dSpace: Composable Abstractions for Smart Spaces

Silvery Fu and Sylvia Ratnasamy
UC Berkeley

Abstract

We present dSpace, an open and modular programming framework that aims to simplify and accelerate the development of smart space applications. To achieve this, dSpace provides two key building blocks — *digivices* that implement device control and actuation and *digidata* that process IoT data to generate events and insights. In addition, dSpace introduces novel abstractions – *mount*, *yield*, and *pipe* – via which digivices and digidata can be composed into higher-level abstractions. We apply dSpace to home automation systems and show how developers can easily and flexibly leverage these abstractions to support a wide range of home automation scenarios. Finally, we show how the dSpace concepts can be realized using a microservices-based architecture and implement dSpace as a Kubernetes-compatible framework.

CCS Concepts: • **Software and its engineering → Abstraction, modeling and modularity**.

Keywords: IoT, smart spaces, design principles, framework

ACM Reference Format:
Silvery Fu and Sylvia Ratnasamy. 2021. *dSpace*: Composable Abstractions for Smart Spaces. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–28, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3477132.3483559

1 Introduction

Living spaces – homes, offices, retail locations – are being transformed by the proliferation of IoT devices. For example, shipments of IoT devices have tripled in the last five years [2, 4] and the global smart home market is projected to surpass 1.1 trillion USD by 2023 [3]. Given these trends, it is important that we have the right systems support for building smart space applications. This need has been recognized by both research [1, 15, 58, 62, 65, 66, 72–74] and industry solutions [11, 29, 44, 45]. These address the challenge of heterogeneity across devices: they propose unified device abstractions [23, 44], common system services [23, 29, 62, 72], common policy languages [44, 45], and so forth. They also address the challenges around discovery, localization, and networking of such devices [23]. Taken together, this seminal work tackled an essential first step – making it easier to discover, network, and program *individual devices*.

However, looking ahead, we anticipate a future in which smart spaces will be increasingly ubiquitous and increasingly sophisticated: incorporating a greater variety of devices and use-cases that are customized to individual needs, integrated with AI/analytics, and capable of advanced operations such as delegating management to trusted 3rd parties (*e.g.,* a gardening service, emergency services, appliance maintenance services, etc.). From a systems perspective, the challenge in realizing the above vision is to make it *easy* for developers to support such flexibility. Unfortunately, today's IoT frameworks fall short in this regard. As we elaborate on in §2, for programmers, developing IoT applications is often tedious. They typically operate at a device level (*e.g.,* lamps, cameras) with limited support for composing these into higher-level abstractions (*e.g.,* home, building). As a result, developing an IoT application involves writing substantial ad-hoc glue code to compose these devices (*e.g.,* configuring *home* settings based on the events inferred from a camera) with limited support for rich policies and few higher-level abstractions.

Not surprisingly then, users also 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 [31, 44] over per-device state which can be hard to reuse, manage, or reason about as the number of devices and scenarios increase. In short, IoT lacks a systems architecture that would *simplify and accelerate* application development.

The well-established approach to simplifying development and improving developer efficiency is via abstraction and modularity [67] - allowing developers to repurpose modular building blocks and combine these into higher-level abstractions that shield downstream developers from lower-level details. This approach both improves developer efficiency and simplifies the task of providing users with high-level abstractions (*e.g.,* programming "the home" vs. individual devices). There's a rich literature on developing such abstractions for specific domains such as analytics [9, 56], AI [42, 51], and networking [64]. The contribution of these systems lies in identifying a minimal set of abstractions that are intuitive for the task at hand yet flexible enough to support a range of applications in that domain.

What is this minimal set of abstractions for smart space applications? In this paper, we propose two modular building blocks and three abstractions that serve to hierarchically

SOSP '21, October 26–28, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483559

compose these building blocks into higher level abstractions. The two building blocks are the **digivice**, which abstracts the devices to be actuated, and the **digidata** which abstracts how IoT data is processed. A digivice is *flexible* in that the entity it represents may be an individual device or aggregates of devices; a physical device (*e.g.,* lamp) or a virtual/abstract one (*e.g.,* "home"). Digivices are *declaratively* controlled – *i.e.,* users specify the desired state of a digivice without specifying *how* that state is achieved. These design choices are in contrast to existing IoT frameworks in which a device abstraction typically corresponds to a single physical device, couples both actuation and data processing for that device, and exposes imperative APIs to control that device.

Digivices and digidata – which we collectively refer to as "digis" – can be *composed* using three abstractions: (1) **mount(A, B)** allows higher-level digivice B to configure/actuate digivice A; (2) **pipe(A, B)** arranges for the output data from A to be consumed by B, and (3) **yield(A, B)**, for *A* mounted to *B*, yield allows *B* to relinquish control over *A*. Thus digis 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 digidata.

Low-level digis can be composed into higher-level abstractions that are controlled in a declarative manner. For example: a lamp digivice may be mounted to a room digivice and a camera digivice piped to a stream digidata which is also mounted to the same room digivice. The room digivice might then configure the lamp based on the objects recognized in the camera's stream; finally, under certain conditions, the room digivice might "yield" control over the camera to a third-party emergency control digivice.

As we'll discuss, dSpace faces unique challenges that stem from its role in integrating with the physical world. We address these challenges with certain novel design choices:

Adaptive composition. In dSpace the composition between digis is *dynamic and automatically adapted based on policies and real-world events*; *e.g.,* a roomba digivice that moves between rooms might automatically detach from one room digivice and get mounted to another.

Embedded policies. In dSpace, policies are expressed and enforced independently within each digi rather than as a standalone flat file of rules as is common today. This modularity makes it easier to correctly maintain and execute policies even as the digi-graph evolves.

Intent reconciliation: A software system that is implemented in a declarative style will typically accept a target intent and implement "state reconciliation" to move its current state to match this intent [54]. As we'll discuss, in dSpace, a system might need to go further and adapt its target *intent* and must do so in an autonomous manner based on policy and real-world conditions (*e.g.,* overriding the home's target "sleep" state when certain human activities are detected). To support this in a modular manner, digis implement "intent reconciliation" in which a digivice may update its *own* intent which is then recursively propagated to its upstream neighbors in the digi-graph.

A final challenge is that we want dSpace's implementation to match the modularity of its design. For this, we implement dSpace using a microservices based architecture. Specifically, we adapt the Kubernetes (k8s) design pattern of stateless controllers that coordinate only via a persistent data store. We show how to map our abstractions to k8s-style controllers and implement them in a manner that is compatible with, yet decoupled from k8s. Doing so allows us to reuse much of the k8s tooling while still providing features tailored to our smart-space domain.

We evaluate dSpace in the context of smart *home* applications.[1] We implement digis for 9 real-world home IoT devices from 9 different vendors, and 4 data processing frameworks. Using these, we implement 10 different deployment scenarios of mounting complexity, showing that dSpace allows developers to easily construct rather sophisticated use-cases. In addition, we show that these scenarios cannot be easily realized by existing smart-home frameworks such as SmartThings and Home Assistant: 40% of our scenarios cannot be supported by any of these other frameworks. For those scenarios that could be supported, doing so requires as much as 4x more lines of code/configuration than dSpace.

2 Design Goals and Rationale

2.1 Goals

Our aim is to design a framework that simplifies the development and use of smart space applications yet is flexible enough to enable a diverse range of these applications. To balance ease-of-development and flexibility, we want *(1) modular building blocks* that are *(2) controlled in a declarative manner*. This simplifies code reuse and evolution and reduces the surface area of the code that a developer must understand. In addition, we want to make it easy for developers to *(3) compose high-level abstractions and aggregates* – *e.g.,* a "room" controller that coordinates all the devices in a room; a "home" controller that coordinates all the rooms, and so forth. Such abstractions simplify development for downstream coders and ultimately simplify the user experience since it is now easier to build applications that expose higher-level controls to users (*e.g.,* putting an entire room into a low-energy mode).

Finally, we also want to simplify the *use* of smart-home apps. For this, our framework aims to simplify *(4) integration with AI/ML frameworks*. Doing so paves the way for policies that are automatically learned rather than manually written by users (§6). In addition, we aim to support *(5) delegation*

[1] We believe dSpace's modularity, flexibility, and rich policy structure are applicable to smart spaces more generally (*e.g.,* offices, residential, campus) but leave an exploration of this to future work.

of controls via which a user can flexibly outsource portions of home management to different third-party services; *e.g.,* outsourcing control over garden irrigation to a landscaping service, or yielding control over the home to a city-run emergency service under certain events. Delegation frees the home owner from the burden of home management but does so in a controlled and fine-grained manner.

2.2 Challenges

To some degree, the above are classic goals and our novelty lies only in tackling them in the context of smart spaces. This context raises certain unique challenges:

1) Composition must be adaptive, based on events and conditions in the physical world. Consider a home with "room" controllers that program the devices in that room; this set of devices changes with device mobility (*e.g.,* a roomba) yet must be automatically handled. Delegation also requires adaptive composition – *e.g.,* a room that is mounted to an emergency controller in the event of extreme heat.

2) Policies abound. A key challenge in smart spaces is that there is rarely a single "correct" action and instead user preferences as captured by policies determine the desired behavior of the system - *e.g.,* does the user want the heat turned on when entering a room? to what level? does she prefer appliances to be run at night or when the user is away from the home? And so on. Today's systems maintain such policies as a flat file of IFTT rules [31, 48] but this is hard to scale with increasing numbers of devices, difficult to maintain in customized deployments, and at odds with the information hiding that higher-level abstractions provide.

3) Intent specification is messy. In our declarative control paradigm, the dSpace application configures the target state or "intent" for each device based on user policies or other control logic. The challenge that arises in applying this paradigm to our context is that intents may also be determined by interactions in the physical world – *e.g.,* consider a scenario in which a user manually turns on a lamp in a home that is in sleep mode. Ideally, this action should cause the home to reconsider its intended state - *e.g.,* "waking" the relevant room, or the entire house, or perhaps raising an alarm. The exact action to be taken is a matter of policy - the challenge for us is to provide the architectural hooks that allow the developer to more easily express how conflicting intents (*e.g.,* from the virtual vs. physical world) should be resolved.

2.3 Design choices

We highlight the key design decisions that allow dSpace to meet the above goals and challenges.

(1) Separation of control and data. Our first design principle is to *decouple data and control processing into distinct abstractions*: digivices and digidata, as mentioned in §1. This is in contrast to several existing IoT systems (*e.g.,* Home-Assistant [29]) but is a deliberate choice that we made for two reasons. First, it allows us to adopt different programming paradigms for each: declarative models for control processing and dataflow models for data processing. Second, it allows us to easily leverage existing analytics and AI frameworks [9, 51, 56, 68, 71] rather than reinvent the wheel (see §3.1). I.e., developers of digivice control logic can leverage systems like Tensorflow or Spark while keeping their control logic cleanly separated and easier to evolve.

(2) First-class composition. Our second design principle is to embrace *composition as a first-class design primitive*, leading to the `mount` and `pipe` operators corresponding to digivices and digidata respectively. Such composition allows us to easily program aggregates of devices (*e.g.,* configuring all the devices in a room R by iterating through the devices mounted to R) and to construct digivices at a higher layer of abstraction (*e.g.,* instead of interacting with individual lamps, we can simply configure the brightness of the room). Composition also simplifies reuse since it allows the same building blocks to be composed in different ways to achieve different goals.

Another important design decision is how we *constrain* composition and the resultant digi-graph. A natural approach would be to compose digis into higher-level aggregates and a hierarchical control tree. This approach captures the natural organization commonly found in the physical world. E.g., consider a scenario where a campus or company headquarters wants to enforce occupancy limits. Each building/office in the campus may have their local policies that translate the campus-wide occupancy limit to per-floor or per-room limit based on which they may adjust the lighting, temperature, humidity, and smart locks settings on individual devices.

However, we do not limit ourselves to just a single static control hierarchy. Instead, a user/developer can define a *multi-rooted control hierarchy*. E.g., a lamp digivice might be mounted to a room controller (that controls devices in the presence of home occupants) and an energy-efficiency controller (that implements power savings when occupants are absent). As a different example, a digi-graph might include three services separately controlled by home owners, a landscaping service, and an emergency service. Note that allowing multiple control hierarchies 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, we enforce that multiple control hierarchies may simultaneously read the device states but only one control hierarchy is allowed to write/configure the device.

We achieve this with programmable "`yield`" and "`mount`" 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 adaptive composition, providing a level of flexibility that goes beyond traditional modular software frameworks that allow code reuse within a single developer/operator context.

(3) Embedded policies. dSpace implements embedded policies and intent reconciliation (§3.5) to meet our last two challenges. In dSpace, a policy is contained within a digi; *i.e.,* it is written as a part of that digi's definition and implementation. Their embedded nature ensures that policies are co-located (*i.e.,* "fate share") with their associated digi. As such, the scope of an embedded policy is immediately clear: it is determined by the position of its corresponding digi within the overall digi-graph. For example, two rooms – a bedroom and a kitchen – may have different policies for their desired light and sound levels and these policies must only be imposed on devices currently "mounted" to that room. Embedded policies (together with the digi-graph topology) make it easy to correctly identify devices and enforce policy even as devices move from one room to another.

Finally, we briefly discuss our approach to accommodating device heterogeneity since this has been the focus of much prior work. The common proposal is to address heterogeneity through standardization (*e.g.,* of configuration parameters for a particular class of devices). However, in practice, adoption of such standards has been slow and at times hampered innovation. Thus, in dSpace, we do not assume any standardized device models. At the same time, we liked the idea of shielding developers from the vagaries of vendor-specific APIs. dSpace achieves this through the notion of a *universal* digivice that exposes a "universal" set of configuration parameters and that includes the code to translate from these universal parameters to a vendor-specific digivice. E.g., consider a universal lamp U that exposes a standardized set of parameters u and contains the logic to translate u to the parameters l of a vendor-specific lamp L. In dSpace, we mount L to U so that any higher-level digivice that U is mounted to is shielded from the details of L. Note that there is no magic here and we are not claiming to have solved the challenge of heterogeneous APIs: someone still has to write U and deal with the idiosyncrasies of L's implementation. Instead, our only claim is that we're providing a framework that makes it easy to systematically *reuse* U and the effort that went into developing it. Moreover, this approach is flexible: developers can choose when and which universal devices to use on a per-device basis.

2.4 Putting the pieces together

To build an end-to-end smart space application, a developer and/or user selects the desired digivices and digidata, composes them into her desired hierarchy, and then "programs the space" 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 the hierarchy. Similarly, events from the physical world travel up the hierarchy. Intents and events are processed at each digi in accordance with its embedded policy and this processing may trigger additional control actions and/or intent reconciliation.

Developers follow the above process when developing a dSpace application, while users (*i.e.,* home owners and occupants) do so when setting up and using the application. Users interact with dSpace by configuring high-level intents and writing/customizing policies. We envision that interaction will be done via a user-friendly user interface (UI) and user experience (UX) design, though the design of this UI/UX is beyond the scope of this paper.

3 Design

3.1 Abstractions

A smart space consists of physical and abstract entities. Using dSpace, developers create digital abstractions ("digis") to control these entities and process the data they generate. Specifically, we introduce two core abstractions: digivice and digidata, which we depict as shown in Fig.1a.[2]

Digivice: A digivice enables declarative control over physical devices and other digivices. Each digivice (D) contains four key components:

(i) A *model* ($D.mod$): this is a set of attribute-value pairs that capture the intended state or the *intent* for the digivice, its current state or *status*, events generated, and other relevant information – the list of attributes can be found in Table 1 and sample digivice models in Fig.1b and Fig.1d.

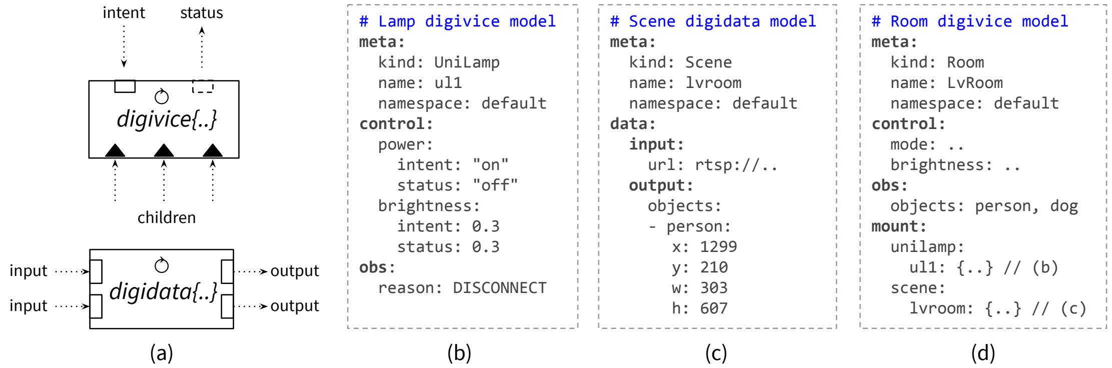
(ii) A list of the digivice's children ($D.ch$) in the digi-graph.

(iii) A driver, $D.drv()$: this is the code whose main function is to *reconcile* the digivice's intended state ($D.mod.i$) with its current state ($D.mod.c$). I.e., based on current events and conditions, as well as policies (described below), the driver takes action(s) to move the digivice's current state towards its intended one. Actions might involve directly interfacing with and controlling physical devices, programming the intents of its children in the digi-graph, generating events *etc.*.

(iv) Policies ($D.pol$): these are rules that can be written by the developer or the user that guide how the dSpace driver responds to physical events. Policies are key to programming/-customizing an application's behavior to specific scenarios and user preferences.

Digidata: A digidata enables data processing to be integrated natively with digivices. A digidata follows the same design pattern as a digivice with a few differences/extensions: (i) the digidata's model ($T.mod$) includes data input ($T.mod.in$) and data output ($T.mod.out$) attributes, (ii) the digidata's driver ($T.drv()$) includes data processing code that implements the data transformation from the input data to the output data as specified by the data input and output attributes. It can also be a thin wrapper around a standalone

[2] Our visual representation of digivice/data is inspired by that of the Click [64] authors.



```
# Lamp digivice model
meta:
  kind: UniLamp
  name: ul1
  namespace: default
control:
  power:
    intent: "on"
    status: "off"
  brightness:
    intent: 0.3
    status: 0.3
obs:
  reason: DISCONNECT
```

```
# Scene digidata model
meta:
  kind: Scene
  name: lvroom
  namespace: default
data:
  input:
    url: rtsp://..
  output:
    objects:
    - person:
        x: 1299
        y: 210
        w: 303
        h: 607
```

```
# Room digivice model
meta:
  kind: Room
  name: LvRoom
  namespace: default
control:
  mode: ..
  brightness: ..
obs:
  objects: person, dog
mount:
  unilamp:
    ul1: {..} // (b)
  scene:
    lvroom: {..} // (c)
```

Figure 1. a. Conceptual models of digivice (top) and digidata (bottom); b. a Lamp digivice model; c. a Scene digidata model; d. a Room digivice model that mounts/controls a Lamp as its child and pipes/reads data from a Scene.

Abstraction	Notation		Description	Attribute
Digivice	D.mod	.i	Digivice D's intended states	intent
		.c	D's current states	status
		.e	Events generated and observed by D	obs
	D.ch		D's children on digi-graph	mount
	D.drv()		D's driver is a function *func*(D.mod.*, D.pol, c.mod.*, physical world events) → (D.mod.*, c.mod.i, action on physical devices), $\forall c \in D.ch$	n/a
	D.pol		Rules specified by the developer or user that update D.mod.* based on D.mod.* and c.mod.*	reflex
Digidata	T.mod	.in	Digidata T's data input	input
		.out	T's output data output	output
	T.drv()		Data processing code that implements the transformation between input and output	n/a
mount	mount(A, B)		Allows B to write to A.mod.i (or .in if A is digidata) and read from A.mod.c (or .out) and A.mod.e	n/a
pipe	pipe(A, B)		Writes A.mod.out to B.mod.in	n/a
yield	yield(A, B)		Revokes B's write access to A.mod.i (read access is unchanged)	n/a

Table 1. Abstractions in dSpace, their notations, and attributes in our implementation of a digi's model.

data processing system such as FFmpeg [16] and Tensorflow [51]. An example "Scene" digidata is shown in Fig.1c. This digidata takes a video stream (from `input.URL`), implements object recognition, and updates detected objects (to `output.Objects`).

We refer to digivice and digidata collectively as digis. Attributes in a digi's model follow a predefined *schema*. The model is accessible via verbs, a predefined set of APIs to access attribute-value pairs using their URIs, *e.g.,* `get(URI)` and `update(URI, new_value)`.

3.2 Composition

Digivices and digidata can be composed to form a "digi-graph". Composition takes place when users call the following *composition verbs* on digis:

mount: Given a *digi* A and *digivice* B, $mount(A, B)$ allows B to control A. Note that A might be a digivice or digidata, while B must be a digivice. Specifically, B can write to $A.mod.i$ and read from $A.mod.c$, and likewise for digidata. We refer to B as the parent digivice and A as the child. Mount can be used to implement different semantics:"B represents A," *e.g.,* a universal lamp digivice represents a vendor-specific one; and "B aggregates A," *e.g.,* a room digivice includes other lamp digivices. A digivice can mount multiple children and can also have multiple parents, subject to the mount rule that we describe in §3.3. Mount can also happen at any level (*e.g.,* A mounted to B and B mounted to another digivice C), forming a hierarchy. Further, each mount has a *mode* that can take the value "expose" or "hide". The latter prevents the parent digivice from accessing the child digivice's children while the former allows it to do so.

pipe: given two digidata A and B, pipe(A, B) writes A's output to B's input; *i.e.,* updates to $A.mod.out$ are written to $B.mod.in$. Note that if $A.mod.out$ is a pointer to data (*e.g.,* a URL to a video stream), only the pointer gets written to $B.in$. Each digidata can pipe to multiple digidata; *i.e.,* write its data output attributes to their input attributes. However, at most one digidata can pipe to an input attribute; *i.e.,* we enforce a single writer per port. The URIs to the input and output attributes in A and B must be specified when calling pipe.

One can use thus the pipe verb to compose digidata to form "data flows". As mentioned earlier, a digivice can mount a digidata and thus update its data input attributes and read from its data output attributes.

yield: composition via the mount verb can lead to a multi-rooted hierarchy. We rely on the yield verb and mount rule (§3.3) to enforce sensible write and mount semantics (§3.3).

Specifically, calling the yield verb, *e.g.,* $yield(A, B)$ means that digivice B's driver no long has write access over A. However, the yielded parent digivice may continue to watch

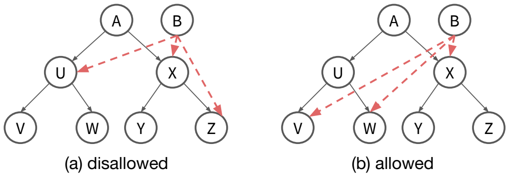


Figure 2. Cases where mount is disallowed or allowed.

for updates to the child's model and act accordingly. Note that one can use the mount, pipe, and yield verbs to unmount, remove pipe, and unyield respectively by simply setting the corresponding flag at the verb call.

Information about composition is maintained as *composition references*, where each composition reference tracks the relationship between a pair of digis. Fig.1d includes examples for *mount reference/attribute*, *e.g.,* `mount.unilamp.ul1`. Developers define mount references in the digivice's schema to specify which digivice/data kinds are compatible and hence can be mounted to this digivice. We'll describe how the composition references are implemented/used at runtime in §5.

3.3 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 form a hierarchy as shown in Fig.2a. We want to mount some of A, U - Z to B (imagine adding digivices in a home hierarchy to a new power controller). Clearly we want to avoid loops since they can lead to erroneous or unpredictable behavior as intents and status are passed around in a cycle. In addition, note that in Fig.2a B can write the intent of both X and Z where Z is already a child of X. This can lead to *intent conflicts* where Z's intent can be set by both B and X.

Our goal is to avoid mount loops and intent conflicts while still allowing a multi-rooted hierarchy in some constrained form. To this end, we require that the digi hierarchy formed by calling mount/yield must be a multitree (more formally, a *diamond-free poset* [61]) and we meet this requirement by enforcing the following rule when mount is called:

Mount rule: *A digivice cannot join a hierarchy that it or any of its descendants is already a part of.*

An example of a hierarchy that is allowed is shown in Fig.2b. However, the mount rule alone still does not solve the multi-hierarchy problem. As long as a digivice has multiple parents, there can be *intent conflicts* among the parents. Our solution to this is simple: for each digivice, we allow only a single writer at a time and who gets to be the writer is determined by the *yield policy* as described next.

3.4 Adaptive Composition

The multi-hierarchy enables *shared* and *delegated* control over digivices. In smart space contexts, such sharing often needs to happen dynamically, adapting to real-world events, *e.g.,* when the fire alarm is on, a home digivice should hand over its control to an emergency-control digivice; or when a roomba robot moves between rooms, its digivice might be unmounted from the previous room and mounted to the new one. We often want such *adaptive composition* to be driven by user-defined policies without a human in the loop.

dSpace achieves adaptive composition by allowing users to define composition policies that invoke yield and mount when a specified condition is met. For example, a yield policy may specify the condition under which control over a shared digivice is transferred from one digivice to another. An example yield policy can be found in the Appendix B.3. Mount policies work in a similar fashion allowing digivices to be mounted/unmounted based on predefined conditions.[3] Yield allows us to enforce a single-writer policy as follows: when $mount(A, B)$ is called, if A has no other active parent (where active means the parent hasn't yielded the child) then the mount call completes and B can write to A. However, if A already has an active parent, then the mount is automatically followed by a yield that ensures B cannot write to A.

3.5 Intent Reconciliation

In dSpace, developers and users "embed" automation logic and policies within a digivice. As described so far, mount and yield policies ensure that at any point in time, a digi's intent can only be modified by one other digivice; *i.e.,* we're enforcing single writer semantics on a digi's intent. However, intent conflicts can still result from events in the *physical* world. E.g., consider a lamp whose digivice is currently mounted to a room digivice and the room has set the lamp's intended state to be "on" and now a user physically turns off the lamp. Should the lamp stay on (as dictated by the room) or off (as dictated by the user's action)? The answer in this case may seem clear - the user's action should be respected and the lamp turned off. However, the answer might be less clear in a scenario that (say) involves disabling a home alarm or unlocking a door. Such scenarios lead to intent conflicts though, unlike above, they involve a conflict between a digivice's parent in the digi-graph and the physical world (versus a conflict between a digi's multiple parents).

dSpace addresses this by allowing developers or users to define intent reconciliation policies in the digivices. These policies are executed whenever an intent conflict occurs and provides conflict resolution. For instance, when the user physically turns off the lamp, the room may react by resetting the lamp to be "on", overwriting the user's intent; or it may adjust its own intent to be compatible with these events. Note that the action taken to resolve an intent conflict is a matter of policy and hence application/user dependent. What dSpace provides is the framework for defining such policies and the runtime guarantees that ensure intent reconciliation is executed correctly. For correct execution, dSpace introduces a version number in the digivice model which is

[3] Although not currently implemented, one might extend adaptive composition to data flow composition with, *e.g.,* pipe policies.

incremented when the model is updated. We rely on opportunistic concurrency control based on this version number to achieve serializability on model updates. The dSpace runtime guarantees that if a writer sees updates to a model with two version numbers V_a and V_b (with $V_a < V_b$), then it must have also seen all updates with version number between the two if any. This ensures that an active parent of D will not miss any intent updates on D. We describe how dSpace's implementation handles concurrent intent updates in §5.2.

3.6 Security and Privacy

Security and privacy are important design aspects for IoT systems and dSpace is no exception. There is an extensive literature on IoT security and privacy that, broadly interpreted, falls under two main categories: understanding user awareness and requirements [77, 78, 80] and providing system support [55, 69, 75, 76, 79] for security and privacy.

With regard to system support, dSpace embraces best practice via the following techniques: *(1) Role-based Access Control (RBAC).* In dSpace, each digi driver is associated with a role that constrains the driver's access to its own model. dSpace controllers (§5.2) are associated with roles that are granted the corresponding access for them to enforce composition semantics (*e.g.,* the mounter is assigned write-access to the parent and its children). Users and third-party digis are assigned access by the admin-user following standard practice for managing roles/permissions. *(2) Isolation.* As we will cover in §5, digis are run in separate application containers which provide OS-level isolation including separate namespaces, network stacks, and performance isolation via cgroups [19]. This helps prevent buggy digis from sabotaging other digis when they are running on the same node. *(3) Admission control and general Kubernetes protections.* Unlike the common single-node/monolithic architecture in existing IoT frameworks, dSpace builds on top of Kubernetes and allows digis to run in different machines in a distributed fashion. As such, security-critical digis can be run on their dedicated nodes separated from the others. Further, all entities are authenticated with standard tools (*e.g.,* client certificates) when the Kubernetes apiserver receives an API request from them. Each API request undergoes integrity checks by the apiserver to filter malformed requests. dSpace thus provides defense in depth via these layers of system-level protections.

Next, we discuss the aspects where we believe dSpace's design can be helpful on the end-user/UX front. Prior studies have reported that the gap between the user's mental model and the real device setup (*e.g.,* the device topology, how devices interoperate) can introduce security and privacy risks [77, 80]. We speculate that dSpace makes it easier to bridge the gap because (i) the dSpace hierarchy reflects natural organization in the real-world (*e.g.,* lamp mounted to room, then room to home); (ii) dSpace enables adaptation that reflects real-world events (*e.g.,* intent reconciliation given human inputs, dynamic composition); (iii) our use of embedded policies means that policies are co-located with their associated objects and operations rather than exposed to users (or UX designers) as unstructured flat files; and further (iv) composition must be explicit and is a prerequisite to imposing policy - *i.e.,* existence of a policy is not enough (unlike today's frameworks). As such, dSpace raises a user's awareness to security and privacy issues, including who has access to what data and devices.

Together, we believe these design choices make it easier for developers and users to understand the security of their solutions. However, proving this conclusively requires a UX design and user study which is beyond the scope of this paper. In summary, we believe that security and privacy in smart spaces requires the right combination of system design and UX design. dSpace addresses the systems side but a complete solution including a UX design is a topic for future work.

4 Programming and Execution Model

In this section, we present the programming and execution model of dSpace. Developers create a digi by specifying its model schema and programming the driver. The driver consists of event handlers that are invoked in response to model updates, executing the digi's embedded policies. dSpace simplifies digi development via a driver programming library, described in this section, and dSpace controllers (§5).

4.1 Model Specification

To create a new digi, one starts by designing its model schema. Below (left) shows the schema of a simple Plug digivice where we specify the digi's identifiers - its group, version, and kind - as well as a control attribute "power" and string as its data type. Table 1 summarizes the list of attributes that one can add to a digi.[4]

```
1  group: digi.dev
2  version: v1
3  kind: Plug
4  control:
5    power: string
```

```
1  import digi, pytuya
2  plug = pytuya.Plug("device_id")
3  @digi.on.control
4  def handle(power):
5      plug.set(power["intent"])
```

4.2 Driver Programming

In essence, dSpace supports standard event-driven programming. Developers program a set of handlers as part of the driver logic that gets executed when models are updated. An example in Python is shown in the code block above (right). It begins by importing the digi driver library and any device libraries (here, `pytuya` for programming Tuya devices [46]) if needed. In this example, a handler **handle(power)** is defined with a decorator **@digi.on.control** (line 4) specifying that this handler will be invoked when changes occur on the control attributes; and when invoked it sets the plug to the power's intent value.

[4] To avoid confusion on the terms: we refer to a digi that has control attribute(s) as a "digivice" and one that has data attribute(s) as a "digidata". No digi can have both control and data attributes (§3).

```
reflex:
motion-brightness:
  policy: >-
    if $time - .motion.obs.last_triggered_time <= 600
    then .control.brightness.intent = 1 else . end
  priority: 1
  processor: jq
```

Figure 3. Example on-model policy with reflex.

We now explain how driver programming works in dSpace. First, recall that in dSpace a digi driver can only access its own model. Programming a digi driver is thus conceptually very simple: writing policies/logic (§3.5) that manipulate the digi's model (essentially processing a JSON document) plus any other actions to affect the external world (*e.g.,* send a signal to a physical plug, invoke a web API, send a text message to your phone etc.). dSpace's runtime handles all other tasks such as syncing states between digis' models when they are composed (§5).

dSpace's driver library provides a few useful programming primitives. They are, briefly, (i) **Filters** are specified via the **digi.on** decorator that decides when a handler should be called. One can supply parameters to the filters, *e.g.,* adding a path to focus further down on an attribute's sub-tree ; as well as the priority of the handler. (ii) **Views** provide convenient transformations of the model, *e.g.,* rearranging the attributes in the model such that they are more easily accessible. Views can be chained. Updates to a view will be automatically applied to the source view it is transformed from. We include code examples of digi drivers in the Appendix B and codebase [12].

Configure Policies with Reflex: Users or operators can define new behaviors or augment a digi's existing behavior by specifying on-model policies (§3). This is done via the reflex API in dSpace. Fig.3 shows an example where a "motion-brightness" (the name) reflex is defined to configure the brightness of a lamp if any motion was detected in the past 10 minutes. This logic is specified in the **policy** field and will be executed by jq [36] defined in the processor field. One can specify the priority of a reflex in the **priority** field following the same rule as the handlers in the driver (default to 0; and negative value disables). Further, one can reconfigure handlers in the driver by specifying a reflex with the handler's name. Under the hood, a reflex is executed as just another handler in the driver, as described next.

4.3 Driver Execution

As depicted in Fig.4, the driver starts when **digi.run()** is called. It creates a *reconciler*, registers handlers with the reconciler, and starts watching for changes in the model. The reconciler is a single process that executes the handlers in order during a reconciliation cycle. When handlers are registered, they are sorted by their priority values. High priority handlers will be executed later than the low priority ones. If the list of handlers are updated (*i.e.,* via the reflex API), the order will be updated accordingly. A handler will be executed only if its model changes satisfy the condition specified in the filter. A reconciliation cycle is triggered whenever there is a new update to the model, unless the update is caused by the previous reconciliation. Once all handlers are executed, the reconciler writes the updated model to the apiserver.

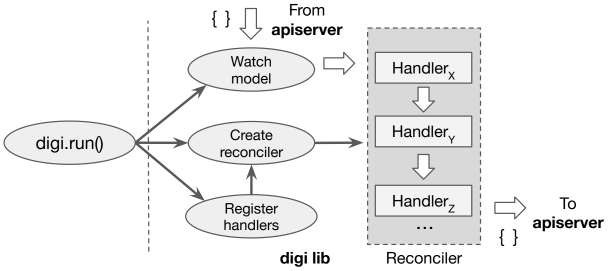


Figure 4. The digi driver subscribes to changes of the model from the apiserver and runs the reconciliation cycle, invoking handlers from priority low to high. Each handler may modify the model and at the end of a reconciliation cycle the (new) model is posted to the apiserver.

5 Runtime Architecture

The architecture of dSpace comprises (1) an application layer running digis in separate containers (Pods in Kubernetes) and (2) a system layer with an apiserver hosting models and the dSpace controllers that provide runtime support for digi composition, policy, and run-time guarantees. In this section, we will focus on the runtime components in the latter.

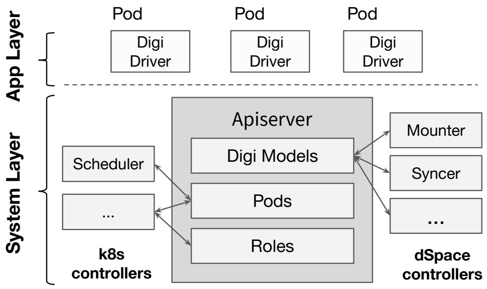


Figure 5. dSpace's architecture has two parts: (i) an application layer that runs application digis as standard Kubernetes Pods and (ii) a system layer that runs apiserver, k8s controllers and dSpace controllers.

5.1 Apiserver

The apiserver, as its name suggests, stores data as API objects (*e.g.,* digi models) and exposes them via standard REST APIs. We reuse the k8s apiserver [10] as we find it a natural choice for hosting attribute-value pairs and allows them to be accessible over REST APIs. The apiserver stores the attribute-value pairs in a persistent key-value store [24]. It exposes an asynchronous Watch API that allows one to subscribe to changes in a model, in addition to standard CRUD operations. Following Kubernetes's convention, we refer to these APIs as verbs.

Before serving and/or executing the verb requests, the apiserver runs a series of checks on whether the verb is valid, including correctness of syntax and semantics, sufficient access rights from the caller, whether the proposed changes violate composition rules etc. The last check is performed by the dSpace's admission webhook (as part of the mounter dSpace controller) which we will describe next.

5.2 dSpace Controllers

Kubernetes provides controllers such as the scheduler, deployment controller, and autoscaler to support container orchestration. While these controllers are sufficient for deploying the digis, there are no existing controllers or mechanisms that provide the support for composition and policy dSpace aims to offer. Hence, we implemented a collection of *dSpace controllers* to support composition and policy.

Mounter When digi A is mounted to B, the mount controller (or mounter for short) synchronizes the states between their models at runtime, *i.e.,* between the model of A (M_A) and model of B (M_B). This is done by, briefly: (i) when the mounter first sees a mount reference to A appear on the M_B, it copies M_A under the mount attribute of M_B. We refer to the copied-over M_A as a *model replica.* It is stored as part of M_B under A's mount reference. (ii) When M_A is changed, the mounter syncs the updates to the model replica in M_B; (iii) when the model replica is changed, the mounter syncs the updates to M_A. Note that the mounter will not sync any updates on .status fields in the control attributes from the model replica to M_A since the status information should never flow southbound. It will, however, sync .intent updates from M_A to the model replica to allow intent reconciliation.

Further, in dSpace, each model contains a *version number*[5] that is incremented whenever the model is updated. The version number is also copied over to the model replica. dSpace ensures that the states in M_A won't get overwritten by any outdated states from its parent/M_B. To do so, the mounter compares the version number of M_A to that of M_A's model replica (in M_B) and syncs the states of the replica only when the replica's version number is no less than the version number of M_A. Finally, the mounter implements the rest of mount semantics, yield and mount modes (hide/expose), by syncing the states accordingly.

Syncer The sync controller tracks two given models, a source and a target, and syncs states between the two models. The sync information is tracked in a *Sync* API object stored on the apiserver. Syncer implements the data-flow composition with pipe. Whenever a user calls `dq pipe A.output.x B.input.x`, `dq` creates a Sync object using A.output.x as the source and B.input.x as the target.

Policer Akin to the Syncer, the policy controller watches all Policy objects (*e.g.,* mount and yield policy) where each Policy object consists of the policy statement and digis involved in the policy. The policer starts watching for changes on these digis and enforces the policy if any of the conditions are triggered. Note that the Mounter, Syncer, and Policer subscribe to model changes via the Kubernetes's Watch API [39], without constantly polling for updates.

Topology webhook An admission webhook [20] is a mechanism in Kubernetes to extend the apiserver's request admission process. When the apiserver receives a request, it will forward the request to a registered webhook and the webhook can decide whether to accept or reject the request. dSpace leverages this mechanism and implements a *topology webhook* to enforce the multi-hierarchy and single-writer constraints (§3). Topology webhook tracks the latest status of the digi-graph and rejects any invalid changes (*e.g.,* an invalid mount/pipe request) that lead to an invalid digi-graph.

5.3 Implementation

Our implementation comprises ≈10.4K lines of code (LoC), 57% in Go for the runtime, 24% in Go for the `dq` command line (including code generators and digi image support), and 19% in Python for the driver library (built on top of `kopf`, an open-source k8s operator framework [33]). All dSpace controllers and web hooks use standard APIs to interact with Kubernetes control plane. All digis, dSpace controllers, and policies can be created and/or composed declaratively via standard Kubernetes configuration (yaml), using its command line `kubectl`, or `dq`, which provides complementary commands/shortcuts such as run, mount, yield, pipe, build, alias, push, pull *etc.* to simplify run-time operations and avoid configuration file sprawl [30]. Currently we provide only a Python front-end for driver programming. We expect both higher-level UI/UX support and other driver language support for in future.

For system security, we reuse k8s's access control mechanisms (Service Accounts and its RBAC module [47]). An implication of our k8s-based implementation is that we inherit many of its desirable features with respect to fault handling (*e.g.,* automatic pod restart), availability, persistence, application delivery, CI/CD (digi image), and configuration management (kustomize, kubectl). Finally, between leaf digis and the physical devices, we preserve vendor-specific security mechanisms such as device-level authentication and encryption. dSpace is open source and additional detail about our implementation can be found in our code repository.[6]

6 Evaluation

Our evaluation focuses on answering two main questions: (a) does dSpace meet our goal of enabling and simplifying the development of advanced smart-space deployments? and (b) is our k8s-style implementation flexible and performant? We evaluate these questions using a combination of four approaches: (i) we use dSpace to implement a range of deployment scenarios using real-world home IoT devices and

[5] Note that this version number is different from the schema version (§4.1).

[6] https://github.com/digi-project/dspace

Device Type	Vendor	Model	Library	Access
Light bulb (L1)	GEENI	LUX800	Tuyapi	LAN
Light bulb (L2)	LIFX	Mini	Lifxlan	LAN
Light bulb (L3)	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. (BS: basestation; VC: vendor cloud)

Digidata	Data Attributes	Language/Tools
Scene	in: url; out: json	OpenCV [53], Tensorflow [51]
Xcdr	in: url; out: url	FFmpeg [16]
Stats	in: json; out: json	PySpark [9]
Imitate	in: json; out: json	Ray [42]

Table 3. Digidata in our evaluation.

report on our experience doing so in §6.2; (ii) we report on our efforts to implement these same scenarios using 2 popular IoT frameworks in §6.3; (iii) we conduct a small-scale programming assignment using 8 developers with no prior exposure to dSpace in §6.4; and finally (iv) we benchmark the performance of our prototype testbed under different deployment scenarios in §6.5. Before presenting our results, we elaborate on our experimental setup, the IoT devices we use, and the deployment scenarios we evaluate in §6.1.

6.1 Experimental Setup

Devices and Testbed. Table 2 describes the physical devices used in our experiments. 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. Similarly, Table 3 lists the data processing frameworks that we encapsulated as digidata.

The digis corresponding to the above devices and frameworks were built in 1,667 LoC; they interact with physical devices via device libraries (*e.g.,* LifxLamp digivices uses lifxlan [41]). We assume that these leaf digis are already available when a developer wants to implement the scenarios. We expect such software reuse to happen in reality just as how people reuse device drivers. We thus focus on the programming efforts required to construct the higher-level digis that fulfill these scenarios.

We run these devices in our homes; by default, the Kubernetes (version 1.18.2), dSpace runtime, and digis run on a Lenovo Thinkcentre M720 (Intel Core i5-8400T, 6 cores).

Scenarios. As described below, we consider a series of scenarios that incrementally build up in complexity. In what follows, we refer to "users" that configure/set up the smart home 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".

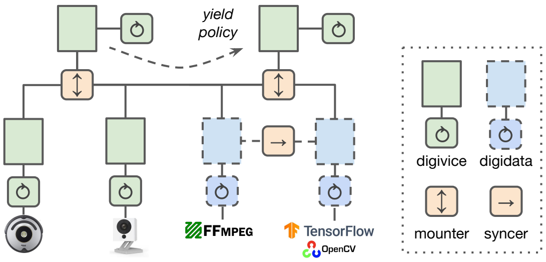


Figure 6. The digi-graph capturing scenario S5, 9, 10.

Scenario 1 (S1): *Unified control over lamps in a room.* In this simplest scenario, we have 2 lamps (L1 and L2) in a room and rather than control each lamp individually, the user wants to simply specify a "brightness level" for the room. Later, the user wants to add a lamp L3 to the room while retaining the same brightness level (which means the room should automatically lower the power/intensity of L1 and L2 to reflect L3's addition). In addition, L3 has configurable colors and the user should be able to set the room's ambiance color as a new configuration option. The lamps come from different vendors each with different APIs; *e.g.,* `Geeni` [25] and `Lifx` [35] lamps have different luminous intensity and color schemes.

Scenario 2 (S2): *Reconciling intents specified in the physical vs. virtual world.* Building on S1, the user may interact with their lamp in different ways: manually dimming it via a physical switch or updating the room's brightness and any of these operations may occur concurrently with other actions in progress. In all of these cases, the room should be able to reconcile its intent (*i.e.,* target brightness) to these user actions and updates from the lamps.

(S3): *Motion-triggered configuration.* Next, the user adds a motion sensor (the Ring Alarm Motion Detector [6]) to the room and would like the room to automatically configure its brightness based on the motion sensor.

(S4): *Multi-level abstractions.* Next, in addition to configuring at the 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.

(S5): *Robot vacuum by scene.* The user now adds a camera and the Roomba [43] vacuum cleaner to the room and would like the camera's video stream to be processed by an ML system (we use TensorFlow) and the robot to be controlled by the output of this ML system; specifically, the robot will start/pause/dock based on whether a human is in the room.

(S6): *Learned 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, enabling a seamless user experience.

(S7): *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.

(S8): *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.

(S9): *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.

(S10): *Delegation of control.* Our user now wants to "yield" control over the home to a city-run emergency service in the event of an emergency.

6.2 Using dSpace to implement S1-S10

We briefly summarize how we implement S1-S10 with dSpace and then quantify the development effort involved.

(S1): *Unified control over lamps in a room.* This simple scenario exposes three challenges: (i) dealing with heterogeneous device APIs, (ii) enabling configuration at a higher layer of abstraction and (iii) dynamic composition and reconciliation. We address these with a digi-graph in which L1 is mounted to a universal lamp UL1 (L1 → UL1) and likewise, L2 → UL2. We then mount UL1 → R and UL2 → R, where R is a room digivice (and note that R does not have to deal with the heterogeneity in L1 and L2). Later, we add L3 → R; in this case, we omit the universal lamp since L3's color features are not present in the universal lamp model (highlighting the fine-grained control that dSpace provides over whether/when to adopt standardized models). R's driver automatically updates L1 and L2's settings when L3 is introduced based on the target room brightness level and without requiring any user action.

(S2): *Reconciling intents.* This scenario captures the need for intent reconciliation; in terms of a solution, it requires no change to S1, other than the correct intent reconciliation logic in both lamp and room digivices: the lamp allows an input from the physical world (*i.e.,* manually toggled switch) to override the intent specified from the room and, once notified of this change, the room digivice will accept the lamp's new intent (*i.e.,* target intensity) and correspondingly adjust the intents of the other lamps to maintain the same aggregate brightness.

(S3): *Motion-triggered configuration.* This is implemented as an on-model policy/reflex as shown in Fig.3.

(S4): *Multi-level abstraction.* This scenario demonstrates the flexibility of implementing multi-level hierarchical composition and control. We implemented this by writing a new home digivice and the room digivices are mounted to the home. The home's model supports predefined modes; based on the mode value the home's driver configures the brightness level of the downstream room; *e.g.,* "sleep" mode sets the room brightness to 0.

(S5): *Robot vacuum by scene.* This scenario highlights the power of rich compositions. To implement this, we pipe the output of the Camera digivice first to the Xcdr digidata for transcoding the video stream; then from the Xcdr to the Scene digidata - the latter fetches the video stream from its specified URL, performs object recognition, and posts objects in its .obs attribute. We mount the Scene and Roomba digis to the Room digivice which reads the objects from the Scene's output. Whenever the Room sees humans in the objects, it will pause the Roomba.

(S6): *Learned automation* To learn user preferences, we implemented an Imitate digidata that uses Ray's RLLib [42] and implements a behavior cloning algorithm that learns and applies a simple policy [63] of updating the home's mode based on the rooms' occupancy (obtained as in S5). Specifically, the digidata is mounted to the Home, which writes the list of objects in each room and the Home's mode to the digidata's input attributes. The digidata reads from its input attribute, learns a policy, infers what the next mode should be, and writes the mode to its output attribute.

(S7): *Service handover* We implemented a RoamSpeaker digivice that can mount the Room digivices and the Speakers digivices are mounted to the Room under "expose" mode (§3.2). RoamSpeaker accesses the source_url, volume, and mode of the Speakers that are mounted to the Room and sets the mode of the Speaker (pause or resume) based on the Room's occupancy; the latter is obtained as in S5.

(S8): *Device mobility* This scenario highlights dynamic composition that happens: (i) at runtime, (ii) without user intervention but maybe driven by user-define policies 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). We achieve this via a mount policy that specifies a Roomba should be unmounted from the Room when it is not listed in the Room's objects list (provided by the Scene digidata).

(S9): *Shared control.* dSpace enables S9 by allowing multiple control hierarchies and we do not program additional digis.

(S10): *Delegation of control.* We define a yield policy for S10 (it can be found in Appendix B.3) between the Room and an Emergency controller that it is mounted to.

Complexity of implementation: Table 4 summarizes the effort involved in implementing the above using dSpace. We see that most scenarios require only a small amount of code or configuration to implement and, in total, these changes add up to only a 15% increase of the codebase given the leaf digis. Among the higher-level (HL) digis, the Home (51) and the universal lamp (Unilamp; 43 LoC, shown together with the Room) require the most code changes, which is a

Feature:	HL abstraction and policies				Data-driven policies		Access policies			
Scenario	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
Leaf/Child digis	GeeniLamp, PhueLamp, LifxLamp	Digis in S1	Digis in S1, MotionSensor	Room	Roomba, Cam, Xcdr , Scene	Imitate, Stats	Room, Speaker	All	All	All
HL digis	Unilamp, Room	Room	Room	Home	Room	Imitate, Stats	RoamSpeaker	All	All	All
LoC (%)	5.0% (84)	1.2% (21)	0.06% (1)	2.9% (51)	0.9% (16)	2.7% (50)	1.9% (35)	-	-	-
LoCF (#)	4	0	2	1	0	4	2	10	10	12

Table 4. Implementing smart space scenarios in dSpace. Lines of Code (LoC) takes into account any changes in the driver code and the model definition (schema). Lines of config (LoCF) takes into account the lines of configurations (in yaml) that end-users needed to change The percentages of LoC in the table was calculated incrementally (taking into account the previous scenarios in the codebase). In total, 15% more LoC (258) were added to the leaf digis codebase (1,667 LoC) to implement these scenarios, as higher-level (HL) digis and policies.

Feature:	HL abstraction & policies				Data-driven policies	Access policies	
Scenario:	S1	S2	S3	S4	S5, S6	S7	S8, S9, S10
EdgeX	-	×	-	×	-	×	×
HomeOS	-	×	-	×	-	✓	×
AWS IoT	-	×	-	-	✓	×	×
HASS	✓	×	-	-	-	✓	×
ST	-	×	-	-	-	✓	×
dSpace	✓	✓	✓	✓	✓	✓	✓

Table 5. Scenarios that can or difficult to use prior systems to implement: '✓' easy to implement. 'x' does not support. '-': partial support with missing/difficult to implement features. HASS: Home Assistant; ST: SmartThings.

result of application-specific logic/configuration rather than framework-specific plumbing. E.g., the Unilamp digivice defines setpoint conversions for each vendor-specific digi lamps (see Appendix B.2).

6.3 Implementing S1-S10 in other frameworks

We were curious to understand how existing IoT frameworks fare in supporting the above scenarios. We thus selected 5 IoT frameworks including some of the most popular ones (SmartThings, Home Assistant), cloud-based services (AWS IoT), a leading open-source project (EdgeX), and a well-known research project (HomeOS). We examined their codebase (when available), API documentation, and code examples. For each framework and each scenario, we determine whether the scenario can be supported: entirely, partially, or not at all. Table 5 summarizes our findings, grouping the scenarios based on the key capability that they highlight. We present a detailed discussion of each category in the Appendix C and here only highlight a few conclusions.

For the HL abstractions and policies, all frameworks have some (partial) support for S1, S3 and S4 (except EdgeX and HomeOS), but no or limited support for S2. For instance, Home Assistant allows grouping devices such that one can access and actuate all devices in the group. However, a group's devices have to be of the same type and only a few predefined device types are supported (*e.g.,* the "Light Group" API). Though Home Assistant also provides a generic "Group" API that allows grouping devices of different types, one can only actuate the group with predefined "Service Calls" `turn_on` and `turn_off`. SmartThings and AWS IoT have similar limitations in this regard. To dig deeper, we made a best attempt at implementing three scenarios – S1, S3, and S4 – in Home Assistant (HASS) since it is open source.

For S1, we are able to implement it with Home Assistant though with substantially more code (240 vs. 84 with dSpace) which is primarily due to workarounds for their Group APIs. Since Home Assistant's built-in Group APIs cannot allow the composition we want in S1, we circumvent it by building a room "service" that interacts with multiple light device "services" (services are components developers build in Home Assistant). Since Home Assistant does not support composition as a native API, we expect the users of this "room" to specify the identifiers of the light services the rooms can set. Fortunately, though awkward, this can be done at runtime by reloading the configuration file of the room service. We use similar ideas to implement the universal lamp service.

Overall this leads to 3x more code relative to dSpace to implement just S1. The Home digivice (S4) is implemented in a similar fashion that leads to 4x more code (203 vs. 51 with dSpace). For S3, Home Assistant allows one to define and (re)load a configuration file that specifies the automation rule (condition, trigger, action etc.). Note that such rules are not embedded and run by the room driver (as in dSpace) but as part of the Home Assistant runtime.

At a high-level, we find that the difficulty in supporting our target scenarios with these frameworks stems from three reasons. The first is the lack of a clean separation between device state and driver code and the ability to declarative program device state. E.g., in Home Assistant, the device "schema" are defined as part of the "driver" codebase and application modules in these frameworks have to expose imperative APIs (*e.g.,* Service Calls in HASS and Capabilities in SmartThings) in order to be invoked. The second reason is the lack of native support for composing and programming aggregates of devices, which instead required clumsy workarounds as described above. Finally, writing complex control loops and automation policies involves writing plenty of low-level plumbing code since the basic building blocks

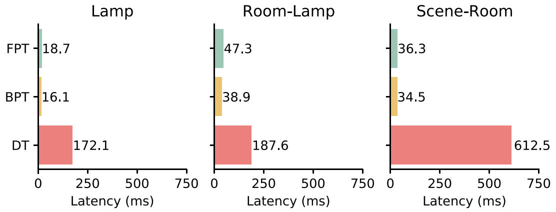


Figure 7. Latency breakdown under on-prem setup. **FPT**: forward propagation time; **BPT**: backward propagation time; **DT**: device actuation time or data processing time.

in these systems correspond to individual physical devices. That said, these other frameworks do offer many additional features that dSpace does not currently provide (*e.g.,* UI/UX modules, device onboarding). Our analysis has only focused on features that relate to the expressivity of the framework.

6.4 Programming assignments in dSpace

To further test whether dSpace is easy to program we conduct a programming assignment based study with a group of 8 student developers; 3 out of the 8 developers had prior experience programming IoT devices via simple automation rules though none had prior exposure to dSpace.

Setup. We create a sequence of programming tasks, starting from building a simple lamp digivice to a room digivice and adding a reflex/on-model policy to support motion triggered brightness. We record the time it takes for the participant to finish building each task/digi. And after they complete all tasks, we ask them to fill in a questionnaire on some feedback questions including soliciting a Mean Opinion Score (MOS). Our questions can be found in Appendix E.

Results. All users successfully completed the assignment. On average, users took approximately 16 minutes to complete the assignment, with the most time (42.7%) spent on implementing the room digivice, followed by adding the reflex for motion triggered action (28.6%), reading the docs (19.3%), and building the simple lamp digivice (12.5%). Overall, dSpace received a MOS score of 4.41 (1-5 where 5 being very easy to use/program). 75% of the developers reported confusion on whether to use the intent or the status field when calculating and setting the Room's brightness level, though all of them eventually figured it out by iterating over the code, running the digis, and observing the behaviors. 37.5% of developers said they didn't immediately realize which attributes can be accessed by the reflex API and 50% of them reported difficulty with writing correct Jq statements in the policy field (*e.g.,* one similar to Fig.3). We believe these issues can be addressed via better documentation and UI/UX.

6.5 Performance benchmarks

Metrics We measure the performance of our dSpace prototype in different environments. Our metrics include: (i) the time it takes for the intent update in the user's request to reach the leaf digi, referred to as the Forward Propagation Time (FPT) through the digi-graph; (ii) the time for the status update at the leaf digi to reach the user's CLI (assuming it is polling for updates), referred to as the Backward Propagation Time (BPT); (iii) the time for the physical device or data framework, interfaced with the leaf digi, to actuate/process the data. This is referred to as the Device-actuation Time or Data-processing Time (DT). Finally, the sum of FPT, BPT, and DT amounts to the time it takes for an intent to be fulfilled, which we refer to as the Time-to-fulfillment (TTF).

Setup: We deploy dSpace to mimic three anticipated setups: on-prem, cloud, and hybrid deployment. We use identical digis, compositions, and policies in each case. In the on-prem setup, we deployed dSpace on a local laptop in the home which runs a minikube cluster [37] In the cloud setup, we deployed a two-node k8s cluster on EC2 (one for the master/control-plane and one for the worker). Note that in the latter setup, all digis will be run on the worker node while the apiserver/etcd and other k8s controllers on the master node. In the hybrid setup, we run only the Scene digidata locally and the other components on the cloud as before.

We benchmark three scenarios: (i) `Lamp` involves updating the brightness of a single digi lamp; (ii) `Room-Lamp` updates room brightness as in scenario S1; and (iii) `Scene-Room` in which the camera generating a video stream is composed with the scene digidata which extracts objects in each frame and reports these to the room; finally, the room actuates the lamp according to the objects observed (akin to Fig.6). We repeat each experiment 3 times.

We show the latency breakdown for the on-prem deployment in Fig.7. As expected, we see that: (1) in all scenarios, the time-to-fulfillment is dominated by the time to actuate devices and process data (*e.g.,* 83.2% of the total time for the `Lamp` scenario), (2) the time spent in dSpace (FPT and BPT) increases with the number of digis involved in intent propagation and reconciliation (`Room-Lamp` and `Scene-Room` involve approximately the same number of digis, and twice as many as `Lamp`; all scenarios include the latency of communicating with the API server). These latencies are on par with those reported for equivalent operations in Kubernetes [34] and we leave further optimization of these latencies to future work. The latencies we observe in the cloud setup show similar findings and can be found in Appendix D.

Our hybrid deployment exploits our disaggregated microservices architecture to run only the Scene digidata locally while keeping other components in the cloud. This avoids sending the entire video stream to the cloud, enabling both privacy and performance benefits. In our experiments, this reduces the bandwidth consumption due to `Scene-Room` from 4.3Mbps to a negligible amount and introduces no additional latency overhead.

In summary, we find that dSpace's microservice implementation is sufficiently performant given that device actuation, data processing, and network communication overheads dominate. On the other hand, it still introduces tens of milliseconds due to propagating events between containers/pods and this may become a performance bottleneck for applications such as real-time robotics. Improving dSpace for these ultra-low latency use-cases is a topic we leave for future work (*e.g.,* leveraging low-latency alternatives for our apiserver/etcd).

7 Discussion and Related Work

Having compared dSpace to specific IoT systems in §6, we now discuss the broader context of IoT.

Industry solutions. The IoT landscape today includes a wide range of solutions that one might view 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 [5, 15, 21, 25, 35, 40, 49]. dSpace can integrate these device-centric solutions as we demonstrate in §6. 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 Samsung SmartThings, Apple, Amazon Alexa, Google Nest, Xiaomi, and IFTTT etc. [7, 11, 27, 31, 44, 50]. Finally, *Infrastructure-centric* solutions are ones in which a vendor offers its infrastructure as a platform on which clients can integrate different devices and construct and run IoT applications. All the major cloud providers [13, 14, 26] as well as Tuya Smart [46] offer such platforms.

The majority of the above are closed or proprietary solutions. However, there are multiple industry consortia and communities developing open source solutions, including: EdgeX Foundry [23], OM2M [22], KubeEdge [32], Home Assistant [29], and OpenHAB [38]; and they typically offer open source SDKs, device services, communication protocols, pub-subs, UI/UX, and automation modules.

Research. There is a large body of prior research in IoT systems closely relevant to dSpace, including: HomeOS [58], BOSS [57], Brick [52], XBOS [60], Beam [72], and Bolt [62]. Among these pioneering works, HomeOS offers PC-like abstractions for programming devices as peripherals and enables cross-device tasks [58]. BOSS provides common system primitives and services such as a query language for metadata, timeseries analytics, and fault-tolerant control for commercial buildings [57]. Beam proposes inference graphs to abstract away sensing and inference tasks, thus simplifying the development of IoT applications [72].

Relation to dSpace. In dSpace, we take inspiration from all of the above and introduce new concepts and abstractions to further simplify the construction of sophisticated smart-space scenarios: first-class and adaptive composition, flexible and higher-level abstractions, intent reconciliation, and support for richer policies that enable runtime adaptation, shared control, delegation, and so forth. As we demonstrate through our system evaluation in §6, the novel features simplify the development of smart space applications.

Programming frameworks. As a programming framework, dSpace is inspired by prior research on modular frameworks in other domains - *e.g.,* Malt [70] and Orion [59] for network management, Click [64] for packet-processing, Spark and Hadoop [8, 9] for analytics, Ray and Tensorflow [51] for AI/ML etc. dSpace differs from these in the nature of its abstractions and features which result from the needs and challenges of smart spaces.

Limitation and future works. There are important aspects of IoT system design that we do not cover in this paper. As mentioned in §2.4, current dSpace doesn't offer UI/UX support (*e.g.,* a GUI) for non-technophile users. We plan to provide such support via integrating or porting existing UI/UX solutions [17, 28]. Besides, we've evaluated dSpace under home automation scenarios and plan to extend our evaluation to multi-occupancy homes [78] and to beyond home contexts such as offices, retail locations, and campuses. Finally, we are exploring techniques to provide safety and privacy guarantees in dSpace, *e.g.,* protecting users from unsafe IoT device states via verification and/or enforcing security policies.

8 Conclusion

This paper presents dSpace, a framework for building smart space applications. With dSpace, developers can flexibly compose heterogeneous devices into versatile smart spaces that are exposed to users via high-level abstractions which are easier to program and customize. dSpace's main contribution lies in identifying the design principles and abstractions for the smart-space domain: high-level abstractions via hierarchical composition, decoupled digivice *vs.* digidata, embedded policies, intent reconciliation, and adaptive composition via policy-driven mount and yield. We show how these abstractions map to the well-known microservices design paradigm and how this paradigm should be adapted to the domain of smart-space applications. We validate our design using real-world IoT devices and scenarios. Compared to existing smart-space systems, dSpace offers: minimalism, better modularity, ease of customization via rich composition and policies, and an open implementation.

Acknowledgements

We thank our anonymous reviewers for their useful comments and feedback, and our shepherd Wenjun Hu who helped shape the final version of this paper. We are also thankful to Amy Ousterhout, Sam Kumar, Louis Ye, Zhihong Luo, Lei (Harry) Zhang, Narek Galstyan, Wen Zhang, Hong Zhang, and Gabe Fierro for the useful discussions and feedback during various stages of this work. This work was partly funded by NSF grant 1553747.

References

[1] 2020. The eXtensible Building Operating System. https://github.com/SoftwareDefinedBuildings/XBOS.

[2] 2020. How IoT devices and smart home automation is entering our homes in 2020. https://www.businessinsider.com/iot-smart-home-automation.

[3] 2020. Internet of Things (IoT) - Statistics & Facts. https://www.statista.com/topics/2637/internet-of-things/.

[4] 2020. iPropertyManagement: Smart Home Statistics. https://ipropertymanagement.com/research/iot-statistics.

[5] 2020. iRobot Ready to Unlock the Next Generation of Smart Homes Using the AWS Cloud. https://aws.amazon.com/solutions/case-studies/irobot/.

[6] 2021. Alarm Motion Detector. https://shop.ring.com/products/alarm-motion-detector-v2.

[7] 2021. Amazon Alexa. https://developer.amazon.com/en-US/alexa.

[8] 2021. Apache Hadoop. https://hadoop.apache.org/.

[9] 2021. Apache Spark. https://spark.apache.org/.

[10] 2021. apiserver. https://github.com/kubernetes/apiserver.

[11] 2021. Apple Homekit. https://developer.apple.com/homekit/.

[12] 2021. artifact. https://github.com/digi-project/sosp21-artifact.

[13] 2021. AWS IoT. https://aws.amazon.com/iot/.

[14] 2021. Azure IoT. https://azure.microsoft.com/en-us/overview/iot/.

[15] 2021. Bose SoundTouch API. https://developer.bose.com/bose-soundtouch-api.

[16] 2021. A complete, cross-platform solution to record, convert and stream audio and video. https://www.ffmpeg.org/,.

[17] 2021. Deploy and Access the Kubernetes Dashboard. https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.

[18] 2021. docker image. https://docs.docker.com/engine/reference/commandline/image/.

[19] 2021. Docker security. https://docs.docker.com/engine/security/.

[20] 2021. Dynamic Admission Control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.

[21] 2021. Dyson Pure Cool link library. https://github.com/CharlesBlonde/libpurecoollink/.

[22] 2021. Eclipse OM2M. https://www.eclipse.org/om2m/.

[23] 2021. EdgeX Foundry. https://www.edgexfoundry.org/.

[24] 2021. Etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.

[25] 2021. Geeni Smart Lighting. https://mygeeni.com/collections/lighting.

[26] 2021. Google Cloud IoT. https://cloud.google.com/solutions/iot.

[27] 2021. Google Nest, build your connected home. https://store.google.com/us/category/connected_home.

[28] 2021. Home Assistant Frontend. https://www.home-assistant.io/integrations/frontend/.

[29] 2021. Home Assistant: Open source home automation that puts local control and privacy first. https://www.home-assistant.io/.

[30] 2021. How To Manage Your Kubernetes Configurations with Kustomize. https://www.digitalocean.com/community/tutorials/how-to-manage-your-kubernetes-configurations-with-kustomize.

[31] 2021. IFTTT: Everything works better together. https://ifttt.com/.

[32] 2021. KubeEdge: A Kubernetes Native Edge Computing Framework. https://kubeedge.io/.

[33] 2021. Kubernetes Operator Pythonic Framework (Kopf). https://github.com/nolar/kopf.

[34] 2021. Kubernetes scalability and performance SLIs/SLOs. https://github.com/kubernetes/community/blob/master/sig-scalability/slos/slos.md.

[35] 2021. LIFX Smart Home Light. https://www.lifx.com/.

[36] 2021. Lightweight and flexible command-line JSON processor. https://stedolan.github.io/jq/.

[37] 2021. Minikube: Run Kubernetes Locally. https://github.com/kubernetes/minikube.

[38] 2021. openHAB: empowring smart home. https://www.openhab.org/.

[39] 2021. package informers. https://godoc.org/k8s.io/client-go/informers.

[40] 2021. Philips Hue. https://www.philips-hue.com/en-us/personal-mood-lighting.

[41] 2021. Python library for accessing LIFX devices locally. https://github.com/mclarkk/lifxlan.

[42] 2021. Ray rllib. https://github.com/ray-project/ray/tree/master/rllib/agents/marwil.

[43] 2021. Roomba Robot Vacuums. https://www.irobot.com/roomba.

[44] 2021. SmartThings. https://smartthings.developer.samsung.com/.

[45] 2021. Things Graph. https://aws.amazon.com/iot-things-graph/.

[46] 2021. Tuya IoT Platform. https://www.tuya.com/.

[47] 2021. Using RBAC Authorization. https://kubernetes.io/docs/reference/access-authn-authz/rbac/.

[48] 2021. Working with Rules. https://smartthings.developer.samsung.com/docs/rules/overview.html.

[49] 2021. Wyze. https://wyze.com/.

[50] 2021. Xiao Mi Smart Home. https://xiaomi-mi.com/mi-smart-home.

[51] Martín Abadi et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proc. USENIX OSDI*.

[52] Bharathan Balaji et al. 2016. Brick: Towards a unified metadata schema for buildings. In *Proc. ACM BuildSys*.

[53] Gary Bradski and Adrian Kaehler. 2000. OpenCV. *Dr. Dobb's journal of software tools* (2000).

[54] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* (2016).

[55] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT.. In *Proc. NDSS*.

[56] Benoit Dageville et al. 2016. The snowflake elastic data warehouse. In *Proc. ACM SIGMOD*.

[57] Stephen Dawson-Haggerty et al. 2013. BOSS: Building operating system services. In *Proc. USENIX NSDI*.

[58] 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*.

[59] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Edwin Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. 2021. Orion: Google's Software-Defined Networking Control Plane.. In *Proc. USENIX NSDI*.

[60] Gabriel Fierro and David E Culler. 2015. Xbos: An extensible building operating system. In *Proc. ACM BuildSys*.

[61] Jerrold R Griggs, Wei-Tian Li, and Linyuan Lu. 2012. Diamond-free families. *Journal of Combinatorial Theory, Series A* 119, 2 (2012), 310–322.

[62] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. 2014. Bolt: Data management for connected homes. In *Proc. USENIX NSDI*.

[63] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. 2017. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 1–35.

[64] 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.

[65] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E Culler. 2019. JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT. In *Proc. USENIX Security*.

[66] Chieh-Jan Mike Liang, Börje F Karlsson, Nicholas D Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: building an internet of safe things. In *Proc. International Conference on Information*

Processing in Sensor Networks.

[67] Barbara Liskov. 2009. The power of abstraction. *Turing Award Lecture* (2009).

[68] 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).

[69] Matt McCormack, Amit Vasudevan, Guyue Liu, Sebastián Echeverría, Kyle O'Meara, Grace Lewis, and Vyas Sekar. 2020. Towards an architecture for trusted edge iot security gateways. In *Proc. USENIX HotEdge*.

[70] Jeffrey C Mogul et al. 2020. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *Proc. USENIX NSDI*.

[71] Philipp Moritz et al. 2018. Ray: A distributed framework for emerging AI applications. In *Proc. USENIX OSDI*.

[72] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. 2016. Beam: Ending monolithic applications for connected devices. In *Proc. USENIX ATC*.

[73] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. 2014. Practical trigger-action programming in the smart home. In *Proc. SIGCHI Conference on Human Factors in Computing Systems*. 803–812.

[74] Ke Xu, Xiaoliang Wang, Wei Wei, Houbing Song, and Bo Mao. 2016. Toward software defined smart home. *IEEE Communications Magazine* 54, 5 (2016), 116–122.

[75] Tianlong Yu, Tian Li, Yuqiong Sun, Susanta Nanda, Virginia Smith, Vyas Sekar, and Srinivasan Seshan. 2020. Learning context-aware policies from multiple smart homes via federated multi-task learning. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE.

[76] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proc. ACM HotNets*.

[77] Eric Zeng, Shrirang Mare, and Franziska Roesner. 2017. End user security and privacy concerns with smart homes. In *Proc. USENIX SOUPS*. 65–80.

[78] Eric Zeng and Franziska Roesner. 2019. Understanding and improving security and privacy in multi-user smart homes: a design exploration and in-home user study. In *Proc. USENIX Security*. 159–176.

[79] Han Zhang, Abhijith Anilkumar, Matt Fredrikson, and Yuvraj Agarwal. 2021. Capture: Centralized Library Management for Heterogeneous IoT Devices. In *Proc. USENIX Security*.

[80] Serena Zheng, Noah Apthorpe, Marshini Chetty, and Nick Feamster. 2018. User perceptions of smart home IoT privacy. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–20.

Command	Description
dq build KIND	Create a new image
dq pull/push KIND	Down/upload image
dq run/stop KIND NAME	Start/stop a digi
dq mount [-d] [-y] A B	Mount/unmount/yield A to B
dq pipe [-d] A.output.x B.input.x	Pipe A's output to B's input

Table 6. Commands to build, deploy, and compose digis.

Appendix

A More on dSpace Deployment

Briefly, digis are built and deployed following standard practice of modern application development and deployment: i. developers define model schema, program the driver, and build a *digi image*; ii. push the image to a repository; one can later iii. pull, run, and/or compose the digi. dSpace's command-line, dq, provides utilities to automate these process as summarized in Table 6.

Digi image: We reuse existing container images and their tooling for the driver. A *digi image* is a simple configuration file that comprises the model schema and a container image ID pointing to the *container image* of the driver (*e.g.*, docker.io/digi/lamp), plus a content addressable hash calculated over the model schema and the container image digest [18]. These simple approaches combined lead to lightweight digi images (a few KB) easy to distribute and verified for integrity.

B More Examples on Digi Programming

B.1 Room Digivice: model

```yaml
group: digi.dev
version: v1
kind: Room
control:
  brightness: number
mount:
  digi.dev/v1/lamps: object
```

Figure 8. Schema of the Room digivice (Fig.8).

B.2 Universal Lamp

Below: Python code implementing a simple universal lamp digivice that unify across two types of lamps and intent back-propagation.

```python
import digi
import digi.on as on

import digi.util as util
from digi.util import put, first_attr, first_type

"""Universal lamp translates power and brightness
to vendor specific lamps."""

converters = {
    "digi.dev/v1/colorlamps": {
        "power": {
            "from": lambda x: "on" if x == 1 else "off",
```

```python
from digi import on
from digi.view import TypeView, DotView
# invoked upon mount or control attributes changes
@on.mount
@on.control
def handle_brightness(model):
    # chained transformation of model
    with TypeView(model) as tv, DotView(tv) as dv:
        # control attribute for room brightness
        rb = dv.room.control.brightness
        # if no lamps brightness status set to 0
        rb.status = 0
        if "lamps" not in dv:
            return
        # count active lamps
        active_lamps = [l for _, l in dv.lamps.items()
                    if l.control.power.status == "on"]
        # iterate and set active lamp brightness
        for lamp in active_lamps:
            # room brightness is the sum of all lamps
            room_brightness.status += \
            lamp.control.brightness.status
            # divide intended brightness across lamps
            lamp.control.brightness.intent = \
            room_brightness.intent / len(active_lamps)
    # At the closing of the "with" clause, changes on
    # DotView will be applied to the TypeView and then
    # to the model.
if __name__ == '__main__':
    digi.run()
```

Figure 9. Python code implementing a simple Room digivice that aggregates brightness of lamps, using filters and views.

```yaml
group: digi.dev
version: v1
kind: UniLamp
control:
  power: string
  brightness: number
obs:
  reason: string
mount:
  digi.dev/v1/lamps: object
  digi.dev/v1/colorlamps: object
```

Figure 10. Schema of the Universal Lamp digivice (Fig.10).

```python
            "to": lambda x: 1 if x == "on" else 0,
        },
        "brightness": {
            "from": lambda x: x / 255,
            "to": lambda x: x * 255,
        },
    },
    "digi.dev/v1/geenilamps": {
        "power": {
            "from": lambda x: x,
            "to": lambda x: x,
        },
        "brightness": {
            "from": lambda x: x,
            "to": lambda x: x,
        }
    },
}


# validation
@on.mount
def h(mounts):
    count = util.mount_size(mounts)
    assert count <= 1, \
        f"more than one lamp is mounted: " \
        f"{count}"


# intent back-prop
@on.mount
def h(parent, bp):
    ul = parent

    for _, child_path, old, new in bp:
        typ, attr = util.typ_attr_from_child_path(child_path)

        assert typ in converters, typ

        # back-prop logic
        put(path=f"control.{attr}.intent",
            src=new, target=ul,
            transform=converters[typ][attr]["from"])


# status
@on.mount("lamps")
def h(lp, ul, typ):
    lp = first_attr("spec", lp)

    assert typ in converters, typ

    put(f"control.power.status", lp, ul,
        transform=converters[typ]["power"]["from"])

    put(f"control.brightness.status", lp, ul,
        transform=converters[typ]["brightness"]["from"])


@on.mount("colorlamps")
def h(lp, ul, typ):
    lp = first_attr("spec", lp)

    assert typ in converters, typ

    put(f"control.power.status", lp, ul,
        transform=converters[typ]["power"]["from"])

    put(f"control.brightness.status", lp, ul,
        transform=converters[typ]["brightness"]["from"])


# intent forwarding
@on.mount
@on.control
def h(parent, child):
    ul, lp = parent, first_attr("spec", child)
    if lp is None:
        return

    typ = first_type(child)
    assert typ in converters, typ

    put(f"control.power.intent", ul, lp,
        transform=converters[typ]["power"]["to"])

    put(f"control.brightness.intent", ul, lp,
        transform=converters[typ]["brightness"]["to"])


if __name__ == '__main__':
    digi.run()
```

B.3 Yield Policy

```yaml
apiVersion: digi.dev/v1
kind: YieldPolicy
metadata:
  name: example-yieldpolicy
spec:
  source:
    kind:
      group: digi.dev
      version: v1
      name: Room
    name: lvroom
    namespace: default
  target:
    kind:
      group: digi.dev
      version: v1
      name: EmergencyAppliance
    name: emerg-app
    namespace: default
  condition: >-
    if (.source.spec.mode.status == \"emergency\")
     then true else false end
```

Figure 11. A example yield policy between a Room digivice and an Emergency appliance digivice.

C Experiences with Other Frameworks

SmartThings: SmartThings allows device grouping and group-level actions but groups cannot be nested and devices in the group have to be the same type; it supports composite devices but they can only be defined/composed statically at code-level and cannot be nested. In AWS IoT, one can adapt the workflow model in the Things Graph to "emulate" HL abstractions and multi-level hierarchy (hypothetically; we cannot find such examples in the public regime), but user cannot interact directly with the abstraction (e.g., setting room's mode) and does not have mechanisms to back propagate the intents and handle policy conflicts. Etc.

Like Home Assistant, SmartThings does not support dynamic composition natively (neither the composite device and parent-child smartapp APIs work in our favor), so we applied similar circumvention as in Home Assistant - creating a parent service and a child service and have the parent call the child imperatively. However, in SmartThings, we are not able to create a universal lamp service and have the room service to call it. We suspect this is due to the universal lamp service we created are not allowed to be registered to the SmartThings runtime (its SDK is semi-open source, so we cannot confirm what's going on underneath). We ended up implementing just the room that directly talked to 3 different lamps. Nonetheless, we programmed SmartThings using its nodejs library and also tried out their Groovy library. For both libraries, similar to the case of Home Assistant, one needs to write a substantial amount of code to do low-level plumbing that is not directly related to the application logic. In SmartThings, this includes setting up the "pages" (as part of their UI/UX logic) and adding "capabilities" - that for each device actuation one needs to define a capability to define the values involved in the action and a routine/function that executes. Note that we are not saying these extra chores aren't useful features, rather we argue there is no way we write application logic without handling these complexity.

Implementing S1 in Home Assistant: To implement the aggregated brightness in Home Assistant, we first created a universal lamp "service" that has all vendor lamp "platforms" registered. For each type of vendor lamp, we wrote a separate set of setup code as part of the lamp service, plus their individual event handlers. In each handler, we also need to explicitly subscribe and invoking other services (this is also not declarative, e.g., to turn off a light, one calls a SERVICE_TURN_OFF). For the room service, we do similar setup and event handling plumbing.

At a high-level, both HASS and SmartThings share similarities with dSpace in terms of their programming model: developers write event-driven handlers that update device states asynchronously in the manner typical of event-driven programming under a pub-sub model. Yet under this umbrella term, important implementation differences remain, e.g., dSpace separates the declarative components (models) and imperative components (driver) these other frameworks define device state inline as part of their "driver" codebase. That said, both frameworks support many other features that are critical for IoT applications dSpace does not currently support such as UI/UX modules.

Data-driven policies: All frameworks have support on integrating data analytics frameworks and data-driven policies, among which AWS IoT supports dataflow composition natively. For access policies, no frameworks have support for complex access policies that involve dynamic composition, shared control, and control delegation, though most frameworks provide support for access control (e.g., ACLs) and authorization (e.g., OAuth).

D Performance Benchmarks under Cloud Setup

E Programming Assignment: Questionnaire

We collect answers to the following questions to each participant after they complete the assignment:

- Have you used IoT devices before?
- Have you programmed an IoT device before?

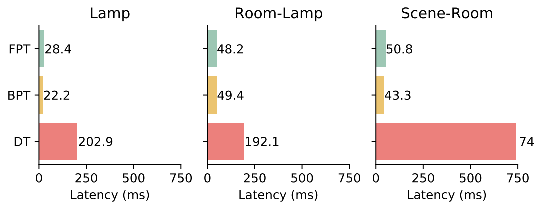


Figure 12. Latency breakdown under cloud setup (avg. ping from on-prem: 9ms). The physical lamp and cameras run on-prem but the rest of dSpace components are on EC2 cloud.

- Which part of the tutorial/programming assignment you found most difficult?
- Does dSpace help with addressing those difficulties you encountered?
- Range from 1-5 where 1 means it is very hard to use/program and 5 very easy to use/program how would you score dSpace?