The Internet of Things in a Laptop: Rapid Prototyping for IoT Applications with Digibox

Silvery Fu, Hong Zhang, Sylvia Ratnasamy, Ion Stoica
UC Berkeley

Abstract

Digibox is a prototyping environment for IoT applications. It enables a novel scene-centric prototyping where developers can program an ensemble of simulated devices to capture not only their individual but also their coordinated behaviors, making it possible to test, debug, and evaluate the behaviors of an IoT application. Using Digibox, developers can download and reuse existing scenes, customize, and repurpose them towards developing new applications; or replicate others' experiment results from scientific research. Digibox's Kubernetes-based runtime further allows developers to easily scale the prototyping environment from a single laptop to a cluster running simulated devices and scenes at a scale appropriate to the application.

CCS Concepts

• **Computer systems organization** → *Embedded and cyber-physical systems*; • **Software and its engineering** → *Abstraction, modeling and modularity*;

Keywords

IoT, simulation, design principles, framework

ACM Reference Format:
Silvery Fu, Hong Zhang, Sylvia Ratnasamy, Ion Stoica. 2022. The Internet of Things in a Laptop: Rapid Prototyping for IoT Applications with Digibox. In *Proceedings of The 21st ACM Workshop on Hot Topics in Networks (HotNets'22).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3563766.3564087

1 Introduction

The proliferation of IoT devices is giving rise to exciting new applications such as smart spaces [4, 18, 24, 30], intelligent transportation [27, 32], urban sensing [12, 14, 39], and industrial automation [2, 11, 37]. For instance, a smart building app may monitor room occupancy, alert building managers about overcrowding during a pandemic, or predictively adjust lighting and HVAC settings [16, 30], while a supply chain app can track cargo and inventory conditions to audit, automate, and optimize operational logistics [4, 11].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
HotNets'22, November 14-15, 2022, Austin, Texas
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9899-2/22/11.
https://doi.org/10.1145/3563766.3564087

Prototyping environments are key enablers for application research and development. For IoT applications, it can be time-consuming and cost-prohibitive to set up a testbed using real-world devices [41, 43, 44] or physics engines [22]. Meanwhile, we notice there are extensive works in the systems and networking community that provide easy-to-use prototyping environments, *e.g.,* NS, Mininet, iBox [21, 34, 47]; and in AI/ML, robotics, and autonomous driving communities likewise [38, 42, 48, 50]. These purpose-built prototyping environments are used extensively in research, class, training, demoing, validation, product development and so forth. Inspired by the huge success of these projects, we aim to build a flexible and extensible prototyping environment for IoT applications[1] with the following design goals:

- **Generality:** It can support a wide range of existing and emerging IoT applications.
- **Ensemble support:** Developers can easily program the test cases as an ensemble of devices capturing their correlated status and behaviors.
- **Interactivity:** They can interact with the simulated devices during testing as if these were real devices.
- **Reproducibility:** Both the setup and the experiment results can be shared and replicated.
- **Scalability:** Easy to run a few and tens of simulated devices in a laptop to thousands and more in cloud or machine cluster.
- **Customizability and reusability:** It should be simple to extend existing simulated devices and reuse them when prototyping new applications.

Meeting these goals simultaneously is non-trivial. Compared to prototyping with real-world testbed or physics engines, existing device simulators [17, 20, 29] are more easily accessible and achieve much better scalability. However, they typically simulate *individual* devices by generating data for a particular device in isolation. As a result, such simulators fall short in capturing real-world events such as device interactions (*e.g.,* correlating the detection of motion across occupancy sensors) and human inputs (*e.g.,* tuning the light intensity of a simulated lamp). Such a lack of ensemble support and interactivity makes it difficult for developers to correctly test, validate, and demonstrate application correctness under more realistic and sophisticated scenarios.

Further, due to a lack of canonical tooling and native support for *reproducibility* in existing IoT prototyping approaches, it can be tedious for others to reproduce the setup

[1] Note that our focus in Digibox isn't on prototyping device hardware, firmware, or driver [51] but the applications that leverage device capabilities.

and results. For example, we observe that 11 recent publications in system/networking and security conferences (*e.g.,* EuroSys, MobiSys, SOSP, BuildSys, Usenix Security [33, 43–46, 52, 53]) use 8 different prototyping frameworks or tools. As such, both the authors and those who attempt to replicate the results need to spend significant efforts dealing with documentations, scripts, and configurations.

This paper proposes Digibox, a prototyping environment that achieves these goals simultaneously. To do so, Digibox enables a novel *scene-centric* prototyping with two abstractions: *mockup device* (or mock for short) and *scene*. The mock simulates individual device behaviors (*e.g.,* sensors generating data, actuators responding to application/user commands), while the scene may generate events (*e.g.,* human presence in a room) and ensemble the behaviors of mocks *attached* to it (*e.g.,* the room scene correlates the detected motion of mock occupancy sensors attached to it and in response to human presence). Developers can easily program mocks and scenes–their event generation and simulation logic–using Digibox's programming library in Python. Digibox uses containers and Kubernetes [19] to run mocks and scenes as microservices on a single laptop or a cluster. Moreover, note that the scene-centric design achieves a clean separation of app logic (*i.e.,* how the application processes data and reacts) and scene logic (*i.e.,* how devices coordinately behave). This decoupling simplifies application development, and improves reusability of the testing scenarios (*i.e.,* the scenes).

To facilitate reproducibility, Digibox implements the concept of *Infrastructure-as-Code* (IaC) [31] where developers can describe and version the setup (*i.e.,* mocks, scenes, and how they are related) succinctly in a standard configuration file that others can download and recreate. Besides reproducing the setup, Digibox logs all events, actions, and messages generated by the mocks and the scenes to allow easy debugging, replay, and sharing any experiment results.

Digibox is open-source and under active development.[2] We create a mock and scene repository using Digibox which currently contains 20 device mocks (*e.g.,* occupancy, fan, lamp, HVAC) and 18 scenes (*e.g.,* building, campus, retail, supply chain, home) with more underway. We hope to create a virtuous circle in which researchers and developers contribute new mocks, scenes, and test data while building upon others' contributions. Finally, as IoT continues to grow - devices become more pervasive and gain new sensing and actuating capabilities [1, 5, 27] - there are open design questions and challenges such as how to incorporate high-fidelity physical effects in Digibox which we'll discuss in the paper.

2 IoT Applications and Prototyping

An IoT application communicates with devices[3] via device drivers, sending/receiving messages either directly from/to devices, or via a device hub [23], or a message broker [9]. The messages sent to a device (*e.g.,* a command setting a lamp's power to "on") may trigger an actuation (*e.g.,* the physical lamp powers on) and the ones received by the application may contain status updates from the device, *e.g.,* the lamp reports its current power status, or an occupancy sensor reports motion being detected.

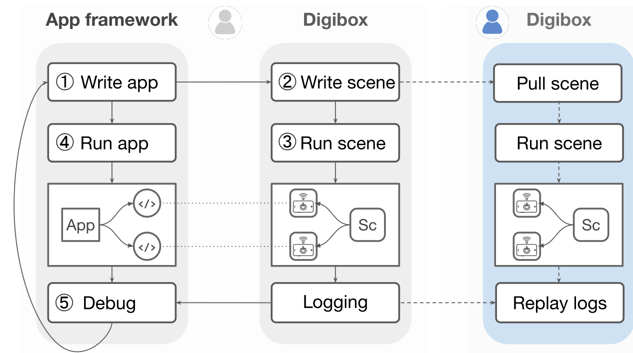

Figure 1: Digibox Workflow. Using Digibox for application prototyping and reproducibility.

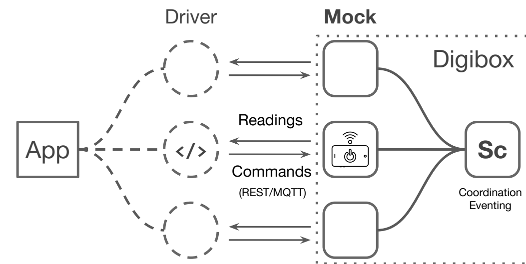

Figure 2: Application and mocks exchange messages in Digibox.

IoT applications are typically developed in the context of a *scene*. We define a scene as the environment where the application runs, which includes the devices, their interactions with the physical world (*e.g.,* human actions, other devices), and the applications. While the exact semantics of a scene may vary across applications, the common property to highlight here is that the device behaviors are usually correlated and affected by the events in the scene. For example, consider a meeting room with lamps and occupancy sensors, one on the ceiling and the others on the desks. In this scenario, the lamps might be turned on/off by humans and the occupancy sensors have correlated motion readings (*e.g.,* when a desk occupancy is triggered the ceiling one must be so too). Therefore, the prototyping environment must allow developers to specify test cases over these devices as an *ensemble* to reflect the correct scene semantics and thereby test an application in a robust manner, *e.g.,* to test whether the app calculates room occupancy in face these two types of sensors and reacts to human actions correctly.

Digibox: Scene-centric prototyping. Digibox provides first-class abstractions and primitives for developers to program not only the behaviors of individual simulated devices (*i.e.,* the device logic) but also their correlated and interactive behaviors as an ensemble (*i.e.,* the scene logic). For instance, in the meeting room scenario, developers can program the room scene to ensure that the readings from the desk occupancy sensors and the ceiling sensor are always consistent. Note

[2] https://digi.dev/digi; mocks and scenes: https://digi.dev/mocks.

[3] A device could be an actuator, a sensor, or a mixture of both.

that the scene logic differs from how an application will process these device data (*i.e.,* the app logic), *e.g.,* deriving the room occupancy based on these occupancy sensor readings. A testbed in Digibox can include multiple scenes.

Fig.1 depicts the high-level workflow of using Digibox to prototype applications and for others to reproduce the setup and results. For prototyping, as shown in the left two columns, a developer writes her application in an application framework (①), *e.g.,* SmartThings, Home Assistant, dSpace [13, 25, 44]), while writing scenes (②) using Digibox's programming library The developer may specify the scene logic relevant to the application functionalities in the scenes; alternatively, she can download, reuse, and customize existing scenes created by other developers. The developer can run the scene(s) with Digibox's command-line tool (③), run the application (④), and have the application use the scene.

Fig.2 zooms in and shows how the application talks to the scene at run-time. The application sends and receives messages to/from the mock with communication protocols such as MQTT or REST commonly used by real-world devices. During execution, Digibox logs the events, actions, and messages generated by/in the scene so that the developer can use them to debug/analyze the application (⑤). The developer may iterate on this workflow by updating the application and the scenes. Further, the developer can upload and share her Digibox scenes with others. As shown in the rightmost column of Fig.1, others can download/pull the scene, run the scene, and replay the logs also shared by the developers.

It's worth noting that with the above workflow, Digibox achieves a *clean separation of app logic and scene logic*. This simplifies the development of the two given they each can use dedicated programming/runtime support and is easier to evolve. This also improves reusability since one scene can be shared and reused to test many different applications.

3 Digibox Workflow

We first present Digibox's core abstractions, mockup device (mock) and scene controller (scene), followed by how to write mocks and scenes (§3.2), how to use them during prototyping (§3.3), along with other useful features of Digibox including sharing, logging, and reproducing testbed and experiments (§3.4, §3.5). We use a smart building application as the walkthrough example with details described as we go.

3.1 Core Abstractions

A testbed in Digibox consists of *mockup devices* (mocks) and *scene controllers* (scenes). A mock *simulates the behavior* of a real device and keeps track of the device *status*. A mock can generate events, *e.g.,* a mock occupancy sensor can generate random motion readings. A scene *coordinates* the mocks that are *attached* to it by generating events and configuring the correlated status of the mocks accordingly. Specifically, each mock and scene has the following components:

Model. A model contains a collection of key-value pairs to describe the *status* of the mock or scene as well as the desired status of it (*i.e., intent*). Fig.3 depicts a few examples.

```
1  meta:
2    type: Occupancy
3    version: v1
4    name: O1
5    managed: true
6    # ..more config
7  triggered: true
8  ---
9  # Lamp L1
10 # ..
11 power:
12   intent: "on"
13   status: "on"
14 intensity:
15   intent: 0.2
16   status: 0.4
```

```
1  meta:
2    type: Room
3    version: v2
4    name: MeetingRoom
5    managed: true
6  human_presence: true
7  attach: [L1,O1,..]
8  ---
9
10 meta:
11   type: Building
12   version: v3
13   name: ConfCenter
14   managed: false
15 num_human: 2
16 attach: [MeetingRoom,..]
```

Figure 3: Example models: mock occupancy sensor (left-top), lamp (left-bottom, "meta" fields omitted), room scene (right-top), and building scene (right-bottom). The occupancy sensor O1 and lamp L1 are attached to the MeetingRoom, which is in turn attached to the ConfCenter building.

Event generator. It generates and mimics real-world events (*e.g.,* motion in an occupancy sensor, L8 in Fig.4) by which the mock or scene configures the model status.

Simulator. A simulator is a piece of code that contains the simulation logic of the mock or the scene. For mocks, this logic specifies how the device behaves according to the intent (*e.g.,* configuring light intensity, L19-28 in Fig.4), which is similar to today's device simulators [20, 29]. For scenes, the simulator provides ensemble support by coordinating the mocks attached to it (*e.g.,* configuring mock occupancy sensors, L8-19 in Fig.5).

Logger. It logs events, changes to the models, and messages for reproducing test runs and debugging.

Further, scenes in Digibox can be *nested*, *i.e.,* a scene can be attached to higher-level scene (*e.g.,* rooms to a building) allowing the higher-level scene controller to write to the lower-level scene's status (*e.g.,* the building scene can configure human presence in different rooms, L28-40 in Fig.5). This allows one to reuse mocks and scenes in a hierarchy. Finally, the model also includes metadata (field "meta") including type, version, name, whether they are managed (§3.2) and other configuration parameters used for event generation (*e.g.,* interval, random seed) or simulator (*e.g.,* value range).

3.2 Writing Mocks and Scenes

To write a new mock or scene, developers first define the schema of its model - which fields/key-value pairs should the model include (*e.g.,* a "triggered" field for an occupancy sensor). Developers can then program the mock or scene using the `dbox` library in Python. Fig.4 presents simple examples for a mock occupancy sensor and a mock lamp. For the event generation, developers supply a handler (*e.g.,* L6-9) that updates the status of the mock (*e.g.,* indicating whether the sensor detects a motion). The decorator `dbox.loop` allows the handler to be run periodically with configurable parameters (*e.g.,* seed, loop interval or distribution) in the code or

```python
import random
from digi import dbox, on

"""Mock occupancy sensor"""
# handler for event generation
@dbox.loop(cond=dbox.managed)
def gen_event():
    motion = random.choice([True, False])
    dbox.model.update({"triggered": motion})
# handler for simulation
@on.model
def sim(status):
    dbox.broker.publish(status)
"""Mock lamp (in separate .py)"""
# handler for simulation
@on.model
def sim_intensity(model):
    power = model["power"]
    intst = model["intensity"]
    if power["status"] == "off":
        intst["status"] = 0.0
    else:
        intst["status"] = intst["intent"]
    dbox.broker.publish({
        "power": power,
        "intensity": intst})
```

Figure 4: Example mock occupancy sensor and mock lamp.

from the "meta" field in the model. Developers supply handler(s) that perform simulation, *e.g.,* L11-13 specifies when the sensor's model is updated it will generate and publish a message to the default message broker. As another example, we can specify how the light intensity status is set for a lamp according to its current power status (L16-26).

Developers can program scenes in a similar manner. Fig.5 shows a simple example of a room and a building scene. Different from the mock whose simulation handlers provide device behaviors, developers supply simulation handlers that coordinate the status of mocks attached to it. For instance, Fig.5 includes the simulation handler of the room scene (L7-17) which ensures the occupancy sensors of two different types - room-level (type: Occupancy) and desk-level (type: Underdesk) - in the room have consistent "triggered" value based on the human presence.

Fig.6 shows an example mock/scene hierarchy corresponding to the code examples in Fig.4, Fig.5, and models in Fig.3 (not all are shown). In this example, the building scene generates number of humans in the building and randomly assigns them to the rooms in the building and configure the human presence (*e.g.,* L35-37). The room in turn configures the status of the occupancy sensors in the room, *e.g.,* to ensure the occupancy sensors readings are consistent (L10-16).

3.3 Using Mocks and Scenes

In what follows, we elaborate how to use Digibox to run the prototyping workflow mentioned earlier in Fig.1.

Using dbox command line. Developers use the `dbox` command line tool to create and interact with mocks and scenes. Table 1 lists the APIs in Digibox. As an example, one calls `dbox run Lamp L1` to create a mock lamp named "L1". Likewise, to create a scene "MeetingRoom", they call `dbox run Room MeetingRoom`. Developers can then attach the mocks to scene and scene to other scenes with the `dbox attach` command.

```python
"""Room scene."""
@dbox.loop(cond=dbox.managed)
def event():
    presence = random.choice([True, False])
    dbox.model.update(
        {"human_presence": presence})
@on.model
def sim_occupancy(model, atts):
    presence = model["human_presence"]
    occs = atts.get("Occupancy", {})
    for _, occ in occs.items():
        occ["triggered"] = presence
    desks = atts.get("Underdesk", {})
    for _, desk in desks.items():
        if not presence:
            desk["triggered"] = False
        ...
"""Building scene (in separate .py)."""
@dbox.loop(cond=dbox.managed)
def event():
    # decide number of humans in building
    num_human = random.randint(0, 2)
    dbox.model.update(
        {"num_human": num_human})
@on.model
def sim_room_presence(sv, atts):
    rooms = atts.get("room", {})
    names = list(rooms.keys())
    if len(names) < 1:
        return
    # decide which rooms have human
    picked = set(random.choices(names,
        k=model["num_human"]))
    # configure room status
    for name, room in rooms.items():
        room["human_presence"] = \
            name in picked
```

Figure 5: Example room scene and building scene.

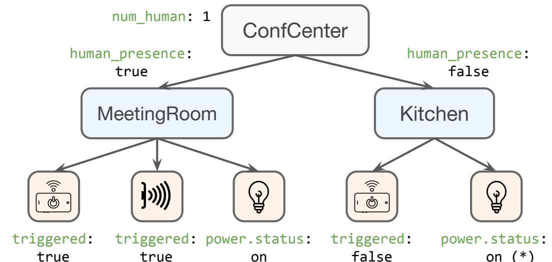

Figure 6: Mocks and scenes in the smart building application.

Testing applications. Developers test the application by having it use/communicate with the mocks as if they were real devices. To check and debug/reason about the application, developers can obtain the status of the mocks or scenes via `dbox check`, which displays the current model states in the console, any time during the execution; or use `dbox watch` to monitor the models continuously. Digibox will also log the events, messages, and changes of the model generated by the mocks and scenes (more in §3.5), by which developers can also analyze Digibox logs to validate whether the application

API	Functionality
dbox run/stop type name	Run/stop a mock or scene
dbox check/watch name	Display model changes in console
dbox attach name name	Attach a mock or scene to a scene
dbox commit type name	Update or create a mock or scene type
dbox pull/push type	Up/download a mock or scene
dbox replay name	Replay the scene trace

Table 1: Digibox command-line APIs.

behaves as expected. Besides, akin to the common practices in software testing, developers can specify test cases (*e.g.,* inputs and expected outputs) to validate application correctness. For example, developers can pause event generation in the scene (*e.g.,* setting building's "managed" field to "true", L14 in Fig.5) and add input-output pairs (*i.e.,* scene status and the expected mock status).

Further, developers can specify *scene properties*, conditions that should be met in the scene, *e.g.,* "the lamp should always be turned off when the occupancy sensor is not triggered" as disallowed model states expressed in k-v pairs, which Digibox checks at run-time and reports any violations. We are working on supporting scene properties with more sophisticated approaches such as temporal logic [53].

Interacting with mocks. During testing, developers can interact with the mocks in Digibox to emulate real-world user interactions such as turning on/off lamps. Fig.3 shows the models for the simple room scene and the mocks (lamp and occupancy sensor) in it. Developers can edit the model of the mock with `dbox edit L1` to set the intent fields of the lamp L1 (*e.g.,* L12 in Fig.3) and test how the application reacts to the user turning on/off the lamp. Developers can also introduce or remove mocks from the scene, *e.g.,* to add lamp L1 to the meeting room one can run `dbox attach L1 MeetingRoom` and to remove L1 with the `-d` option.

3.4 Sharing and Customizing

Developers can *commit* their Digibox setup which will generate a set of shareable configuration files describing all the mocks and scenes - based on their models - and how they are attached to one another in the current setup. These files point to the configuration files of mocks and scenes which in turn point to the container images (of the simulator and event generator code) which are managed by Digibox. As such, developers can share and publicize (via `dbox push`) the configuration files to the *scene repository* where others can download (via `dbox pull`) the files. By default, Digibox uses Git and GitHub [10] as the scene repository following standard practice of Infrastructure-as-Code [31] (§4).

One can add new mocks and new scenes and/or customize existing ones with updated device/scene logic. A typical workflow is where developers create mocks and attach to a scene (with `dbox attach NAME s1`) and use the `dbox commit s1` command to create a new version of the scene that includes all the mocks or scenes attached to it. This new scene can then be publicized in the scene repository.

3.5 Reproducing and Logging

To recreate the setup, the Digibox at the receiver side will parse the shared configuration files, and run the mocks and scenes and attach the mocks to scenes and scenes to scenes accordingly, which includes pulling the container images [8] from the container repository [7].

Logging and replay. In addition to reproducing the setup, Digibox also allows reproducing a trace. Digibox logs the model changes, generates events, and sends messages for each mock and the scene. For example, when the scene sets the power of the lamp to off, this is logged both at the scene controller and the lamp mock. Traces are shared as a zip file which the recipient Digibox can parse and replay. A simplified sample trace is shown below:

```
1 {name:confcenter,num_human:1,ts:00:01}
2 {name:meetingroom,human_presence:false,ts:00:03}
3 {name:kitchen,human_presence:true,ts:00:03}
4 {name:o1,triggered:true,ts:00:04}
5 {name:l1,triggered:true,ts:00:05}
```

To replay this trace, developers first run the building scene corresponding to this trace. Then developers run `dbox replay building-trace` which instructs the mocks and scenes to replay the behaviors according to the trace for debugging, analysis, and performance validation.

Logging with real devices. Note that the logging functionality can be used when real devices are used by the application by including the dbox library and explicitly using `dbox.logger`. This allows Digibox to capture real-world device behaviors which can be used by others to test or improve the fidelity of emulation which we'll discuss in §5.

4 Deployment and Scalability

We implement Digibox (runtime) using a recent open-source IoT framework dSpace [44] where we deploy each mock and scene controller as a "digi" microservice on Kubernetes. We use EMQX [9] as the default MQTT message broker. Specifically, we made the following design choices:

Containers, Kubernetes, and dSpace. Similar to Mininet, containers allow us to simulate multiple devices in a laptop and easily share them. Using Kubernetes allows us to scale the scene across multiple machines in a cloud environment. Using dSpace allows Digibox to easily manage the lifecycles of mocks and scenes as digi [44] microservices.

Infrastructure-as-Code (IaC). Refers to the use of declarative configurations that uniquely reproduce the setup. One can use a version control (VCS) to manage these configuration files. For Digibox, this allows users such as researchers to easily reproduce the experiment setup (*e.g.,* during artifact evaluation) as described in §3.5.

We microbenchmark the Digibox in local and cloud environments to understand whether Digibox is performant at different deployment scales. For the local environment, we run Digibox in a Macbook Air M1 laptop (with Docker engine [6] and its Kubernetes distribution) where we are able to run 50 occupancy sensors in 2 room scenes with average request latency (the time it takes for a REST GET to return a mock's status) under 20ms. It's able to run 1000 occupancy sensors across 100 rooms and 5 buildings with 2 m5.xlarge EC2 instances, with the average request latency (network delay included) under 60ms. Since prior work [44] shows

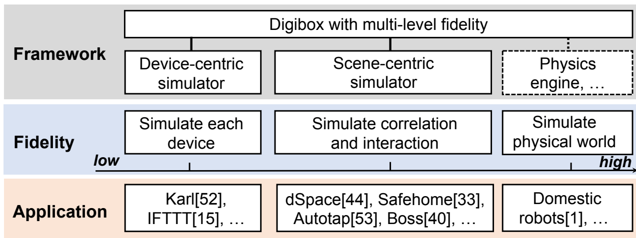

Figure 7: Simulation fidelity. (1) IoT applications require varied levels of fidelity; (2) Scene-centric simulator supports a wide range of apps; (3) Extending Digibox to support higher fidelity levels.

that device message and actuation latency can take tens to hundreds of milliseconds, our early results suggest Digibox's microservice-based implementation is sufficiently performant. We'll discuss the scalability issues further in §5.

5 Use Cases and Related Work

In what follows, we briefly discuss IoT applications, highlight how Digibox may help facilitate their research and development, and how it complements related work.

Smart spaces. This refers to the applications that leverage IoT devices in the living spaces to improve the quality, efficiency, and safety of our daily life/work such as smart homes [33, 33, 41, 44, 52, 53] and smart buildings [36, 40, 43]. Research work here can involve highly different testbed setup (*e.g.,* devices, scripts, configurations) and Digibox can simplify the steps to reproduce the setup and results, *e.g.,* for system artifact evaluation [28]. Besides, for smart space products/apps [13, 15, 16, 25], developers can leverage Digibox to build mocks and scenes for testing which can be (re)used and customized across different applications and frameworks.

Supply chain logistics. This includes applications that monitor and optimize the supply chain operations [3, 11]. Supply chain applications can incorporate data feeds from IoT devices spans across different locations and administrative domains (*e.g.,* transportation companies) with a large amount of IoT devices. Besides the aforementioned benefits, Digibox can help model these scenes with its microservice-based architecture and the ability to scale to large deployment sizes.

Urban sensing. These applications involve having mobile devices such as users' phones to collect data about the environment (*e.g.,* occupancy, temperature, noise levels); the data are later aggregated across users to provide insights. Prototyping the urban sensing applications often need to cope with the device mobility which can be emulated by dynamically re-attaching mocks to different scenes in Digibox.

Device simulators. There exist device simulators either as part of IoT frameworks [17, 26] or as standalone products [20, 29]. These simulators provide scalable simulation of individual devices, similar to the mocks in Digibox. However, they lack native support for scene-centric prototyping (programming scene logic, sharing, customizing, and reproducing scenes etc.) which Digibox targets at addressing.

IoT frameworks. While there exist numerous IoT frameworks [13, 25, 41, 44, 52] for application development in industry and academia, Digibox works in complementary with existing IoT frameworks: developers build the application (*i.e.,* program the app logic) using IoT frameworks while building scenes (*i.e.,* program the scene logic) using Digibox to test the functionalities and performance of the application.

6 Open Challenges and Research Questions

High-fidelity simulation. As shown Fig.7, IoT applications may require different levels of *fidelity* of the simulation. First, correctly simulating each individual device is sufficient for some home automation applications [15, 52] and thus device-centric prototyping is good enough. Second, for a wide range of IoT applications [15, 44, 52], correctly emulating the correlation across devices is important, but faithfully emulating the physical world is an overkill. For these applications, scene-centric prototyping, as we proposed in this paper, is a great fit. Third, more forward-looking applications such as domestic robots [1] require simulating the physical effects in the real-world to make the right actions, including more comprehensive support for human interactions and device mobility.

While Digibox currently supports both device-centric and scene-centric prototyping, how to enable Digibox to cover higher-level fidelity remains an interesting future work. In particular, Digibox should provide additional APIs that allow developers to easily model (1) Human interactions and device mobility; (2) Hardware intricacies such as device actuation delays, faults/failures, and network connectivity between devices; and (3) Physical effects, *e.g.,* via integrating with existing physics engines [49, 50].

Efficient simulation. While the microservice-based implementation allows Digibox to scale to large amounts of mocks and scenes (§4), an open question is how to make these large-scale simulations more *efficient*, *i.e.,* running a higher number of mocks/scenes with a fixed amount of compute resource budget. E.g., given the event-driven nature of IoT apps, whether/how we can leverage Function-as-a-Service (FaaS) to run the simulator logic of mocks and scenes.

Supporting new applications. Digibox currently provides 20 simulated devices and 18 scenes; still, simplifying the task of supporting the broad spectrum of IoT applications (§5) and their test cases remains challenging. First, there are devices/scenes that are highly complex and thus require expert-based modeling (*e.g.,* HVAC systems in commercial buildings). As such, Digibox should allow developers and researchers to easily integrate Digibox with existing modeling tools (*e.g.,* using Brick ontology [36] in a building scene to model the HVAC systems). Further, the IoT market is highly fragmented today: devices from different vendors may differ in the command/message schema, format, and behaviors, even when they provide similar functionalities. While we envision/encourage community support for contributing and maintaining the mocks (and scenes), manually creating and maintaining the mocks can be a tedious task. We are investigating technical solutions such as schema inference [35] and programming synthesis [53] to simplify/automate the generation and maintenance of mocks and scenes.

References

[1] The next generation of home robots will be more capable — and perhaps more social. https://www.washingtonpost.com/technology/2021/11/10/home-robots-more-personal/, 2021.
[2] Build an intelligent network for your smart factory. https://www.cisco.com/c/m/en_us/solutions/internet-of-things/intelligent_network_smart_factory.html, 2022.
[3] The connected operations cloud. https://www.samsara.com/, 2022.
[4] The cornerstone solution for fresh. https://www.afresh.com/, 2022.
[5] Density: Trusted space analytics for a flexible workplace. https://density.io/, 2022.
[6] Docker engine overview. https://docs.docker.com/engine/, 2022.
[7] Docker hub: Build and ship any application anywhere. https://hub.docker.com/, 2022.
[8] docker image. https://docs.docker.com/engine/reference/commandline/image/, 2022.
[9] Emqx: Distributed mqtt broker for iot. https://www.emqx.io/, 2022.
[10] Github. https://github.com/, 2022.
[11] Go beyond supply chain visibility with logistics intelligence. https://www.cargosense.com/, 2022.
[12] Helping federal agencies accelerate innovation through public participation. https://www.citizenscience.gov/, 2022.
[13] Home assistant: Open source home automation that puts local control and privacy first. https://www.home-assistant.io/, 2022.
[14] How much urban area can we monitor by putting sensors on taxis? http://senseable.mit.edu/urban-sensing/, 2022.
[15] Ifttt: Everything works better together. https://ifttt.com/, 2022.
[16] Intelligent workplaces. https://comfyapp.com/, 2022.
[17] Iot device simulator. https://go.aws/3eiNjR1, 2022.
[18] Is the office dead? what covid-19 means for the future of property tech. https://bit.ly/3zqvBmT/, 2022.
[19] Kubernetes: Production-grade container orchestration. https://kubernetes.io/, 2022.
[20] Mimic applications. https://bit.ly/3Mn1iC6, 2022.
[21] Network simulator. https://www.nsnam.org/, 2022.
[22] Nvidia omniverse. https://www.nvidia.com/en-us/omniverse/, 2022.
[23] Samsung smartthings hub. https://www.samsung.com/us/smart-home/smartthings/hubs/samsung-smartthings-hub-f-hub-us-2/, 2022.
[24] Smart retail & industry 4.0 – pushing sales into the future. https://www.viewsonic.com/library/business/smart-retail, 2022.
[25] Smartthings. https://smartthings.developer.samsung.com/, 2022.
[26] Smartthings simulator. https://stdavedemo.readthedocs.io/en/latest/device-type-developers-guide/simulator-metadata.html, 2022.
[27] Streetlight data. https://www.streetlightdata.com/, 2022.
[28] Systems research artifacts. https://sysartifacts.github.io/, 2022.
[29] Test automation for enterprise iot apps. https://iotify.io/, 2022.
[30] What is a smart building? https://www.cisco.com/c/en/us/solutions/smart-building/what-is-a-smart-building.html, 2022.
[31] What is infrastructure as code? https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code, 2022.
[32] What really works in iot: smart transportation. https://www.fierceelectronics.com/electronics/what-really-works-iot-smart-transportation, 2022.
[33] S. B. Ahsan, R. Yang, S. A. Noghabi, and I. Gupta. Home, safehome: smart home reliability with visibility and atomicity. In *Proc. ACM EuroSys*, 2021.
[34] S. Ashok, S. S. Duvvuri, N. Natarajan, V. N. Padmanabhan, S. Sellamanickam, and J. Gehrke. ibox: Internet in a box. In *Proc. ACM HotNets*, 2020.
[35] M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Parametric schema inference for massive json datasets. *The VLDB Journal*, 28(4):497–521, 2019.
[36] B. Balaji et al. Brick: Towards a unified metadata schema for buildings. In *Proc. ACM BuildSys*, 2016.
[37] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson. The industrial internet of things (iiot): An analysis framework. *Computers in industry*, 101:1–12, 2018.
[38] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.
[39] J. A. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M. B. Srivastava. Participatory sensing. 2006.
[40] S. Dawson-Haggerty et al. Boss: Building operating system services. In *Proc. USENIX NSDI*, 2013.
[41] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and V. Bahl. The home needs an operating system (and an app store). In *Proc. ACM HotNets*, 2010.
[42] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*. PMLR, 2017.
[43] G. Fierro and D. E. Culler. Xbos: An extensible building operating system. In *Proc. ACM BuildSys*, 2015.
[44] S. Fu and S. Ratnasamy. dspace: Composable abstractions for smart spaces. In *Proc. ACM SOSP*, 2021.
[45] W. Kim, S. Lee, Y. Chang, T. Lee, I. Hwang, and J. Song. Hivemind: social control-and-use of iot towards democratization of public spaces. In *Proc. ACM MobiSys*, 2021.
[46] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler. Jedi: Many-to-many end-to-end encryption and key delegation for iot. In *Proc. USENIX Security*, 2019.
[47] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proc. ACM HotNets*, 2010.
[48] S. Lemaignan, A. Hosseini, and P. Dillenbourg. Pyrobots, a toolset for robot executive control. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.
[49] F. Messaoudi, G. Simon, and A. Ksentini. Dissecting games engines: The case of unity3d. In *2015 international workshop on network and systems support for games (NetGames)*, pages 1–6. IEEE, 2015.
[50] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, 2012.
[51] X. Vilajosana, P. Tuset, T. Watteyne, and K. Pister. Openmote: Open-source prototyping platform for the industrial iot. In *International Conference on Ad Hoc Networks*. Springer, 2015.
[52] G. Yuan, D. Mazières, and M. Zaharia. Extricating iot devices from vendor infrastructure with karl. *https://arxiv.org/abs/2204.13737*, 2022.
[53] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur. Autotap: synthesizing and repairing trigger-action programs using ltl properties. In *Proc. IEEE/ACM ICSE*.