Modular IoT with dSpace

In preparation - do not distribute

Abstract

While there has been explosive growth in the diversity and availability of IoT devices, systems support for the same has lagged far behind. As a result, existing IoT applications tend to be tedious to implement and inflexible to operate. To remedy this, we propose dSpace, an open and modular programming framework that aims to simplify both the development and operation of IoT applications.

The key argument driving dSpace's design is that an IoT framework should allow developers and operators to focus on the scenarios they want to construct, rather than battle low-level device abstractions. To achieve this, dSpace provides two key building blocks – digivices that implement device control and actuation and digilakes that process IoT data to generate events and insights — together with novel abstractions for composing these building blocks into higher-level abstractions. We apply dSpace to home automation systems and show how developers can easily and flexibly compose these basic abstractions to support a wide range of home automation scenarios.

1 Introduction

Shipments of IoT devices have tripled in the last five years [21, 24] and it is estimated that the revenue for the global smart home market will surpass 1.1 trillion USD by 2023 [23]. The proliferation of these devices promises to make life safer, greener, more convenient, and more comfortable. However it also poses a growing challenge to application developers and homeowners: how do we make intuitive and innovative use of these IoT devices?

We argue that systems support for IoT applications is sorely lacking. As we elaborate on in §2, for programmers, developing IoT applications is often tedious. These devices expose low-level APIs that vary across vendor and device type. Hence, developing an IoT application involves writing a lot of ad-hoc glue code to compose these devices, with little code reuse and few higher-level abstractions. In short, IoT lacks a systems *architecture* that would simplify and accelerate application development.

Not surprisingly then, users often find these applications tedious to configure and/or limited in function. For example, in current home automation products, users specify automation policies by writing rudimentary if-this-then-that rules over per-device state which can be hard to reuse, manage, or reason about as the number of devices and scenarios increase. These challenges lead to vendor-specific, narrow, and vertically integrated solutions. Vendor-specific silos come with the usual problems of lock-in, inflexibility, etc. In addition, because the range of IoT devices is fast outstripping any one vendor, we are seeing multiple disjoint verticals emerging even within a single context such as home automation leaving the user to interact with, and coordinate across, each vertical independently [1, 8, 15, 18, 29, 34, 37, 39, 43].

We argue that IoT needs an open development *framework* that provides: (i) programming abstractions and modularity tailored to the domain; (ii) reusable implementations of commonly used functions; (iii) a common runtime framework (system runtime, CLI, deployment tools, etc.). Specialized frameworks have been deeply influential in other domains – *e.g.,* Hadoop/Spark for data analytics, Tensorflow for ML/AI, Click for packet processing, Ray for reinforcement learning – and we would like to replicate their impact for IoT.

We propose dSpace, a new software architecture for building flexible and configurable abstractions for IoT. With dSpace, an IoT solution (*e.g.,* home automation) is assembled from actuation modules called digivices and data processing modules called digilakes. An individual digivice implements simple actuation logic for automation or optimization, configures other digivices, and interfaces with physical device(s). Each digivice has a set of attribute-value pairs called a model and a piece of code that acts on the model called a driver. The model describes the target and current states of the digivice; and the driver watches for changes in the model and takes actions to reconcile its current state to the target state.

A digilake implements data transformations like converting a video stream to a list of objects in json. Like a digivice, each digilake has a model and a driver, with the difference that the digilake model describes input and output data (or, more typically, pointers to data) and the digilake driver transforms the input data to the output data as specified.

Digivice/lakes can be easily reused and composed to form sophisticated actuation/data processing relationships which we call a digi-graph. A digi-graph describes the flow of control (or "intents") among digivices and the flow of data through digivices and digilakes. dSpace imposes constraints on the construction of the digi-graph to avoid invalid and erroneous compositions. When composed together, digivice/lakes can achieve diverse and advanced usage scenarios.

We apply dSpace to a popular class of IoT applications - home automation. We implement a collection of digivice/lakes leveraging and integrating real-world home IoT devices (from major vendors including Philips, Tuya, Ring, Bose, Dyson, and iRobot [1, 8, 15, 34, 37, 40]) and data processing frameworks (including Tensorflow, OpenCV, Spark, Ray [5, 45, 48, 61]) and show how developers and users can easily compose these to accomplish a range of usage scenarios. While we focus on home automation, we believe our framework extends

naturally to other "smart space" contexts such as campus, corporate, and retail buildings.

The remainder of this paper is organized as follows. We first motivate the need for, and design of, dSpace in §2. §3 and §4 describe the design and implementation of dSpace, respectively. We evaluate dSpace in §5 with a case study that focuses on achieving scenarios in home automation.

2 Motivation and Goals

Done right, a programming framework should simplify the development and use of applications in that domains by addressing its unique challenges and requirements. In this section, we illustrate these challenges and requirements by walking the reader through a series of usage scenarios. As mentioned earlier, we focus on home automation applications and our usage scenarios build up incrementally to arrive at a cohesive end-to-end solution for home automation.

In what follows, we refer to "users" that configure/setup the home automation application. This user could be the actual home occupant but could also be a third-party application service provider that manages the home on behalf of the home occupant, exposing only a highly simplified UI to the home occupant. We simply refer to either as "user".

Scenario 1: Unified control over two lamps in a room. Our first scenario is a very simple one. We have 2 lamps (L1 and L2) in a room and, rather than control each lamp individually, we would like the user to simply be able to specify a "brightness level" for the room. Even this simple scenario faces challenges. First, if the lamps come from different vendors then even though both may have the same underlying set-points, power and intensity, these parameters are exposed through different APIs. E.g., `Geeni` [18] and `Lifx` [29] lamps have different luminous intensity and color schemes.

Challenge. This simple scenario exposes two pain points: First, developers must program to per-vendor APIs and perform unit conversions. Hence the challenge here is addressing the *heterogeneity* in different implementations even for what is logically the same device. Second, allowing users to set the "room brightness level" (vs. setting the intensity and/or power at each lamp individual) requires composing/aggregating devices and exposing the ability to configure these higher-level aggregates (room vs. lamps). In the absence of explicit architectural support for this, each application developer instead reinvents the wheel on how to implement composition and higher-level abstractions.

One might argue that the heterogeneity in this example should be addressed through standardization. E.g., that industry consortia will define a standard schema and API for programming smart lamps. However in practice, waiting for standardization is often problematic: standards can be slow to emerge and evolve [32, 47, 52, 59] and can even impede innovation; *e.g.,* slowing vendors that want to offer novel features to differentiate their products.

Scenario 2: introducing a new multi-color lamp. A lamp L3 that has a new "multi-color" feature (e.g., `Hue` [34]) is now added to the same room. Ideally, this addition should not impact how the user configures room brightness – instead, the home automation app should automatically adjust the intensity/power level of all three lamps to achieve the desired brightness. At the same time, the user should now have the option of configuring a "color" setting for the room.

Challenge. This scenario highlights the need for *dynamic composition* (*i.e.,* upgrading the room abstraction as new devices join/leave the room) and *automatic reconciliation* between the desired state of a higher-level abstraction (room) and its component devices (L1-L3). This scenario also highlights the value of not constraining all devices to a one-size-fits-all schema as might occur with enforcing a standard.

Scenario 3: Motion-triggered configuration. The user now adds a motion sensor (*e.g.,* a Ring Alarm Motion Detector [1]) to the room and would like the "room" to automatically configure its brightness based on the motion sensor.

Challenge. We need the ability to *compose* devices and define actuation loops across these devices.

Scenario 4: Rooms in a home. Next, in addition to configuring at a room granularity, the user would like the ability to easily configure her entire home. E.g., setting the "home" in vacation mode which in turn causes each "room" to enter a power-down mode. Similarly, the user might want to configure at the granularity of floors, bedrooms, etc.

Challenge. We need to go beyond just device-level aggregation and composition. Developers and users must have the flexibility to implement *multi-level/hierarchical composition and control.*

Scenario 5: Robot vacuum by scene. The user now adds a camera and a robot vacuum cleaner (*e.g.,* Roomba [37]) to the room. The user would like the camera's video stream to be processed by an ML system and the robot to be controlled by the output of this ML system. E.g., the robot may start/pause/dock based on whether a dog is in the room.

Challenge. This scenario shows the need to support *advanced processing (e.g., analytics/ML) over IoT data streams.* As in our previous scenarios, we also see the power of composing devices in sophisticated ways (the camera, ML system, and robot within the context of a "room").

Scenario 6: Device mobility. Building on the previous scenario, consider what happens when the robot vacuum moves between rooms. In this case, we want control over the robot vacuum to be "handed over" from one "room" to another. (A similar need arises if the user moves a lamp between rooms.)

Challenge. This scenario once again highlights the need for *dynamic* composition but with additional challenges. In this scenario, the change in composition happens: (i) at runtime, (ii) without user intervention and (iii) the devices being composed depends on the context - i.e., the robot must now be

composed with the camera in the new room (and detached from that in the previous room).

Scenario 7: Access control. The user wants to grant the children control over the lamps in a room but not the cameras. Children can reconfigure room color but not brightness.

Challenge. Supporting *flexible and fine-grained access control.*

Scenario 8: Shared control. So far, the lamps have been configured by a controller associated with the room. Now the user would like to introduce an independent power controller that also configures device settings for energy efficiency. The user's policy is that the power controller only takes over when the room's status is IDLE.

Challenge. A framework must support defining and enforcing policies for shared control over devices. These policies must be constrained to avoid access conflicts, *e.g.,* avoiding situations in which the power and room controller are trying to configure the same device to achieve conflicting outcomes.

Scenario 9: Delegation of control. Our user now wants to outsource portions of home management to different third-party services; for example: (i) outsource control over garden irrigation to a landscaping service, (ii) in the event of an emergency, yield control over the home to a city-run emergency service. Note the difference between the previous scenario#7 and this one. There the home administrator is granting users access to the home's automation application. In this scenario, the administrator is allowing a portion of the home automation application (*e.g.,* emergency or garden controllers) to be *implemented and operated* by third parties. In other words, we're allowing multiple control hierarchies, each potentially operated by different users, to share control over the devices. We believe that such delegation will be a common aspect in future smart homes that allows home owners to outsource certain home management tasks.

Challenge. Delegating control over devices in a manner that is flexible yet prevents access conflicts.

Scenario 10: Service handover. A user in Room-A listening to news on Speaker-1 moves to Room-B which has Speaker-2. The user wants the audio stream to be automatically redirected from Speaker-1 to Speaker-2.

Challenge. Enabling the dynamic and automated transfer of configuration and control state across devices.

Scenario 11: Learning-based automation. In the longer term, rather than manually specifying their desired configurations (*e.g.,* room brightness level), users may have their home automation system automatically learn per-user settings using AI/ML techniques. As with delegation, we believe that incorporating these techniques will be important for simplifying the user experience.

Challenge. Making it easy for developers and users to incorporate state-of-the-art AI/ML techniques.

We design our dSpace framework to address the challenges highlighted above. Recall that our overarching goal is to simplify the development and use of smart-space applications. The complexity in current solutions stems from the fact that they do not adequately address the above challenges. Instead, developers build atop the low-level device-centric APIs and on a case-by-case basis reinvent the wheel on how to compose devices, implement access control, integrate with other tools for data storage, analytics, and so forth. This leads to a specialized and often narrowly-targeted stack. The impact of these limitations is then passed on to users who likewise deal with a set of specialized/disjoint applications with no easy way to customize or extend them. Domain-specific frameworks have successfully simplified and accelerated development in areas such as analytics [4, 5, 51], ML [45], packet-processing [13, 57], and learning [61]. Our goal is to do the same for smart spaces.

3 Design

3.1 Design Rationale and Principles

The design of dSpace rests on a few key design choices that we describe below.

Modularity: the digivice and digilake abstractions. We observe that the central task in the above scenarios is to control or actuate a physical device. In addition, many scenarios involved processing data generated by these devices. Hence, automation involves a control component and, often, a data component. This leads to our two basic abstractions: digivice and digilake. The former implements the control loop that configures device parameters to achieve a desired outcome (*e.g.,* a lamp's intensity, a fan's speed). The latter implement the collection and processing of data streams generated by a device (*e.g.,* a camera's video stream). dSpace applications are assembled from these two modular building blocks.

The decision to decouple data and control processing into distinct abstractions was a deliberate one that we made for two reasons. The first is that the intuitive programming paradigm for device control is different from that for data processing. Device control is well suited to declarative or "intent based" programming models in which the user merely states their desired target state and the actuation logic implemented within the digivice reconciles its current state to the desired one [49]. Data processing on the other hand is generally well served by dataflow or stream-based architectures. Decoupling control and data into separate abstractions allows us to embrace the appropriate paradigm for each.

The second reason for the decoupling is that we believe that processing IoT data to extract insights is best done by leveraging existing analytics and AI frameworks [5, 45, 50, 58, 61]. These systems have been actively developed and researched for over a decade and hence IoT developers should not have to reinvent the wheel. Hence, the role of the dSpace framework should be to make it easy for IoT developers to *integrate* with existing data processing frameworks. Keeping the actuation logic out of a digilake achieves this since it

allows digilakes to essentially be thin wrappers around existing analytics systems (see §3.2). Hence developers of digivice control logic can easily leverage systems like Tensorflow or Spark while keeping their control logic cleanly separated. I.e., control logic and data processing techniques can evolve separately while still leveraging each other.

In our context, a physical device always has an associated digivice and (optionally) a digilake. However, the inverse need not be true. I.e., a digivice/lake need not strictly correspond to a physical device. Instead, they can represent control/data processing at a purely logical level - *e.g.,* a "room" or "home" digivice as we discuss shortly.

Providing higher-layer abstractions via composition. The power of dSpace comes from the ability to compose digivices and digilakes in flexible ways. Digivices can be composed via the *mount* call: when a digivice X is mounted to a digivice Y (denote $Y \leftarrow X$) this means that Y is allowed to configure/control X, *e.g.,* two lamp digivices L1, L2 might be mounted to a room digivice R, as per Scenario-1 (§2).

Such composition serves two purposes. First, it groups devices that have a common context to simplify programming them as an aggregate - *e.g.,* configuring all the devices in the room by iterating through the devices mounted to R. In addition, composition allows us to construct digivices at a higher layer of abstraction, *e.g.,* instead of interacting with L1 and L2, a user or developer can simply configure the higher-level room digivice R, which then takes care of configuring the lower-level L1 and L2. As a different example, to realize Scenario#3, we'd mount the motion detector digivice M to the room R, and have R implement the logic that configures the lamps L1-L3 based on the status of M. In effect, R hides the detailed information about its "southbound" digivices from its "northbound" users/digivices. And one can continue such composition in a hierarchical manner - *e.g.,* a home digivice composed from multiple room digivices.

dSpace also provides a data *pipe* abstraction that allows digilakes and digivices to be composed to form dataflow processing pipelines - *e.g.,* streaming data through three digilakes that are connected in a sequence (via pipes) and implement filtering, compression, and encryption respectively.

Composition addresses a major concern discussed earlier: that existing systems require developers/users to grapple with low-level device-centric abstractions. Instead, composition allows us to raise the level of abstraction for both users and developers. Composition also simplifies reuse since it allows the same building blocks to be composed in different ways to achieve different goals.

Tackling heterogeneity via composition Heterogeneity across vendor API stem from differences in (i) data/protocol formats, (ii) programming style (*e.g.,* imperative vs. declarative), (iii) which config parameters exist (i.e., schema), or (iv) the semantics of how this schema is interpreted. Representing a physical device by its digivice addresses the first two causes but does not help with the latter two. However, composition can help us address the difficult challenge of heterogeneity across vendor schema and semantics. Industry's attempts to address this challenge through standardization have long struggled.[1] Thus our rationale in designing dSpace was to avoid assuming standardized schemas, although we can certainly embrace them should they emerge. But we also want to shield developers from vendor-specific APIs. Hence we'd like to support the emergence of de-facto standards that developers can *opt* to use in a flexible manner.

How do we envision this happening?: Consider lamps L1 and L2 from our Scenario#1 and let's say that their APIs are essentially similar but differ in the details (*e.g.,* different range scales for their intensity parameter, different data formats, etc.). Naively done, the developer of the room digivice R (to which L1 and L2 are mounted) would have to understand both sets of APIs, internalize any differences, etc. In fact, any developer implementing a digivice that uses L1/L2 would have to do the same. Instead, one might write a new "universal" lamp digivice (denoted UL) that instead exposes a "universal" set of configuration parameters and that includes the code to translate from these universal parameters to L1 and L2's parameters. Now, instead of mounting $R \leftarrow L1$ and $R \leftarrow L2$, we'd do $UL1 \leftarrow L1$ and $UL2 \leftarrow L2$ and then mount $R \leftarrow UL1$ and $R \leftarrow UL2$.[2] Hence, R only "sees" a universal lamp abstraction and no longer has to deal with the vendor-specific APIs of L1 and L2. Thus universal digivices act as a layer of indirection, providing a unifying abstraction on top of heterogeneous device implementations. Note that there's no magic here and we're not claiming to have "solved" the challenge of heterogeneous APIs: someone still has to write UL and deal with the idiosyncrasies of L1 and L2's implementations. Instead, our only claim is that we're providing a framework that makes it easy to systematically *reuse* UL and the effort that went into developing UL. Moreover, this approach is flexible: a UL may only support certain vendors' lamps, a developer might choose to use a UL for some lamps but not others (*e.g.,* L3 in Scenario#2), and multiple universal lamps can co-exist thus allowing de-facto standards to emerge without constraining device vendors or developers.

Flexibility: Delegation and multi-rooted control hierarchies. So far, we've described digivices and digilakes that are composed into higher-level aggregates and a control hierarchy; *e.g.,* a home controlling rooms which control lamps, etc. To build an end-to-end home automation application, a developer/user selects the desired digivices/lakes, composes them into her desired hierarchy, and then "programs the home" using the declarative API exposed by the digivice at the root of the hierarchy. This high-level input is then translated into control actions (or "intents") that travel down

[1] For example, despite numerous efforts to standardize configuration of network equipment [32, 47, 59], in practice operators continue to struggle with a hodge-podge of vendor-specific APIs.

[2] As will be clear later, UL1 and UL2 are merely different instances of the same digivice UL.

the hierarchy. Similarly, device status/data travels up the hierarchy potentially triggering new control actions.

However, we do not limit ourselves to just a single static control hierarchy. Instead, a user/developer can define a multi-rooted control hierarchy. This requirement follows from §2: Scenario#8 involved two control hierarchies, one that controls devices in the presence of home occupants and the other for power-savings when occupants are absent. Similarly, Scenario#9 involves three hierarchies separately controlled by home owners, a landscaping service, and an emergency service. Note that allowing multiple control hierarchical to simultaneously configure a device can lead to *access conflicts* in which different controllers overwrite each other's configurations (*e.g.,* a lamp's power level) leading to unpredictable and undesirable outcomes. To avoid access conflicts, multiple control hierarchies may simultaneously read device state but only one control hierarchy is allowed to write/configure the device with programmable "yield" policies that explicitly determine which hierarchy is allowed to control the device at any point in time (*e.g.,* "yield control to the emergency digivice under <. . . > condition").

Multi-rooted hierarchies with explicit yield policies allows dSpace to support delegation and assembling applications from different providers, providing a level of flexibility that goes beyond traditional modular software frameworks that allow code reuse within a single developer/operator context.

Summary. To recap, dSpace's key design decisions are: (i) *digivice* and *digilake* as modular building blocks, (ii) *mount* and *pipe* operators that allow these buildings blocks to be composed into higher-level abstractions, and (iii) multi-rooted hierarchies with explicit *yield* policies for multi-provider assembly and operation.

3.2 Abstractions

We introduce two core abstractions: digivice and digilake. Fig.1(a) depicts digivice and digilake as conceptual models.[3]

Digivice: A digivice enables declarative control over physical devices and other digivices. Each digivice has a set of attribute-value pairs (the *model*) and a piece of code that operates on these attributes (the *driver*). A digivice model includes *control attributes* which describe the desired/target state and the current state of the digivice. The digivice driver watches changes in the digivice model and takes action to reconcile the current state to the target state.

Attributes follow a predefined schema. A value can be a nested collection of attribute-value pairs. Digivices may use other digivices (by mounting them) so that one can build up a hierarchy of digivices. There are verbs, a predefined set of APIs to access the attribute-value pair using their URIs, *e.g.,* get(URI) and update(URI, new_value).

[3]We would like to acknowledge that our visual representation of digivice/lake is inspired by that of the Click [57] authors.

Model and schema: A digivice's set of attribute-value pairs is called the digivice model. The model's schema is identified by a schema identifier called *kind*. All digivice schemas include control attributes, observation attributes, and metadata attributes. Each *control attribute* has an intent field and a status field. The intent fields describe the desired/target state of the digivice and the status fields describe the digivice's current state. The observation attributes provide additional information about the digivice's runtime state (*e.g.,* indicating if a device is in need of repair, whether a room is occupied, etc.). Metadata attributes describe the digivice kind, name, namespace, and other information to be used by the digivice runtime and drivers. One specifies the kind when creating a new digivice and each digivice instance can be uniquely identified by its namespace and name, *e.g.,* `dhome/UL1` (Fig.1(b)). One can reuse predefined digivice models or define new ones for particular domains and use cases.

Driver: Each digivice is associated with a driver. The driver is a piece of code that runs a reconciliation loop: watching changes to the control attributes in the digivice model and taking actions to reconcile the status to the intent. These actions may include calling verbs on digivice models and translating commands to/from physical devices, *e.g.,* an update to `power.intent` is translated and sent to a Philips Hue lamp via the phue [35] library. Digivices of the same kind share the same driver. We will use digivice to refer to a digivice instance including its model and driver. An example digivice - a "digi-lamp" - is shown in Fig.1(b).

Digilake: A digilake enables data processing to be integrated natively with digivices. Similar to digivice, each digilake has a model and a driver. The model is a set of attr-value pairs, among which there are *data attributes* describing the input data (*e.g.,* pointers to data, format, schema etc.) and output data (ditto). The driver transforms the input data to the output data as specified by the input and output attributes. In particular, the data input/output attributes are organized by "ports" with each port describing a data input/output, *e.g.,* "URL" and "Objects" in Fig.1(c). The digilake model also has metadata attributes and observation attributes. The digilake driver is a piece of code that runs data processing: transforming the input data to output data, extracting insights/events from the data, running machine learning model inference etc. As an example, Fig.1(c) shows the model of "Scene" digilake that takes a video stream (from `input.URL`), does object recognition, and updates detected objects (to `output.Objects`).

Shadow: Each digivice/lake is accompanied by a *Shadow*, a set of attribute-value pairs that follows the same schema as the digivice but each value an *access matrix* in which columns represent different verbs, rows represent different roles, and entries represent different access modes. We will describe the access matrix in detail later. There is a one-to-one mapping between the attributes in digivice and those in the Shadow. For convenience, we refer to the attributes in

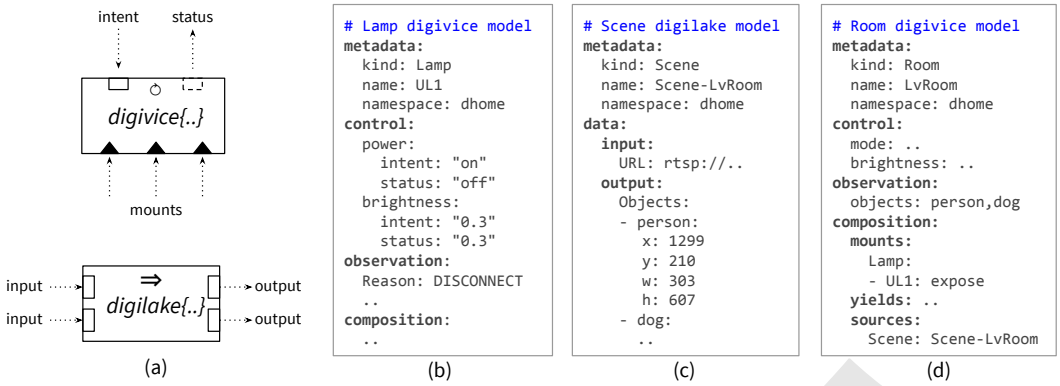

Figure 1. (a) Conceptual models of digivice (top) and digilake (bottom); (b) a Lamp digivice model; (c) a Scene digilake model; (d) a Room digivice model that mounts/control a Lamp and pipes/read data from a Scene.

the ACL as *Shadow Attributes*. §3.5 describes more details on how the Shadow is used for fine-grained access control.

The model and shadow of a digivice or digilake can be accessed by URIs at attribute level granularity (*e.g.,* `/dHome/UL1/control/power`; the backslash can be replaced by ".").

3.3 Composition

Digivices and digilakes can be composed to form digivice/lake hierarchies (or the "control multitree"). Composition takes place when users call the following APIs called *composition verbs* on digivice/lakes:

mount: given a digivice A and B, mount(B,A) allows digivice A's driver to access digivice B's model. Specifically, A can write to the intent fields and read from the status fields of B's control attributes. Mount can have two semantics: "A represents B," *e.g.,* a universal lamp digivice represents a vendor-specific one; and "A aggregates B," *e.g.,* a room digivice includes other lamp digivices. In addition, each mount has a *mode* that can take the value "expose" or "hide". Given digivices A and B: if B is mounted to A, then B can be accessed via URI A.B *if* the mount is set to "expose" mode but not otherwise (if set to hide).

pipe: given a digilake A and B, pipe(A,B) writes A's output to B's input and does so whenever A's output is updated. Note that if A's output is a pointer to data (*e.g.,* a URL to a video stream), only the pointer gets written to B's input. Each digilake can pipe to multiple digilakes (write its data output attributes to their input attributes); but at most one digilake can pipe to an input attribute (*i.e., single-writer-per-port*).

One uses the pipe verb to compose digivices and digilakes to form "data flows" too. Given a digivice A and a digilake B, pipe(A,B) means that the digivice driver can write to the digilake's input attributes/"ports" (*e.g.,* providing a pointer to data); pipe(B,A) allows *digivice A* to read from the digilake B's output (*e.g.,* reading transformed data/pointer to data). A digivice cannot pipe to another digivice directly, though one can achieve the same via an identity digilake (a digilake that map input to output without change). This enforces explicit separation between control/actuation flows and data flows.

yield: composition via the mount verb can lead to a multi-rooted hierarchy. We rely on the yield verb and mount rule (§3.4 discusses the mount rule in detail) to enforce sensible write and mount semantics (§3.4). Specifically, calling the yield verb, *e.g.,* yield(B,A) means that digivice A's driver no long as write access over the model of digivice B. The yielded parent digivice may continue to watch updates from the child's model and act based on them - but can no longer write to the model until the child is unyielded.

Note that one can also use the mount, pipe, and yield verbs to unmount, remove pipe, and unyield respectively by simply setting the corresponding flag at the verb call.

To support composition, each digivice/lake model contains a set of *composition references* where each composition reference tracks the relationship between a pair of digivice/lakes, *e.g.,* mount, yield, and pipe. Specifically, each reference specifies the type of composition (*e.g.,* mount vs. pipe), the target digivice/lake *kind* it refers to, and the target's name and namespace. Fig.1(d) includes a few examples (*e.g.,* `mounts.Lamp` and `sources.Scene`). Developers define composition references in the digivice/lake's model to specify which digivice/lake kinds are compatible and hence can be composed with this digivice/lake.

3.4 Handling Multi-hierarchy

Digivices can be composed into multi-rooted hierarchies. But not all hierarchies make sense! To see this, consider digivice A, B, U - Z above where A, U - Z forms a hierarchy (Fig.2(a)). We want to mount some of A, U - Z to B (imagine adding home digivices in a hierarchy to a new power controller) and/or mount B to these digivices. What are the "mounting rules" we want to follow? Clearly we want to avoid loops, e.g. A → B → Z → X → A, so A → B and B → Z shouldn't be allowed simultaneously: not only do mount loops break the semantics of mount ("to abstract things"), but it can also

lead to erroneous behavior, especially when passing intents and status in a cycle.

Besides loops, we also want to avoid *intent conflicts*. For instance, if we mount X → B and Z → B; given the existence of Z → X, when B updates the intent of X and Z (ignoring A for now), X may in turn update Z which conflicts with B's updates on Z. Though B (or X) can be made aware of the existence of Z → X and handle X and Z with care, this complicates B's logic and likely breaks digivice abstraction, *e.g.,* B may need to know how X's driver updates Z's intent.

What if B wants read-only access to X and Z? Can we allow X → B and Z → B to co-exist in such a case? Note that B will read status from X and Z where X's status may already account for Z's status, leading to status conflicts at B. To see this, say if X is a room, Z is a lamp, and B is a power controller that calculates total power consumption; Z's power consumption gets double-counted if X's power counter includes Z's power consumption already.

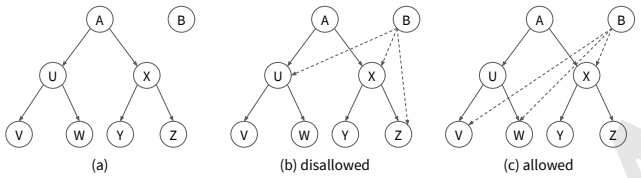

Figure 2. (a): Example digivice hierarchies; (b) and (c): cases where mount is disallowed or allowed.

Our goal is to avoid mount loops, intent conflicts, and status conflicts - that developers and users won't create them at mount and deal with the induced complexity - while still allowing multi-hierarchy in some constrained form. To this end, we decide to enforce the resultant hierarchy to be multitree (or "diamond-free poset" [56]). We describe how we achieve so with *mount rule* in §4. An example of what are allowed and what are not is shown in Fig.2(b) and (c).

The mount rule alone does not solve the multi-hierarchy problem. As long as a digivice has multiple parents, there can be intent conflicts among the parents besides event ordering issues. Our solution to this is simple: for each digivice, we allow only a single writer at any moment and this is achieved by the *single-writer/yield policy*. That is, for every digivice that is mounted to multiple parents there is a yield policy embedded in the digivice's model to specify when and how to yield (*e.g.,* if the room is set to IDLE mode, yield it to the power controller). Yield policies may be specified at development time and enabled by the user at mount time, or rewritten dynamically at run time by the user. It is the combination of mount rule and yield policy that enforces sensible semantics of digivice hierarchies.

3.5 Access control

In dSpace, we address access control by implementing Role-Based Access Control (RBAC) based on digivice shadow

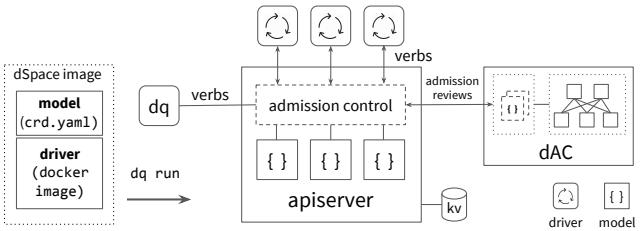

Figure 3. The overall architecture of dSpace.

aforementioned (§3.2). RBAC is a classic access control mechanism and has been widely used in production systems [38, 42, 54]. In brief, RBAC associates each user/identity with roles and each role with access rights. In dSpace, each digivice/lake driver is assigned with an identity and the identity is associated with some role(s) where the role to access rights association is tracked by the Shadow.

When a digivice/lake is accessed by a verb on an attribute, the attribute's and its invoking ancestors' Shadow attributes are checked for whether the entry in their access matrix, given the verb and the role (of the user who calls the verb), is non-zero. The invoking ancestors of an attribute X are those attributes included in the URI to access X, *e.g.,* Z and Y in /Z/Y/X. A new role is associated to the digivice/lake driver, and, by default, the role has complete access to the digivice/lake. There is an Admin role that simply has complete access to all digivice/lakes in the system. Admin can also create new roles with different access right combinations.

4 Implementation

We implement a prototype of dSpace (runtime and client lib) with 4,143 LoC in Go, excluding code and libraries we reused. This section presents the details of this prototype, and in addition, a collection of example digivices and digilakes we built with the prototype for home automation use cases.

4.1 Overview

The dSpace prototype comprises four components, as shown in Fig.3: a command line interface *dq*, an *apiserver* serving attribute-value pairs, a *dSpace admission controller* (dAC) that enforces composition rules and policies, and dSpace applications (models and drivers). We refer to the apiserver and dAC as the digi-runtime (or runtime). Dq and dSpace applications are written using digi-lib (or client lib).

We reuse the k8s apiserver [6] as a drop-in implementation as we find it a natural choice to host attribute-value pairs (i.e., models and shadows) and allow them to be accessible over REST APIs. The apiserver stores the attribute-value pairs in a persistent k-v store (typically etcd). Internally, each model at the apiserver is associated with an event queue; updates to the model are first appended to the queue and then applied in FIFO order to the model. Drivers use an asynchronous Watch API to subscribe to the changes of a model. These changes are sent to the driver as events that get appended to

a driver-local event queue. The worker thread (a goroutine) fetches events from the queue in FIFO order and invokes the driver's event handlers, *e.g.,* a reconciliation loop.

A driver invokes verbs (provided by digi-lib) to interact with the apiserver over REST/HTTP. Before serving and/or executing the verb requests, the apiserver runs a series checks on whether the verb is valid, including correctness of syntax and semantics, sufficient access rights from caller, whether the proposed changes violate composition rules etc. Many of these checks (*e.g.,* the latter two) are performed at dAC. This is done by having apiserver submit an "admission review" to the dAC where dAC can decide whether to reject the verb.

Composition verbs, *i.e.,* mount, yield, and pipe, are implemented as APIs at the digi-lib. The verbs are translated to an "update" access verb that updates the corresponding composition reference before being sent to the apiserver. In other words, apiserver and dAC only see access verbs. This implementation choice allows dSpace to be backward compatible with existing k8s tools (*e.g.,* `kubectl`) widely used.

Three sets of checks on composition verbs are performed during admission control: compatibility, access rights, and composition rules. Compatibility checks confirms whether two digivice/lakes are compatible, *i.e.,* whether one can be mounted/piped to another. It happens at apiserver during admission control in the form of schema validation - an invalid composition will appear to be a model schema violation and rejected by apiserver (§4.2). Access right checks happen at dAC by referring to the shadows to authorize the verb. Finally, composition rules make sure the resultant digi-graph is still valid after applying the proposed changes and fail the verb if otherwise. dAC uses composition references to construct the digi-graph.

dSpace runtime (apiserver and dAC) and application drivers can be run on any server(s) that supports running Kubernetes. Because the runtime components communicate over REST, they can run on separate machines. For instance, drivers that have special software/hardware requirements (*e.g.,* digilakes that require GPUs) can be in a separate and specialized machine. We envision typically the servers are part of clusters running in the cloud or edge by some application providers for smart spaces. They can also be run on on-prem devices for privacy, cost, or performance concerns. We describe more on the deployment issues in §4.4. Next, we present more details about the prototype.

4.2 Runtime

Digi-runtime consists of an apiserver (AS) and a dSpace admission control (dAC). AS hosts digivice/lake models and shadows. We use the Kubernetes (k8s) apiserver [6] as the off-the-shelf, best-of-breed option for AS, although our designs in §3 are not tied to it. In a nutshell, the k8s apiserver is a server hosting API objects. Each API object is a set of attribute-values pairs where attributes follow a predefined schema (like dSpace). One can interact with apiserver via REST operations to create, read, update, and delete API objects. As part of the k8s container orchestrator, the k8s apiserver has been traditionally used for serving API objects used in datacenter operations.

The underlying data storage for apiserver is etcd [17], a strongly consistent k-v store, by default. Alternative data storage are possible, *e.g.,* `sqlite`; since dSpace does not interact with the data storage directly (via apiserver only), the choices and implications of different storage is beyond the scope of this paper except for nuances in deployment (§4.4).

dAC is where dSpace enforces policies and composition rules. To do so, dAC is interfaced with AS via webhooks (as part of k8s's Dynamic Admission Control mechanism [14]) by which AC can intercept, review, authorize, and in some cases modify the operations towards API objects before the verbs are applied. The webhook mechanism is implemented by having the AS send *admission reviews* to the dAC where each admission review includes information about the verb, the caller, the caller's role, the original version of the object, the new version after the proposed change etc.

With this information (plus additional information on the apiserver that dAC actively acquires; described in Phase 3), dAC is able to enforce the composition rules (*e.g.,* multi-tree, single-writer, single-writer-per-port) and policies (*e.g.,* access control, dynamic mount/yield/pipe). In addition to dAC, k8s apiserver has a set of built-in admission controllers that we reuse. Admission controllers are "chained" together so that one can think of admission control as a logical AND statement where each item in the statement is an admission controller. Overall, the following steps/phases are performed before a verb can be executed (if all phases succeed).

Phase 1: Authentication and schema validation. This step first checks whether the user (their token/identity file) is recognized and then whether the request is syntactically correct. It includes checking whether the verb is known, whether the model's schema is valid after the verb is performed, *e.g.,* having all the required attributes, the right value types, and no unrecognized attributes. We rely on existing k8s admission controllers [41] to perform this step. If the verb passes Phase 1, the AC submits an admission review (described in §4.1) to dAC for follow-up checks. The verb is rejected otherwise. Upon receiving the admission review, dAC performs the following (phases of) operations:

Phase 2: Shadow lookups. To authorize access verbs, dAC fetches the shadow of the target model from the apiserver (we will talk about caching later) and performs shadow lookups. If the lookups/checks fail, the verb is rejected.

Phase 3: Handling composition. This step guarantees the resultant digi-graph satisfies properties discussed in §3.4, namely, digivices should form a multi-tree with a single-writer on each digivice; and digilakes should follow single-writer-per-port policy. To achieve the former, dAC validates the mount verb against a *mount rule* and *a single-writer*

rule. Specifically, dAC identifies the digivice hierarchies on the digi-graph by tracking the mount references. A digivice hierarchy is a tree where each node in the tree is a digivice. Each (unidirectional) edge represents the mount relationship between a parent node (the mounting digivice) and one of its child nodes (the mounted digivice).

We now explain how the mount rule works. Given definitions: (i) A digivice is said to be *part of* the hierarchy if it is a node in the tree; (ii) A digivice is said to *join* a hierarchy if it gets mounted to another digivice part of the hierarchy; (iii) if a digivice D joins a hierarchy H, then all D's descendants also join H; we enforce the following rule at every mount:
Mount rule: *A digivice cannot join a hierarchy that it or any of its descendants is part of already.*

The mount rule ensures that digivices form multitree(s) and is what dAC checks for verbs that update mount references. The verb is rejected if the mount rule will be violated. Finally, as optimizations, dAC caches states from apiserver including: i. *Shadow cache:* A local, write-through cache of the shadows at the dAC. Upon restart, dAC reconstructs the cache from apiserver. ii. *Digi-graph cache:* dAC constructs and caches the digi-graph based on the composition references in apiserver and is reconstructed upon restart.

4.3 Client Library

The client library (digi-lib) contains utility functions for developing digivice/lake models, drivers, and CLIs. We reuse existing k8s libraries including client-go and controller-runtime libraries [9, 27] with minor changes and simplifications. The main changes are as follows:
Model generation: digivice/lake models can be defined in Go structs that are used to generate Custom Resource Definition (CRD) [11] to register new API objects in the apiserver. We extend existing code-generators in k8s [10] to generate CRDs and shadows.
Verbs: digi-lib converts composition verbs to access verbs (§4.2) that update composition references. Since Go currently does not support generic, the digi-lib handles composition verbs by serializing the model received from apiserver to JSON and manipulating the JSON object directly.
Event Watch API: the Watch API takes CRUD events for a digivice model and enqueues a reconciliation request to the driver program to trigger the reconciliation. We extend the k8s Watch API to add a pair of muxer and demuxer for events when they are enqueued and dequeued respectively. This helps route the event to the corresponding event handler of the driver, *e.g.,* when a digivice has many mounted/children digivices that it needs to watch for changes.

4.4 Putting Pieces Together

In addition to digi-runtime and digi-lib, there are a few other components as part of the dSpace that developers and users can leverage which we briefly mention as follows:

Dq: the CLI of dSpace. It is the command line interface for users and developers to interact with the digi-runtime.
dSpace image: We refer to the model-driver pair of a digivice/lake a dSpace Image (DSI). Besides the model identifier/kind that one can use to locate a DSI, all DSIs are named by their content, which is approximated by the SHA2 hash of the model-driver pair themselves. DSIs are implemented based on docker image [12].

With the modules and components described in this section, we discuss the development, deployment, and how user interacts with dSpace in what follows.

Development: With the modules described in this section, developers take the following steps to develop and distribute a digivice (and digilake similarly): i. write a digivice model in a model.yaml file and run code-generator to generate the corresponding custom resource definition (CRD [11], a mechanism to register new types of API objects on k8s apiserver), code snippets, and type definitions (optionally, in the case of strongly-typed languages such as Go); ii. write a digivice driver in a language of choice (*e.g.,* one that the library to access physical devices uses); iii. use dq to compile the two into a dSpace image. Dq also generates the shadow of the model and includes it in the image (not shown in Fig.3).

Deployment: We envision application providers, *e.g.,* home automation SaaS, will provision compute clusters at cloud or edge to run dSpace runtime (say for home automation). The dSpace runtime components (apiserver, dAC, drivers) can be deployed on most k8s distributions (v1.17-), including minikube [31], k3s [30], and hosted k8s cluster on the cloud [3]. The dSpace models and drivers are likewise hosted/running in those remote clusters. The dSpace drivers are run as docker containers inside k8s pods [28]. dSpace may also be deployed on an on-prem device (*e.g.,* a Raspberry Pi) depending on the desired privacy-manageability trade-offs.

Using dSpace: Users will start digivice/lakes via the CLI (*e.g.,* dq), submitting the model (CRDs) and running the drivers (Fig.3). Users will connect physical devices to dSpace by performing an on-boarding process. This is also required for today's IoT devices and frameworks and typically involves peering the device with an app or hub. Similar to the commercial IoT frameworks (*e.g.,* Apple Home/HomeKit [7]), we assume there is a local hub (at LAN) to perform "phone home" operations to the remote drivers to do initial peering. After the setup, users compose digivice/lakes as needed.

4.5 Example Digivice/lakes

We implemented a set of digivice/lakes for home automation as an example dSpace (we call it "dHome"). The case study in §5 uses these digivice/lakes to implement the scenarios from §2. Table 1 summarizes some of these digivices. These digivice/lakes are implemented in different programming languages and typically involve tens of LoC depending on the logic to be implemented. For digivices that are directly interfaced with physical digivices (*e.g.,* 2-3, 5-8), the choice of

ID	Digivice/lake	Control	Data	Observation	Mounts	Pipes	Language/Tools
1	GeeniLamp	Power, Brightness	N/A	Reason	N/A	N/A	TypeScript (tuyapi)
2	LifxLamp	Power, HSBK	N/A	Reason	N/A	N/A	Go, Python (lifxlan)
3	PhilipsLamp	Power, CIE 1931 Color Spaces	N/A	Reason	N/A	N/A	Python (phue)
4	UniversalLamp	Power, Brightness	N/A	Reason	2, 3	N/A	Go
5	Room	Modes, Brightness	N/A	Presence, Objects	1-4, 6-11	12-14	Go
6	MotionSensor	Power, Sensitivity	N/A	Triggered	N/A	N/A	Python (ring)
7	Speaker	Power, Play, Volume	N/A	Reason	N/A	N/A	Go
8	Plug	Switch	N/A	Reason	N/A	N/A	Typescript (tuyapi)
9	Fan	Power, Airflow speed	N/A	Reason	N/A	N/A	Python (libcoolpurelink)
10	RobotVacuum	Power, Mode	N/A	Reason	N/A	N/A	Python (dorita980)
11	Camera	Power	N/A	Reason	N/A	12	Python
12	Scene	N/A	in: url; out: json	Reason	N/A	5, 15	Python (opencv)
13	Stats	N/A	in: json; out: json	Reason	N/A	5, 15	Python (pyspark)
14	Imitation	N/A	in: json; out: json	Reason	N/A	5, 15	Python (ray)
15	Home	Modes	N/A	Reason	5	12-14	Go

Table 1. Example digivices and digilakes in dHome.

programming language is decided by the existing third-party drivers for that device (Table 2). We pick a few representative digivice/lakes and describe them as follows.

Geeni/Lifx/PhilipsLamps: These digivices interact with the physical lamps directly via vendor or third-party APIs. The models expose vendor-specific power, brightness, or color scheme setup as shown in Table.1.

UniversalLamp: This universal digivice translates brightness level defined in percentage to vendor-specific brightness configurations, *e.g.,* unit name (brightness to intensity) and unit conversion (percentage to bitmap).

Scene: The scene digilake transforms a video stream given the URL in its data input attributes to a list of objects (and their locations in the video frame) that one can read from its data output attributes. The driver uses OpenCV, and Yolov3 to process the video stream and perform object recognition.

Room: The room digivice provides control attribute Modes and Brightness where each Mode maps to a set of pre-defined set-points for each digivice mounted to it. The room can read from a scene digilake (described below) to read the objects in the scene and update its event attributes based on the room's occupancy, *e.g.,* the presence of humans.

Stats and Imitation: the Stats digilake reads a data stream of set-point values (*e.g.,* energy readings in watts) in its input, calculates the moving average, and places the results in the output. We use PySpark to implement Stats (doing a *rangebetween*) [5]. The Imitation digilake is an (strawman) example of integrating learning to digivices. We use a simple behavior cloning algorithm in Ray's RLLib [36] to learn a simple policy of updating home's mode based on rooms' Observation.presence given past <time of the day, room presence, home mode> tuples as training data.

5 Evaluation and Case Study

Experimental Setup Table 2 describes (a subset of) physical devices studied and used to build the scenarios (in dHome). Most of these devices (8/9) can be accessed via local RPCs. The GEENI lamps, for instance, can be accessed via the Tuya IoT platform where one can use a third-party library (tuyapi) to access the device on LAN. The one exception is the Bose ST10 speaker. The speaker can only be accessed via the vendor (Bose) cloud and hence RPC calls have to be sent to/from the vendor's server and then relayed to/from the device. We run the digi-runtime and drivers on a Lenovo Thinkcentre M720 (Intel Core i5-8400T, 6 cores). We use minikube to run the apiserver (k8s version 1.18.2) on Debian 10.

Device Type	Vendor	Model	Library	Access
Light bulb	GEENI	LUX800	Tuyapi	LAN
Light bulb	LIFX	Mini	Lifxlan	LAN
Light bulb	Philips	Hue	Phue	BS/LAN
Motion sensor	Ring	Ring kit	Ring-client-api	BS/LAN
Camera	Wyze	WYZECP1	RTSP Stream	LAN
Robot vacuum	iRobot	Roomba 675	Dorita980	LAN
Speaker	Bose	ST10	Soundtouch	VC
Fan | Heater	Dyson	HP01	Libpurecoollink	LAN
Plug	Teckin	SP10	Tuyapi	LAN

Table 2. IoT devices we support in the dHome prototype. BS: basestation. VC: vendor cloud.

5.1 Response latency

We measure the performance of dSpace in terms of the *response latency:* the time a user submits its intent to the time the user observes its fulfillment, typically as a physical effect. The response latency is measured and estimated as follows. We use a video camera to record (i) the command line screen where we submit the configuration (ii) the managed device. We kept the camera's frame-rate fixed (30fps; i.e., each frame accounts for ≈33ms). We ran and recorded the experiment, then played the video frame by frame counting the number of frames between the frame in which the shell command is executed and the one in which the intent is fulfilled.

We benchmark five devices from Table 2: light bulbs in the first three rows, the Dyson fan/heater (including a panel displaying heat setpoint on its body), and Scene + GeeniLamp (lamp turns on when the Scene has any humans; the Scene is fed with videos stream from the Cam digivice on a Wyze camera). We ran the experiment 10 times. Fig.5 shows the

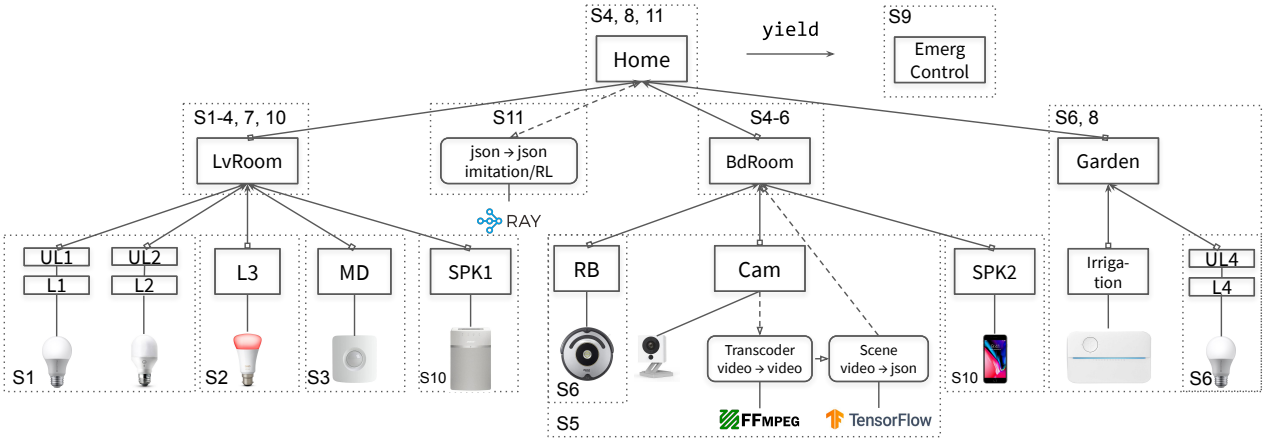

Figure 4. Summary of scenarios in dHome.

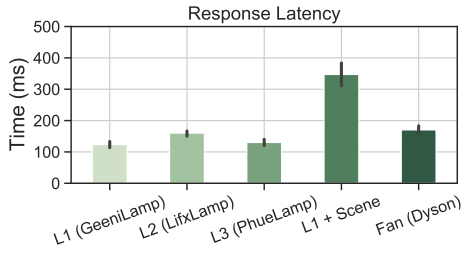

Figure 5. Response latencies of five actuations.

response latencies for the devices/scenarios. All scenarios receive sub-second response latency. The L1 + Scene sees high latency due to the time for object recognition ($\approx 200ms$).

5.2 Case Study: dHome

In what follows, we describe how we (as end users would, via some UI/wrapper around dq) build the scenarios mentioned in §2 with devices and digivice/lakes in Table 2 and 1.

```
# create lamp digivices L1 and L2
dq run -f geeni.yaml # L1
dq run -f lifx.yaml  # L2

# create and use universal lamps
dq run -f geeni-ul.yaml # UL1
dq run -f lifx-ul.yaml  # UL2
dq mount "L1 UL1, L2 UL2"

# create a room and add lamps
dq run -f lv-room.yaml  # LvRoom
dq run -f phue.yaml     # L3
dq mount "UL1 LvRoom, UL2 LvRoom, L3 LvRoom"
..
# create and use Transcoder and Scene
# digilakes with Cam and BdRoom
dq run -f txcoder.yaml scene.yaml
dq pipe "Cam | Transcoder | Scene | BdRoom"
```

Snippet 1. Commands to create and compose digivice/lakes.

S1-S2, Basic Compositions: given two smart light bulbs from vendor Geeni and Lifx, we run digivice L1 and L2 with dq (after the on-boarding process described in §4.1) for each of these lamps (Line 2-3 in Snippet.1). To use L1 and L2 with a Room digivice (LvRoom), we mount L1 and L2 to universal lamps (Line 6-8 in Code Snippet.1). Snippet.2 shows the reconcile function of L2. We create another lamp L3 (device from Philips) and then mount all three lamps to LvRoom (Line 11-13 in Snippet.1). We then add a Ring motion sensor and its digivice (MD) to LvRoom (Fig.4). LvRoom checks `MotionSensor.triggered` to set the brightness of `LvRoom.Mounts.Lamps`.

```
func (r *ReconLifx) Reconcile(req reconcile.Request) (
    reconcile.Result, error) {
    // fetch the LifxLamp instance
    l := &dhomev1.LifxLamp{} ..
    // diff or no action
    if reflect.DeepEqual(l.Intent, l.Status) {
      return reconcile.Result{}, nil
    }
    // post to the endpoint
    _, err = (&dev.Device{
      Endpoint: l.Endpoint,
    }).Post(dev.Request{
      "power": l.Intent.Power, ..
    }) ..
}
```

Snippet 2. Reconcile function of LifxLamp digivice.

Advanced Compositions (S5-S11): We select two examples. First, we want to use a Roomba vacuum robot in living room. The Roomba robot out-of-the-box: i. cannot pause cleaning whenever any humans in the room; ii. it has a localization camera but the localization algorithm does not function well under dark environments. We compose the digi-graph (Line 17-18) to address the above, leveraging a camera supplying video stream. The control/actuation logic is depicted in Snippet.3.

Second, we develop a sample emergency controller (Fig.4) that sets all digivices mounted to it to a power of "off" or a mode "idle" (in the case of rooms). We define a yield policy, shown in Snippet.4 that will yield any digivices (if

`Home.observation.alarm` is "on") that are mounted and active to Home, and activate their mounts to EmergControl (*i.e.,* unyielding them) to delegate control.

```
# Scene digilake (in Python)
if self.moved(frame):
    frame = self.detect(frame)
    self.scene.update() ..
// Room digivice (in Go)
_, ok := room.Scene.Objects["human"]
if ok {
  room.Observation.Presence = true
  room.Roomba.Intent.Mode = "pause"
} ..
if room.Roomba.Intent.Power == "on" {
    room.adjustBrightnessRoomba()
} ..
```

Snippet 3. Room with Scene

```
metadata:
  kind: YieldPolicy
  name: emergency
  namespace:
    default
policy:
  condition: |
    source.
     observation.
     alarm=on
  source: Home
  target:
    EmergControl
```

Snippet 4. Yield

6 Discussion and Related Work

The IoT landscape today includes many closed-source vendor products. We view these solutions as taking one of three forms: device-centric, app-centric, or infrastructure-centric. *Device-centric* solutions are ones in which the vendor starts with their own device(s) and then offers optional add-on software and cloud services for these devices. Such vendors are typically device manufacturers such as Philips, LIFX, Geeni, BOSE, Dyson, and WYZE [8, 15, 18, 25, 29, 34, 43]. We use the term *app-centric* to refer to solutions in which the vendor has built an application and they expose APIs that allow heterogeneous devices to integrate with this application or service. Such vendors are typically smartphone companies and/or cloud providers such as Apple, Amazon Alexa, Google Nest, and Xiaomi etc. [2, 7, 19, 44]. Finally, infrastructure-centric solutions are ones in which a vendor offers its infrastructure (and corresponding infrastructure-level services) as a platform on which clients can integrate different devices and construct and run IoT applications. All the major cloud providers offer such platforms as well as some standalone SaaS vendors such as IFTTT [22]. A canonical example is AWS IoT that allows tenants to construct IoT applications by leveraging a variety of AWS services as building blocks – e.g., AWS GraphQL (to model control flow), MQTT (for pub-sub), AWS Lambda (to run custom code), AWS Greengrass (as infrastructure).

Each approach solves an important problem but also imposes significant constraints on users. Device-centric solutions do solve the problem of tedious/low-level device abstractions but limit users to that particular vendor's devices. App-centric solutions on the other hand do allow diversity in devices but lock users and developers into a particular application stack; e.g., Home/HomeKit on Apple devices and iOS, Google services on Nest devices etc. Similarly, infrastructure-centric solutions simplify the challenge of operationalizing an IoT application but lock the user into a particular cloud provider's infrastructure and software services (e.g., one cannot obviously run an AWS IoT solution on Google Cloud). By contrast, dSpace targets flexibility on all fronts: the choice of devices used, the application logic by which they are composed, the infrastructure on which they run (which may include one or more cloud providers), and the software stacks and services that they leverage (e.g., Spark, Tensorflow, Ray, OPA, or a custom implementation).

In addition to the above proprietary products, there are multiple industry consortia developing open source solutions: EdgeX Foundry [16], KubeEdge [26], Home Assistant [20], and OpenHAB [33]. Based on the available documentation and codebases, these efforts currently target lower-layer issues of device abstraction and unification and container orchestration in edge deployments. These systems do not offer the high-level abstractions that dSpace provides to simplify *application* development: e.g., the digivice/digilake modularity, declarative control, delegation, hierarchical composition, ease of integration with AI/analytics, etc. In this sense, we view our efforts as complementary and hope to leverage existing efforts (e.g., EdgeX drivers for the lowest-level digivices in dSpace, or KubeEdge as we currently do Kubernetes components).

Related Research. dSpace is inspired by prior research on modular frameworks in other domains - *e.g.,* Ray [61] for learning, Malt [60] for network management, Click [57] for packet-processing, Spark/Hadoop [4, 5] for analytics, Tensorflow [45] for AI/ML. dSpace differs from these in the nature of its abstractions and features which result from the needs of IoT and smart spaces (*e.g.,* the focus on device actuation, vendor heterogeneity, delegation, etc).

In terms of research on IoT apps: dSpace is closest to prior work on HomeOS [53], Brick [46] and XBOS [55]. These systems implement specific services and functions that are useful in developing home or building management apps (respectively): e.g., a time-series data store, an app store, and a query processor. In other words, these systems are specific instantiations of an IoT stack. By contrast, dSpace is a *framework* that allows developers to implement, compose, and integrate services such as they present but does not dictate any particular set of services. Hence we view these efforts are complementary.

References

[1] 2020. Alarm Motion Detector. https://shop.ring.com/products/alarm-motion-detector-v2.
[2] 2020. Amazon Alexa. https://developer.amazon.com/en-US/alexa.
[3] 2020. Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks/.
[4] 2020. Apache Hadoop. https://hadoop.apache.org/.
[5] 2020. Apache Spark. https://spark.apache.org/.
[6] 2020. apiserver. https://github.com/kubernetes/apiserver.
[7] 2020. Apple Homekit. https://developer.apple.com/homekit/.
[8] 2020. Bose SoundTouch API. https://developer.bose.com/bose-soundtouch-api.
[9] 2020. client-go. https://github.com/kubernetes/client-go.

[10] 2020. code-generator. https://github.com/kubernetes/code-generator.
[11] 2020. Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.
[12] 2020. docker image. https://docs.docker.com/engine/reference/commandline/image/.
[13] 2020. DPDK: Data Plane Development Kit. https://www.dpdk.org/.
[14] 2020. Dynamic Admission Control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.
[15] 2020. Dyson Pure Cool link library. https://github.com/CharlesBlonde/libpurecoollink/.
[16] 2020. EdgeX Foundry. https://www.edgexfoundry.org/.
[17] 2020. Etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.
[18] 2020. Geeni Smart Lighting. https://mygeeni.com/collections/lighting.
[19] 2020. Google Nest, build your connected home. https://store.google.com/us/category/connected_home.
[20] 2020. Home Assistant: Open source home automation that puts local control and privacy first. https://www.home-assistant.io/.
[21] 2020. How IoT devices and smart home automation is entering our homes in 2020. https://www.businessinsider.com/iot-smart-home-automation.
[22] 2020. IFTTT: Everything works better together. https://ifttt.com/.
[23] 2020. Internet of Things (IoT) - Statistics & Facts. https://www.statista.com/topics/2637/internet-of-things/.
[24] 2020. iPropertyManagement: Smart Home Statistics. https://ipropertymanagement.com/research/iot-statistics.
[25] 2020. iRobot Ready to Unlock the Next Generation of Smart Homes Using the AWS Cloud. https://aws.amazon.com/solutions/case-studies/irobot/.
[26] 2020. KubeEdge: A Kubernetes Native Edge Computing Framework. https://kubeedge.io/.
[27] 2020. Kubernetes controller-runtime Project. https://github.com/kubernetes-sigs/controller-runtime.
[28] 2020. Kubernetes Pods. https://kubernetes.io/docs/concepts/workloads/pods.
[29] 2020. LIFX Smart Home Light. https://www.lifx.com/.
[30] 2020. Lightweight Kubernetes: The certified Kubernetes distribution built for IoT & Edge computing. https://k3s.io/.
[31] 2020. Minikube: Run Kubernetes Locally. https://github.com/kubernetes/minikube.
[32] 2020. OpenConfig: Vendor-neutral, model-driven network management designed by users. https://www.openconfig.net/.
[33] 2020. openHAB: empowring smart home. https://www.openhab.org/.
[34] 2020. Philips Hue. https://www.philips-hue.com/en-us/personal-mood-lighting.
[35] 2020. phue: A Python library for Philips Hue. https://github.com/studioimaginaire/phue.
[36] 2020. Ray rllib. https://github.com/ray-project/ray/tree/master/rllib/agents/marwil.
[37] 2020. Roomba Robot Vacuums. https://www.irobot.com/roomba.
[38] 2020. SELinux/Role-based access control. https://wiki.gentoo.org/wiki/SELinux/Role-based_access_control.
[39] 2020. Teckin home. https://www.teckinhome.com/.
[40] 2020. Tuya IoT Platform. https://www.tuya.com/.
[41] 2020. Using Admission Controllers. https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/.
[42] 2020. Using RBAC Authorization. https://kubernetes.io/docs/reference/access-authn-authz/rbac/.
[43] 2020. Wyze. https://wyze.com/.
[44] 2020. Xiao Mi Smart Home. https://xiaomi-mi.com/mi-smart-home.
[45] Martín Abadi et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proc. USENIX OSDI*.
[46] Bharathan Balaji et al. 2016. Brick: Towards a unified metadata schema for buildings. In *Proc. ACM BuildSys*.
[47] Martin Bjorklund et al. 2010. YANG-a data modeling language for the network configuration protocol (NETCONF). (2010).
[48] Gary Bradski and Adrian Kaehler. 2000. OpenCV. *Dr. Dobb's journal of software tools* (2000).
[49] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* (2016).
[50] Benoit Dageville et al. 2016. The snowflake elastic data warehouse. In *Proc. ACM SIGMOD*.
[51] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[52] Steve Deering, Robert Hinden, et al. 1998. Internet protocol, version 6 (IPv6) specification.
[53] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Brush, Bongshin Lee, Stefan Saroiu, and Victor Bahl. 2010. The home needs an operating system (and an app store). In *Proc. ACM HotNets*.
[54] David Ferraiolo, Janet Cugini, and D Richard Kuhn. 1995. Role-based access control (RBAC): Features and motivations. In *Proc. ACSAC*.
[55] Gabriel Fierro and David E Culler. 2015. Xbos: An extensible building operating system. In *Proc. ACM BuildSys*.
[56] Jerrold R Griggs, Wei-Tian Li, and Linyuan Lu. 2012. Diamond-free families. *Journal of Combinatorial Theory, Series A* 119, 2 (2012), 310–322.
[57] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.
[58] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078* (2012).
[59] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.
[60] Jeffrey C Mogul et al. 2020. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *Proc. USENIX NSDI*.
[61] Philipp Moritz et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *Proc. USENIX OSDI*.