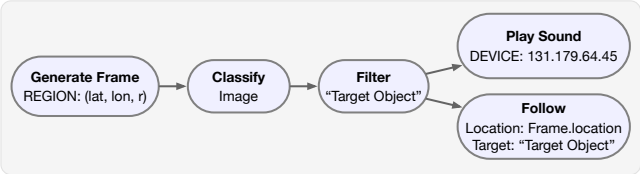

Figure 2: Motivating application in DDFlow: camera frames in a region of interest are classified in search of a target object, upon which a speaker plays an alert and drones are deployed to follow the target.

image classifier, which is filtered to focus solely on identification of the target. Once identified, a speaker is notified to play a sound, and drones are deployed to the last observed location of the target, initiating a follow sequence.

Dataflow is a natural choice for both visualizing and describing applications spanning IoT networks. It is a programming model that has been previously validated by the community (e.g., [13]), and is a logical choice for an event-driven abstraction that begins with sensing and ends with actuation, capturing the perception ⇒ cognition ⇒ actuation paradigm that is pervasive across IoT applications [16]. Applications developed in this manner are highly visual and intuitively understood, providing an interface that allows developers, project managers, and future engineers to fully grasp an application at a glance. Furthermore, and most importantly, dataflow enables a decoupling of application specification from the deployment network, enabling both portability between networks, and an opportunity for a system runtime to provide optimization. To this end, there are a set of abstraction primitives provided by DDFlow extending dataflow to support a declarative application programming interface.

3.3 Model Primitives

While dataflow allow for the decoupling of application semantics from the deployment environment, it lacks the sufficient descriptors necessary to enable effective runtime scaling. In deploying an application to diverse environments with varying regions, devices, and capabilities, the precise scale of an application is often not known a priori. DDFlow aims to allow an application to be implicitly and dynamically scaled depending on the resources available at runtime and the constraints of an application. Existing work generally lacks this notion; not only should multiple tasks be assignable to the same device, but an individual task may be collectively accomplished by replicating across a dynamic set of devices. DDFlow achieves runtime scaling via the Node and Wire fundamental primitives.

3.3.1 Node. DDFlow applications are defined as a sequence of actions, or Nodes in a dataflow graph. A Node is a computational abstraction representing a stateful function that maps inputs to outputs, either of which are optional. Each Node corresponds to at least one instantiation of a task that must be deployed onto a device in the network (e.g., generating camera frames, classifying images, playing a sound). Inputs and outputs are key-value dictionaries (i.e., JSON messages) that contain application data as well as metadata including timestamp and sender.

Nodes are constrained via a set of parameters relevant to a particular task (e.g., Filter contains a key-value to filter incoming messages). Due to the spatiotemporal nature of IoT applications, two fundamental parameters underlying all Nodes are *Region* and *Device*, optional parameters restricting the deployment of a task to a particular spatial region or set of devices. In Figure 2, the Node generating camera frames is associated with a circular region (lat, lon, r). Only devices capable of generating camera frames within the specified region of interest are potential candidates during runtime scaling. Regions can be described as a bounding box, with other structured location information, or as a list. Dynamic and moving regions can be specified via the *input* keyword, which monitors a given Node's inputs for a *region* value to update the existing deployment region. The *Device* parameter supercedes the *Region* parameter and allows for precise Node placement.

To deploy the motivating application in Figure 2 to a new environment, a developer needs only to change the *Region* parameter for generating camera image frames and the *Device* parameter for playing a sound. During deployment, the system runtime will dynamically scale the application to the available devices in the regions of interest that are capable of accomplishing the specified Node tasks.

3.3.2 Wire. In defining a sequence of actions, Nodes are connected via Wires, representing a connection in a dataflow graph. Each Wire carries a key-value dictionary from the output of one Node to the input of the downstream Node. Due to the elastic runtime scaling of Nodes, Wire definitions follow one of three forms: *Stream* (one-to-one), *Broadcast* (one-to-many), and *Unite* (many-to-one).

In the motivating application, all devices that are performing the Follow task should receive updates to target location regardless of which camera generated the detected frame. As such, the connection between Filter and Follow is a *Broadcast* Wire. On the other hand, only one device is responsible for playing a sound upon target identification; as such the connection from Filter to Play Sound is a *Unite* Wire. Finally, in order to take advantage of motion detection capabilities, each camera generates a stream of frames to an independent Classify instance; as such, Generate Frame and Classify are connected via a *Stream* Wire.

3.4 Extending DDFlow Capabilities

Given a suite of Nodes, developers can express a wide variety of IoT applications. However, certain applications will require custom logic and capabilities not already defined within DDFlow.

The DDFlow UDF Node allows developers to encode custom logic within their programs. Developers define a simple callback function to be executed upon receiving a message, additionally allowing for basic state to be preserved across callback executions.

For more complex applications, Nodes may be required that are not yet defined within the DDFlow framework. Any developer can enhance the suite of available Nodes by defining new, custom Nodes. To support this, the DDFlow system runtime provides a means for a developer to define a class following the DDFlow interface, insert the class into the system, and thereby make new Nodes available for writing applications. Heterogeneous hardware can have differing underlying class implementations while exposing

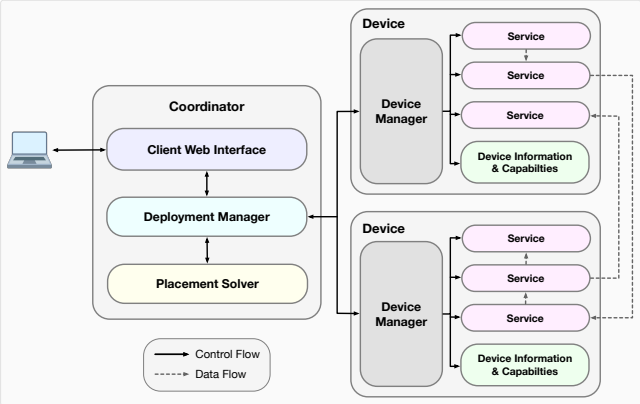

Figure 3: DDFlow architecture. The Device Manager provides intra-device coordination. The Coordinator orchestrates applications and provides inter-device coordination.

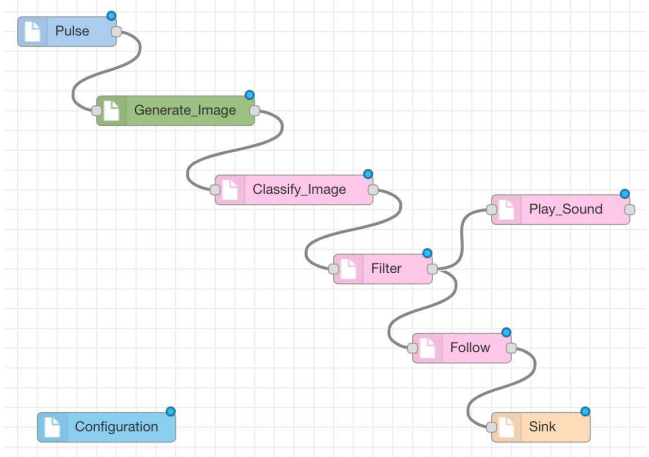

Figure 4: Screenshot of the DDFlow user interface presenting the motivating application described in Section 3.1. The *Pulse* Node triggers a periodic message to initiate sensing. The *Sink* Node forwards *Follow* outputs to the client.

the same interface, thus allowing DDFlow to seamlessly integrate different devices with the same high-level semantic capabilities.

4 SYSTEM RUNTIME

To support the DDFlow abstraction, we have developed a system runtime based on Node-RED [3], an open source framework that provides a graphical user interface for programming IoT devices. While Node-RED provides a single-device programming environment, we have built a distributed system atop the basic graphical interface, provided support for declarative DDFlow applications, and created system tooling for dynamic deployment and reconfiguration of applications spanning heterogeneous IoT networks.

The DDFlow system follows a service-oriented architecture, a proven architecture for dataflow and IoT systems [12, 20], depicted in Figure 3. Each device, whether a powerful cloud server or a lightweight edge device, offers a distinct set of *Services*. Services represent a particular implementation for a Node in the DDFlow abstraction (e.g., image classification, playing a sound). In a heterogeneous environment, different physical devices may have different underlying implementations, but offer the same *Service* to DDFlow via the same high-level interface.

Intra-device coordination is accomplished through a *Device Manager*, a lightweight web server that runs on every device in the network. It is responsible for activating/deactivating service instances and exposing device details (e.g., available services, current state). Resource constrained devices that cannot deploy a *Device Manager* expose their capabilities through a proxy device.

Inter-device coordination is accomplished through a *Coordinator*, a web server that accepts and manages DDFlow applications as they are issued onto the available network. Figure 4 is a screenshot of the web interface presenting the DDFlow specification of the motivating application. The *Coordinator* is composed of three main components: an end-user facing web interface that accepts DDFlow applications, a deployment manager that interfaces with the *Device Managers* running on each device, and a placement solver that maps an application task graph to available devices. The *Coordinator* monitors deployed applications to detect significant network changes, such as a disconnected or failing node, and adjusts a deployment mapping as needed. For resilience and fault tolerance, the *Coordinator* can be replicated onto many devices; placement of the *Coordinator* is recommended on devices with high availability.

4.1 Dynamic Deployment

When an application is deployed, the *Coordinator* contacts all *Device Managers* to obtain updated state information and decide which devices to map an application task graph. In doing so, it may map many services to the same device, such as a powerful server with accelerators, and it may map the same service to many devices, such as a fleet of drones or sensors performing a group task.

Each *Device Manager* provides the *Coordinator* with information including location, utilization, estimated service and network latencies, and devices within wireless range. From this information, the *Coordinator* constructs a network topology graph and a task graph. The topology is modeled with wired devices connected to a backbone network and wireless devices connected to other devices within range. The task graph is generated from the DDFlow application graph by scaling Nodes based on the region, availability, and capability constraints.

Given a network topology, task graph, and device capabilities, the *Coordinator* formulates computation mapping as a linear programming problem with the objective to minimize the longest path's end-to-end latency in the task graph. While admittedly a simplified metric, latency serves as baseline by which a system can begin to compare relative network speeds and model network characteristics. The solver will find the best solution to the objective function given the following constraints: (1) Neighbors in the task graph must also be accessible from each other in the network graph. (2) Devices must possess the necessary Node implementations for all assigned tasks. (3) Devices must have available the necessary resources required to execute the assigned task. The *Coordinator* solves this linear programming problem and issues task requests

to all the relevant *Device Managers*. This placement algorithm, described in detail in [26], is only one such algorithm for assigning tasks to devices. It is trivial to swap for another placement solver.

4.2 Dynamic Adaptation

To enable dynamic adaptation and recovery, the *Coordinator* probes devices in the network at an application-defined periodicity to monitor for environmental changes (e.g., disconnected node, overloaded device). Upon detection of a significant deviation of system characteristics (i.e., compute or network latency), the *Coordinator* computes a new placement mapping of an application. This new mapping is evaluated with respect to the objective function (e.g., end-to-end latency). In the case of projected improvement greater than a threshold, a remapping is triggered. Any failed or disconnected device will additionally trigger a remapping.

The *Library* system service provides a key-value data storage API for services to preserve local state. Thus, when the *Coordinator* issues the pertinent task activation/deactivation requests, all task-relevant key-values are forwarded from the terminating service to the device launching the new task instance.

Communication adapts at a finer granularity. The *Router* system service hides network-specific details by providing transparent messaging to devices. For a given device-service destination, a forwarding table identifies the optimal next-hop, either pushing the packet via TCP/IP infrastructure mode or via peer-to-peer Wi-Fi ad hoc mode. Techniques such as [27] are in consideration for further abstracting network communication.

5 EVALUATION

The key enabling feature for real-time adaptation is an application specification that does not rely on a particular network instantiation. By providing a declarative macroprogramming abstraction, DDFlow recovers from failures gracefully by transferring failing computation or networking to alternative devices capable of similar semantic functionality. Systems such as [4] rely on a static computation assignment and standard TCP over IP for networking. This leads to a lack of portability and resiliency. In mission critical applications, such as those providing aid to first responders and law enforcement, failure is crippling. These high stress environments require an underlying runtime that does not rely on a human-in-the-loop for recovery.

To showcase the benefits of a declarative programming interface with an adaptive runtime, we developed a simulation testbed. Devices are inserted into the Airsim [22] environment simulator and connected via the Mininet [2] network emulator. The main system components simulated in this evaluation, to represent a simplified version of the motivating application, are the following:

(1) A camera used to identify the target
(2) Three servers that are capable of providing image classification: a server with a GPU, a server without a GPU, and a camera-local accelerator
(3) A client device with accompanying speaker used to alert upon target identification
(4) A drone that follows the target after identification
(5) Three wireless access points that provide the drone with communication to the backbone network.

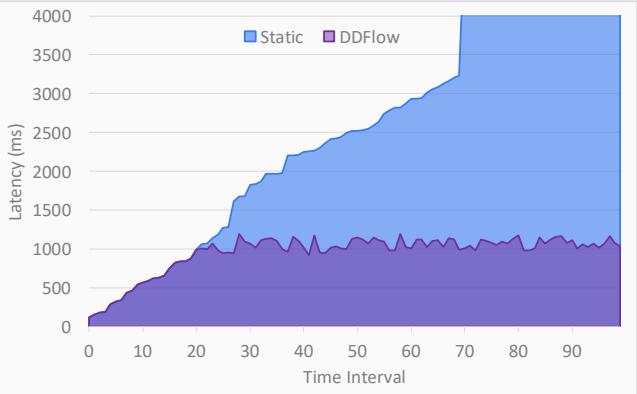

Figure 5: Adaptation during device overload. DDFlow will switch to an alternate device that is capable of providing the same service to minimize impact on end-to-end latency.

Single-hop wired network links are modeled with a 2ms link latency, to account for processing, queuing, transmission, and propagation delays [14]. Due to the inherent variability, wireless links are modeled as varying from 30-50ms [23].

Three devices are capable of providing classification. The first is a server using an NVIDIA Titan X GPU and the YOLOv3 model (~20ms per frame) [21]. The second is a server relying on its Intel Xeon CPU E5-2620 v3 @ 2.4GHz with the YOLOv3-tiny model (~880ms per frame). Finally, a Google Vision Kit camera comes equipped with an Intel Movidius VPU [6]. It contains support for constrained TensorFlow Lite models, with its default image classifier requiring ~3.2s per frame.

In the following scenarios, either device or network degradation causes a static deployment to slow and ultimately fail, whereas DDFlow is able to recover from failures and preserve application semantics.

5.1 Device Overload and Failure

The first application scenario is the following: the camera is streaming image frames to an available classifier. Upon successful target classification, the speaker is notified to play a sound.

A static deployment streams frames to the fastest classifier, the GPU server. If that server becomes overloaded, performance degrades. DDFlow is able to switch to another available device to minimize impact on end-to-end latency and preserve application semantics. This is shown in Figure 5.

As the GPU server becomes overloaded, both static mapping and DDFlow see end-to-end latency increase. After a certain threshold it becomes advantageous to switch to the CPU server, and as such DDFlow is able to maintain minimal impact to end-to-end latency. Eventually, the GPU server fails, crashing the statically deployed application, but the DDFlow application continues.

5.2 Access Point Failure

The second application scenario illustrates adaptation during network over-utilization and access point failure. The objective is to stream live video from the drone to the client. As the drone moves

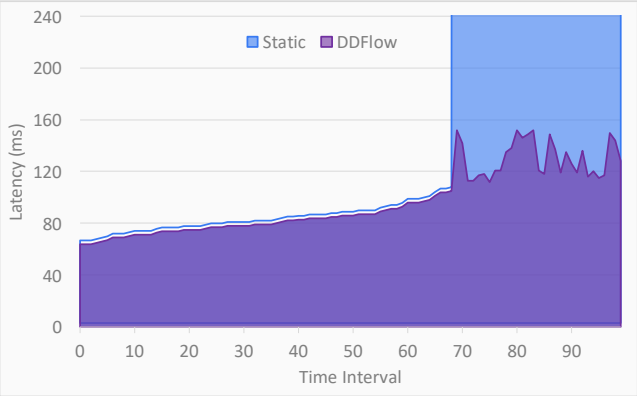

Figure 6: Adaptation during access point failure. DDFlow will switch to an alternate networking mode to maintain device connectivity during access point failure.

in physical space, it switches wireless access points. When a backbone access point fails, the static deployment becomes unable to establish a routing path from drone to client. In DDFlow, upon wireless access point failure the networking system service dynamically switches to a Wi-Fi ad-hoc peer-to-peer communication protocol. With only a single peer-to-peer hop, we can re-establish a routing path to the client and preserve the application. The results are shown in Figure 6.

6 CONCLUSION

We introduce DDFlow, an IoT macroprogramming abstraction that provides a visual tool and unified model for programming distributed applications in a declarative manner. To enable this abstraction, the DDFlow runtime deploys applications in an ad hoc fashion to a heterogeneous network, dynamically adapting to minimize end-to-end latency. Computation is remapped to other devices when a critical node fails or becomes disconnected from the system. Adaptive networking is employed to resiliently adjust to a varying network. Using DDFlow leads to intuitive applications that are easy to understand, yet powerful and robust in deployment.

DDFlow is a system under active development. In future work, we plan to expand DDFlow capabilities with system features including enhanced placement mappings, distributed adaptation, distributed data storage, robust fault tolerance, and safety guarantees.

ACKNOWLEDGMENTS

Research reported in this paper was sponsored in part by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, DARPA, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. 2015. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials* 17, 4 (2015), 2347–2376.
[2] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. 2015. Mininet-WiFi: Emulating software-defined wireless networks. In *Network and Service Management (CNSM), 2015 11th International Conference on*. IEEE, 384–389.
[3] JS Foundation. 2018. Node-RED. https://nodered.org
[4] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor CM Leung. 2015. Developing IoT applications in the fog: a distributed dataflow approach. In *Internet of Things (IOT), 2015 5th International Conference on the*. IEEE, 155–162.
[5] Google. 2018. Edge TPU - Run Inference at the Edge. https://cloud.google.com/edge-tpu/
[6] Google. 2018. Google Vision Kit. https://aiyprojects.withgoogle.com/vision/
[7] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems* 29, 7 (2013), 1645–1660.
[8] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming wireless sensor networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems*. Springer, 126–140.
[9] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. 2013. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*. ACM, 15–20.
[10] Apple Inc. 2018. HomeKit. https://developer.apple.com/homekit/
[11] Amazon Web Services Inc. 2018. AWS DeepLens - Deep learning enabled video camera for developers. https://aws.amazon.com/deeplens/
[12] Valérie Issarny, Georgios Bouloukakis, Nikolaos Georgantas, and Benjamin Billet. 2016. Revisiting service-oriented architecture for the IoT: a middleware perspective. In *International Conference on Service-Oriented Computing*. Springer, 3–17.
[13] Rajnish Kumar, Matthew Wolenetz, Bikash Agarwalla, JunSuk Shin, Phillip Hutto, Arnab Paul, and Umakishore Ramachandran. 2003. DFuse: A framework for distributed data fusion. In *Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM, 114–125.
[14] Ratul Mahajan, Ming Zhang, Lindsey Poole, and Vivek S Pai. 2008. Uncovering Performance Differences Among Backbone ISPs with Netdiff.. In *NSDI*. 205–218.
[15] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. 2017. FogStore: toward a distributed data store for fog computing. In *Fog World Congress (FWC), 2017 IEEE*. IEEE, 1–6.
[16] Luca Mottola and Gian Pietro Picco. 2011. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 19.
[17] Intel Movidius. 2018. Intel Movidius Myriad VPU 2: A Class-Defining Processor. https://www.movidius.com/myriad2
[18] Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The regiment macroprogramming system. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*. IEEE, 489–498.
[19] Francesco Nex and Fabio Remondino. 2014. UAV for 3D mapping applications: a review. *Applied geomatics* 6, 1 (2014), 1–15.
[20] Pushkara Ravindra, Aakash Khochare, Siva Prakash Reddy, Sarthak Sharma, Prateeksha Varshney, and Yogesh Simmhan. 2017. ECHO: An Adaptive Orchestration Platform for Hybrid Dataflows across Cloud and Edge. In *International Conference on Service-Oriented Computing*. Springer, 395–410.
[21] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *arXiv* (2018).
[22] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2018. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*. Springer, 621–635.
[23] Kaixin Sui, Mengyu Zhou, Dapeng Liu, Minghua Ma, Dan Pei, Youjian Zhao, Zimu Li, and Thomas Moscibroda. 2016. Characterizing and improving wifi latency in large-scale operational networks. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 347–360.
[24] Niranjan Suri, Mauro Tortonesi, James Michaelis, Peter Budulas, Giacomo Benincasa, Stephen Russell, Cesare Stefanelli, and Robert Winkler. 2016. Analyzing the applicability of internet of things to the battlefield environment. In *Military Communications and Information Systems (ICMCIS), 2016 International Conference on*. IEEE, 1–8.
[25] S Tibken. 2015. Samsung, SmartThings and the open door to the smart home. *cnet CES* (2015).
[26] Hsiao-Yun Tseng and Sandeep Singh Sandha. 2018. nesl/Heliot. https://github.com/nesl/Heliot/blob/dev/main/placethings/docs/problem_formulation.pdf
[27] Lateef Yusuf and Umakishore Ramachandran. 2010. VirtualConnection: opportunistic networking for web on demand. In *International Conference on Distributed Computing and Networking*. Springer, 323–340.