dSpace
Composable Abstractions for Smart Spaces

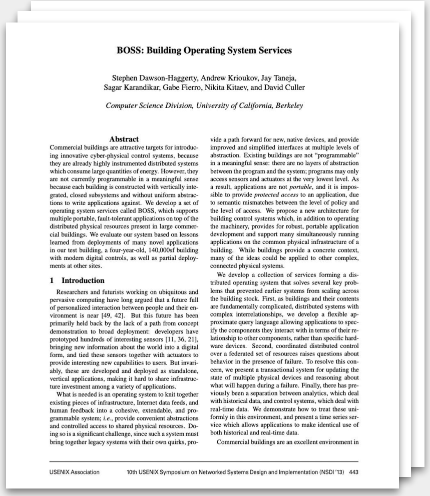
Silvery Fu,
Sylvia Ratnasamy

40+ example providers

3x IoT devices, 2015-2020

13.6 per person by 2023

Still complex to program smart spaces!



BOSS: Building Operating System Services

Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja,
Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler

Computer Science Division, University of California, Berkeley

Abstract

Commercial buildings are attractive targets for introducing innovative cyber-physical control systems, because they are already highly instrumented distributed systems which consume large quantities of energy. However, they are not currently programmable in a meaningful sense because each building is constructed with vertically integrated, closed subsystems and without uniform abstractions to write applications against. We develop a set of operating system services called BOSS, which supports multiple portable, fault-tolerant applications on top of the distributed physical resources present in large commercial buildings. We evaluate our system based on lessons learned from deployments of many novel applications in our test building, a four-year-old, 140,000sf building with modern digital controls, as well as partial deployments at other sites.

1 Introduction

Researchers and futurists working on ubiquitous and pervasive computing have long argued that a future full of personalized interaction between people and their environment is near [49, 42]. But this future has been primarily held back by the lack of a path from concept demonstration to broad deployment: developers have prototyped hundreds of interesting sensors [11, 36, 21], bringing new information about the world into a digital form, and tied these sensors together with actuators to provide interesting new capabilities to users. But invariably, these are developed and deployed as standalone, vertical applications, making it hard to share infrastructure investment among a variety of applications.

What is needed is an operating system to knit together existing pieces of infrastructure, Internet data feeds, and human feedback into a cohesive, extendable, and programmable system; *i.e.*, provide convenient abstractions and controlled access to shared physical resources. Doing so is a significant challenge, since such a system must bring together legacy systems with their own quirks, provide a path forward for new, native devices, and provide improved and simplified interfaces at multiple levels of abstraction. Existing buildings are not "programmable" in a meaningful sense: there are no layers of abstraction between the program and the system; programs may only access sensors and actuators at the very lowest level. As a result, applications are not *portable*, and it is impossible to provide *protected access* to an application, due to semantic mismatches between the level of policy and the level of access. We propose a new architecture for building control systems which, in addition to operating the machinery, provides for robust, portable application development and support many simultaneously running applications on the common physical infrastructure of a building. While buildings provide a concrete context, many of the ideas could be applied to other complex, connected physical systems.

We develop a collection of services forming a distributed operating system that solves several key problems that prevented earlier systems from scaling across the building stock. First, as buildings and their contents are fundamentally complicated, distributed systems with complex interrelationships, we develop a flexible approximate query language allowing applications to specify the components they interact with in terms of their relationship to other components, rather than specific hardware devices. Second, coordinated distributed control over a federated set of resources raises questions about behavior in the presence of failure. To resolve this concern, we present a transactional system for updating the state of multiple physical devices and reasoning about what will happen during a failure. Finally, there has previously been a separation between analytics, which deal with historical data, and control systems, which deal with real-time data. We demonstrate how to treat these uniformly in this environment, and present a time series service which allows applications to make identical use of both historical and real-time data.

Commercial buildings are an excellent environment in

USENIX Association 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13) 443

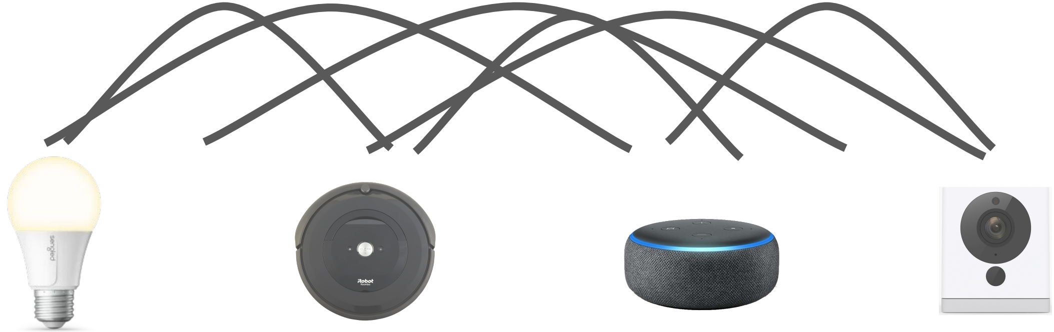
Programming Smart Spaces

Today: why it's complex?

device-centric APIs

monolithic implementations

if-then-that policies



"The basic technique we have for managing the complexity of software is modularity" - Barbara Liskov

"The basic technique we have for managing the complexity of software is modularity" - Barbara Liskov

For smart spaces:

What is the right modularity?

What is the <u>minimal</u> set of abstractions to achieve it?

dSpace:

Modules

A lamp /
smart light bulb

dSpace:

Modules

Digivice

Digivice

dSpace:

Modules

Digivice

Model

D.mod.power
.intent: **on**
.status: **on**

Driver

D.drv():
set(lamp, .intent)
.status = get(lamp)

Digivice

Each digivice D has:

Model - *D.mod*: attribute-values that capture D's intended states (intent) current states (status), and events

Driver - *D.drv()*: code that **reconcile** status to intent

dSpace:

Modules

Digivice

Model

D.mod

Driver

D.drv()

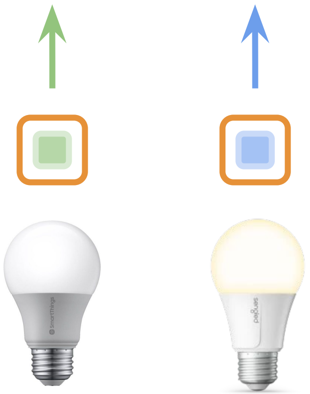
Digivice L1

L1.mod.**power**
.intent: on
.status: off

Digivice L2

L2.mod.**turn_on**
.intent: true
.status: false

Digivices can have different
device libraries (driver)
programming lang. (driver)
schema (model)



Heterogeneity → Complexity

Idea: use a **universal** digivice to configure a **device-specific** digivice

dSpace:

Modules

Digivice

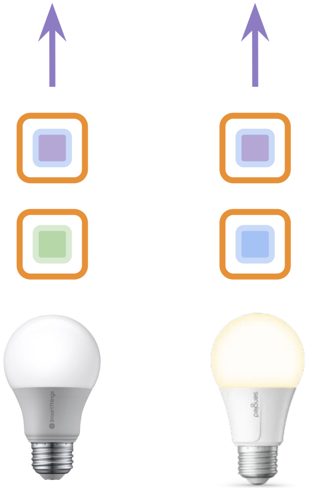
Model

D.mod.**power**
.intent: on
.status: off

Driver

D.drv():
switch vendor
case **L1**: ...
case **L2**: ...

Universal Lamps



Heterogeneity → Complexity

Idea: use a **universal** digivice to configure a **device-specific** digivice

Compose Digivices with Mount primitive

dSpace:

Modules

Digivice

Composition

Mount

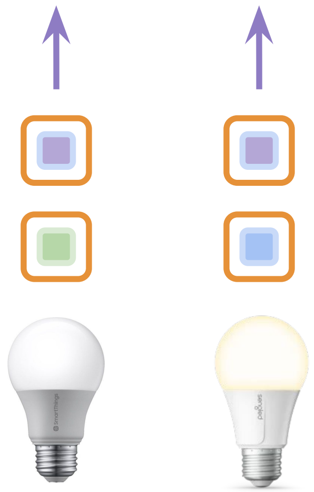
Model

D.mod.power
.intent: on
.status: off

Driver

D.drv():
switch vendor
case L1: ...
case L2: ...

Universal Lamps



Heterogeneity → Complexity

Idea: use a **universal** digivice to configure a **device-specific** digivice

dSpace:

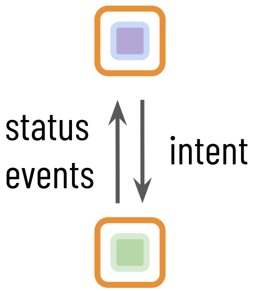
Modules

Digivice

Composition

Mount

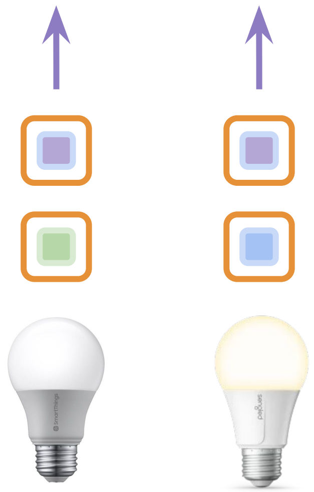
Parent



Child

Compose Digivices with Mount primitive

Universal Lamps



Mount(A, B) allows B.drv() to:

1. Write to A.mod.intent
2. Read from A.mod.status

B: parent; A: child

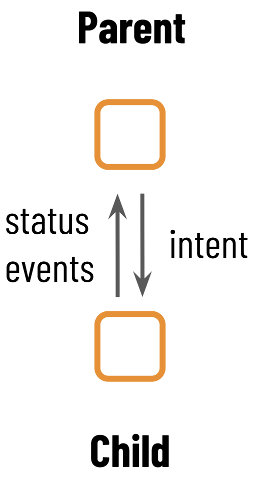
dSpace:

Modules

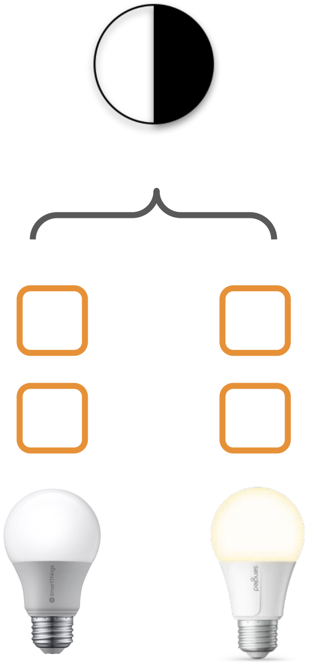
Digivice

Composition

Mount



Living Room



Aggregate brightness of the living room

dSpace:

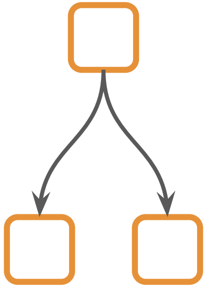
Modules

Digivice

Composition

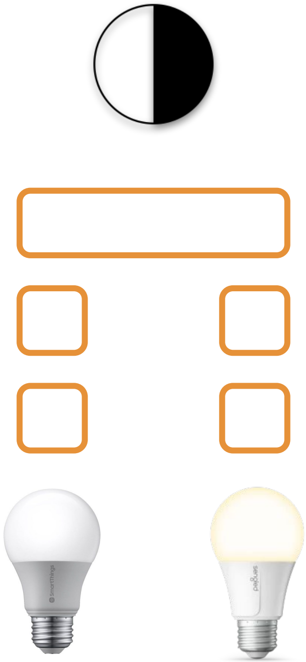
Mount

Parent



Child **Child**

Living Room



Aggregate brightness of the living room

Idea: Introducing a living room digivice and mount both lamps!

dSpace:

Modules

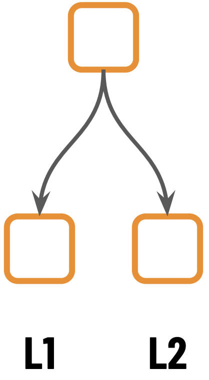
Digivice

Composition

Mount

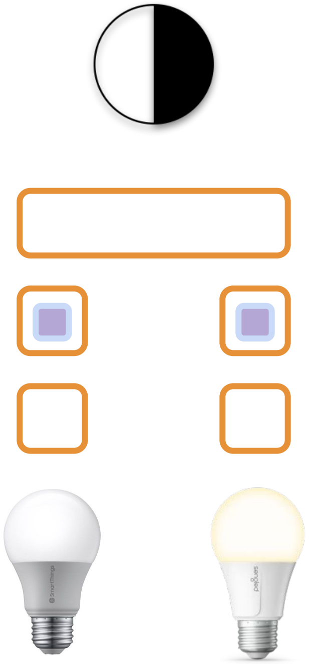
Room.mod.bri
.intent: 0.8
.status: 0.8

Room



L1.mod.bri
.intent: 0.4
.status: 0.4

Living Room



Developers of the room:

Don't interact with physical devices

Program universal lamps

Idea: Introducing a living room digivice and mount both lamps!

dSpace:

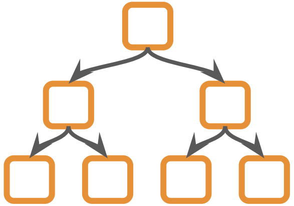
Modules

Digivice

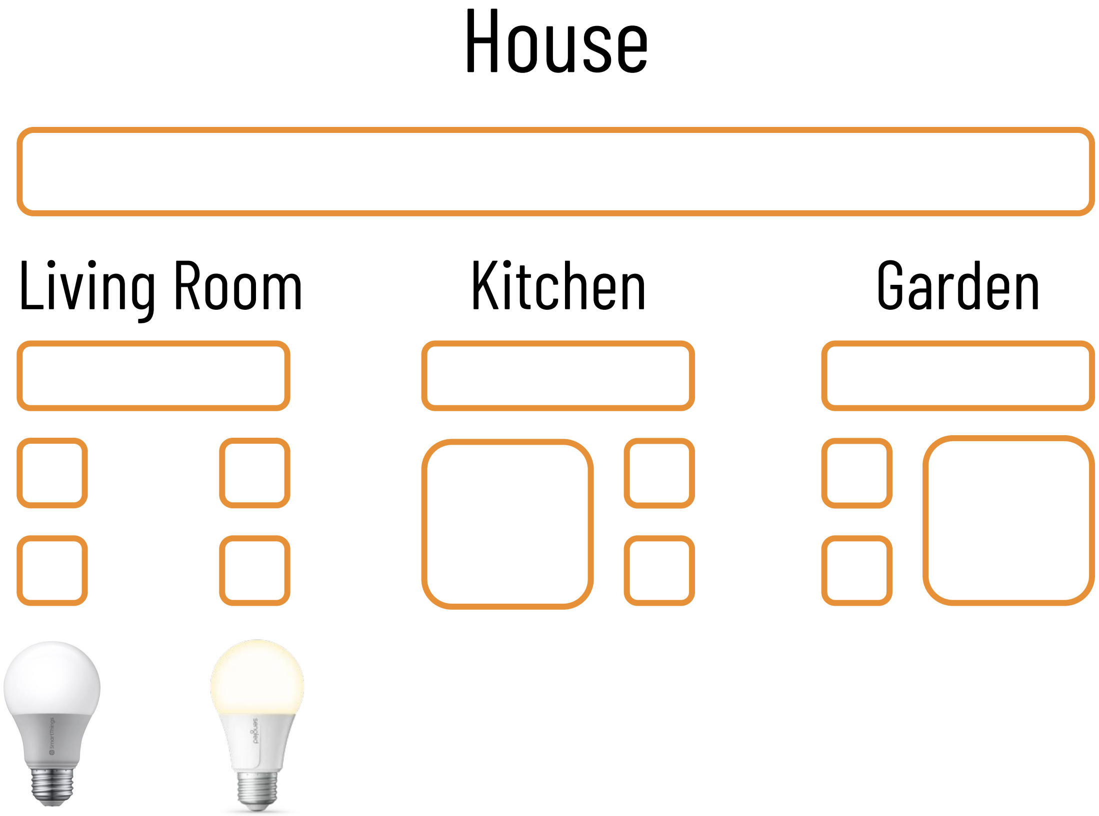
Composition

Mount

Digivices form **control hierarchy**



Raising the level of abstractions



dSpace:

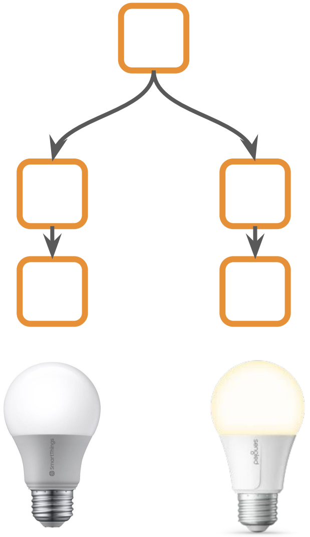
Modules

Digivice

Composition

Mount

Living Room



dSpace:

Modules

Digivice

Composition

Mount

Goal: integrate data processing with digivices

Video processing

ML/object recognition

Living Room

dSpace:

Modules

Digivice

Digidata

Composition

Mount

Digidata

Model

CV.mod
.in: rtsp://..
.out: human

Driver

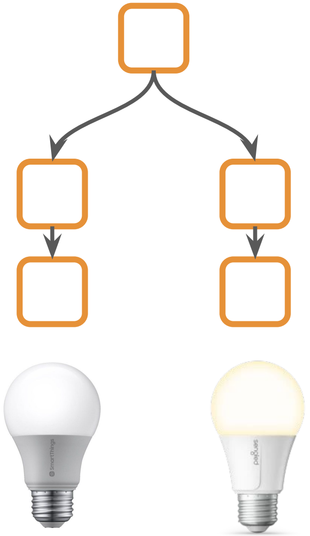
CV.drv():
frame = capture(.in)
.out = detect(frame)

Goal: integrate data processing with digivices

Video processing — FFMPEG

ML/object recognition — OpenCV

Living Room



Each **digidata T** has:

T.mod.in: data input
T.mod.out: data output

Driver *T.drv():* data processing code to transform T.mod.in to T.mod.out

dSpace:

Modules

Digivice

Digidata

Composition

Mount

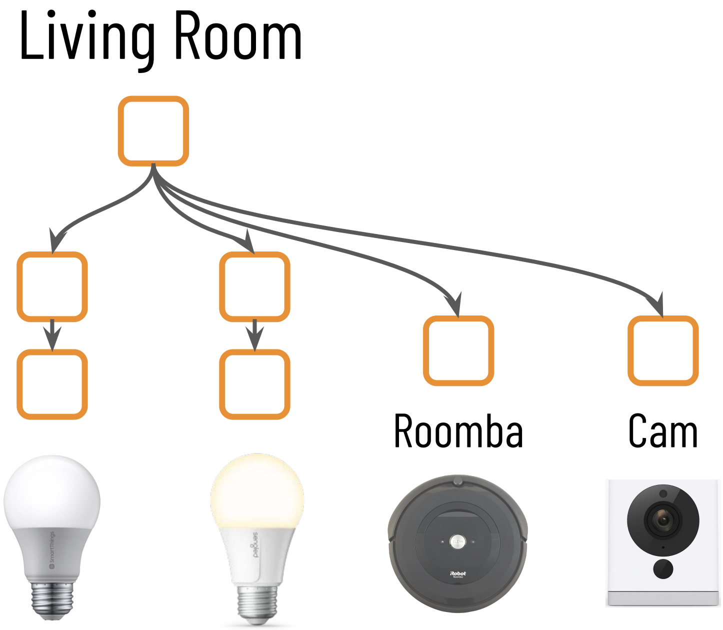
Digidata

Model

CV.mod
.in: rtsp://..
.out: human

Driver

CV.drv():
frame = capture(.in)
.out = detect(frame)



dSpace:

Modules

Digivice

Digidata

Composition

Mount

Digidata

Model

CV.mod
.in: rtsp://..
.out: human

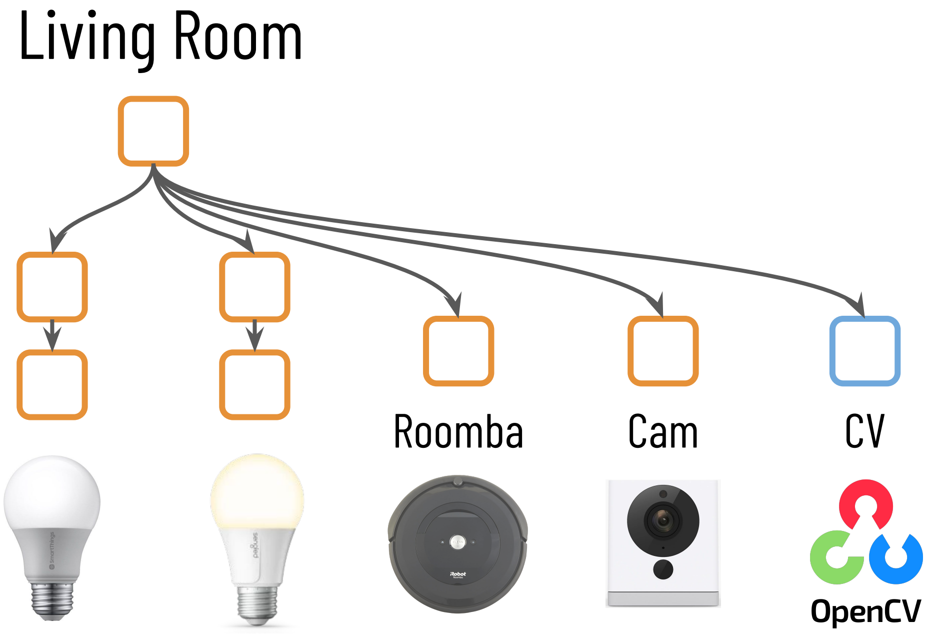
Driver

CV.drv():
frame = capture(.in)
.out = detect(frame)

Mounting digidata T to digivice D allows:

D to write T.mod.in

D to read T.mod.out



Digidata

Model

CV.mod
.in: rtsp://..
.out: human

Driver

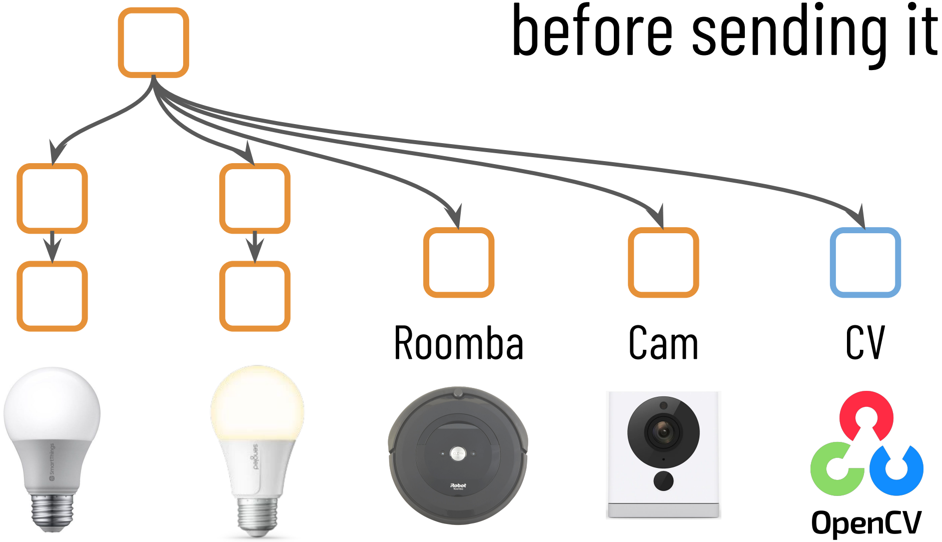
CV.drv():
frame = capture(.in)
.out = detect(frame)

Mounting digidata T to digivice D allows:

D to write T.mod.in

D to read T.mod.out

Living Room



Process the video before sending it to CV?

dSpace:

Modules

Digivice

Digidata

Composition

Mount

Pipe

Digidata

Model

CV.mod
.in: rtsp://..
.out: human

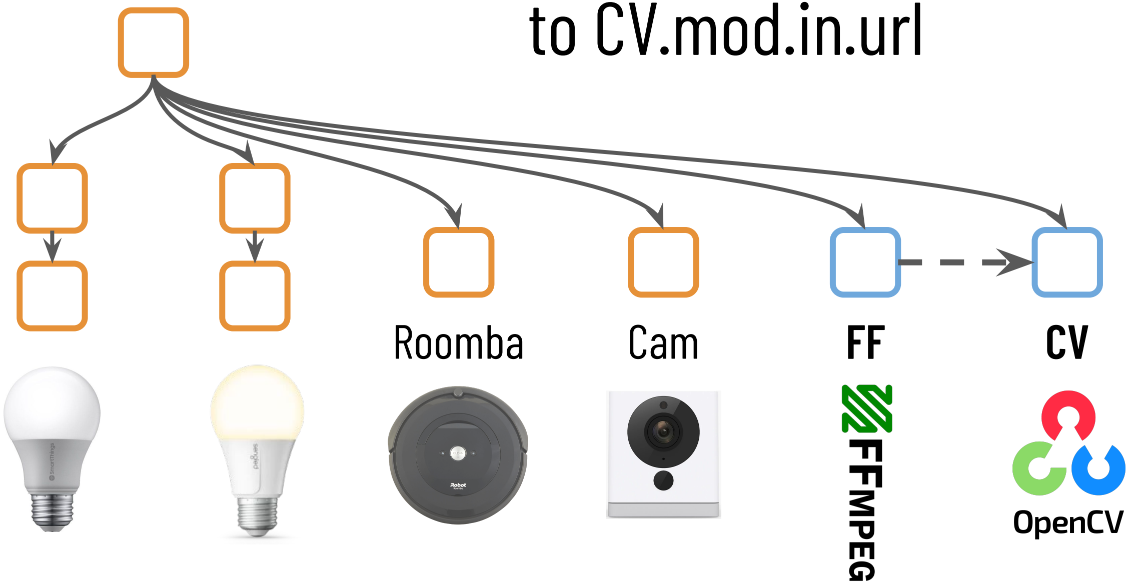
Driver

CV.drv():
frame = capture(.in)
.out = detect(frame)

Pipe(A, B) writes A.mod.out to B.mod.in

Living Room

Write FF.mod.out.url
to CV.mod.in.url



dSpace:

Modules

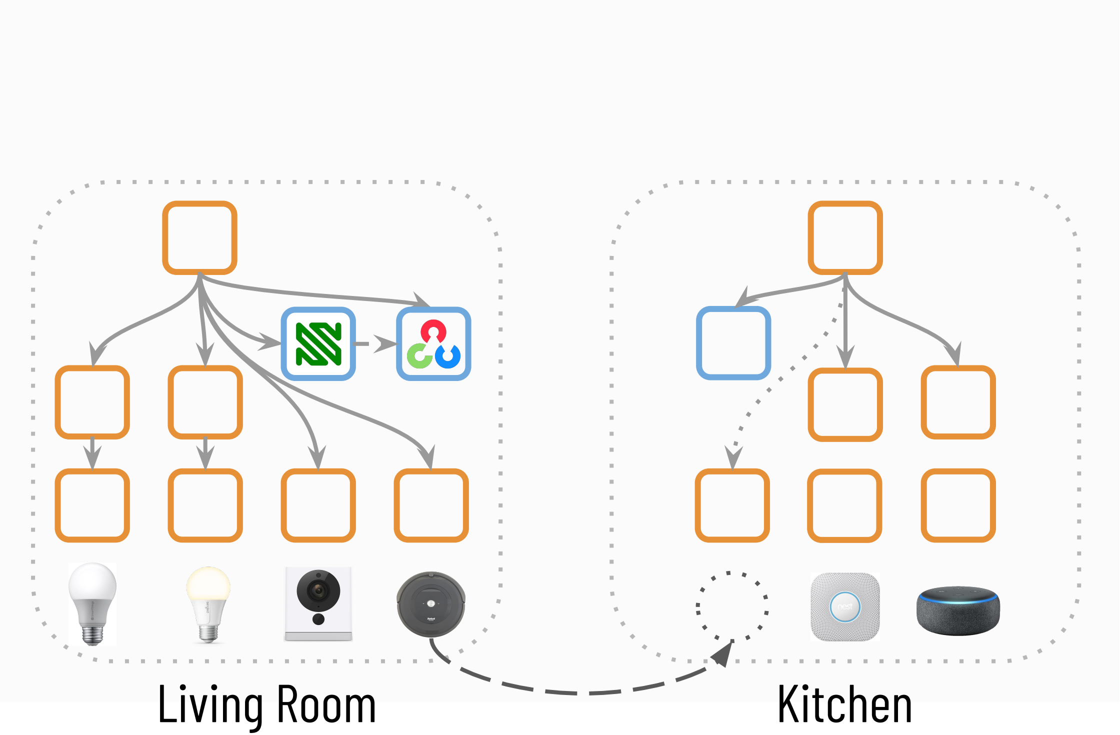
Digivice

Digidata

Composition

Mount

Pipe



Device moves across rooms

dSpace:

Modules

Digivice

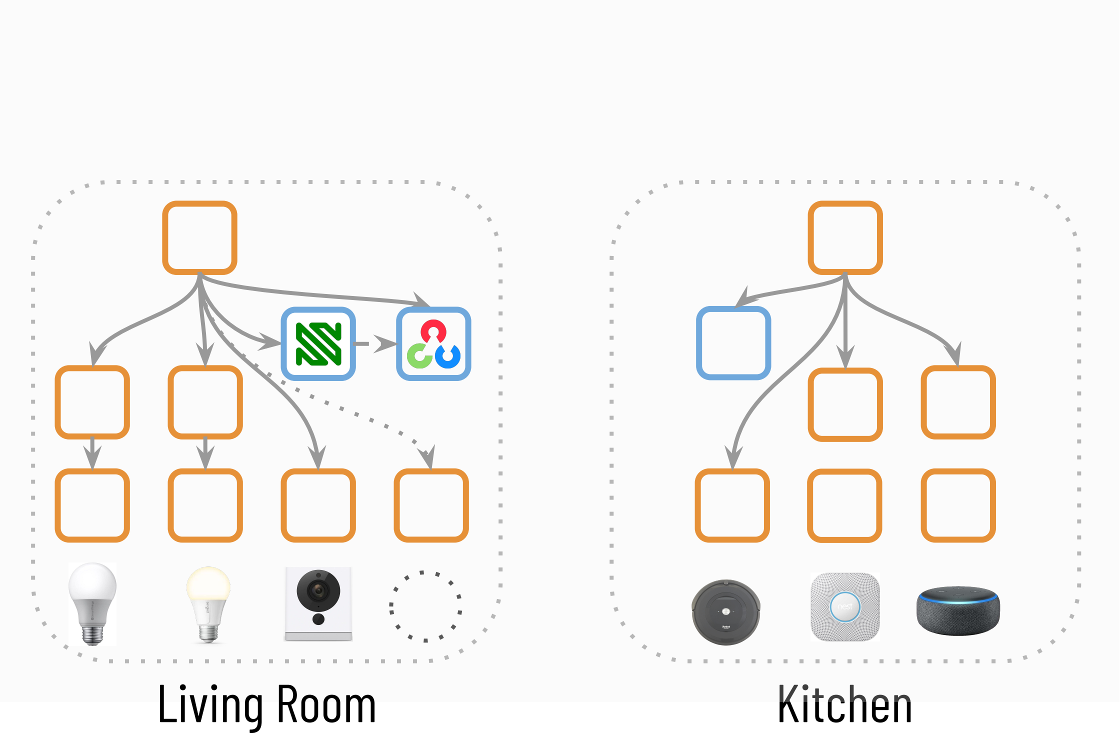
Digidata

Composition

Mount

Pipe

Yield



Device moves across rooms

Allow multiple parents, but only one can have write-access and the other(s) are **yielded**

dSpace:

Modules

Digivice

Digidata

Composition

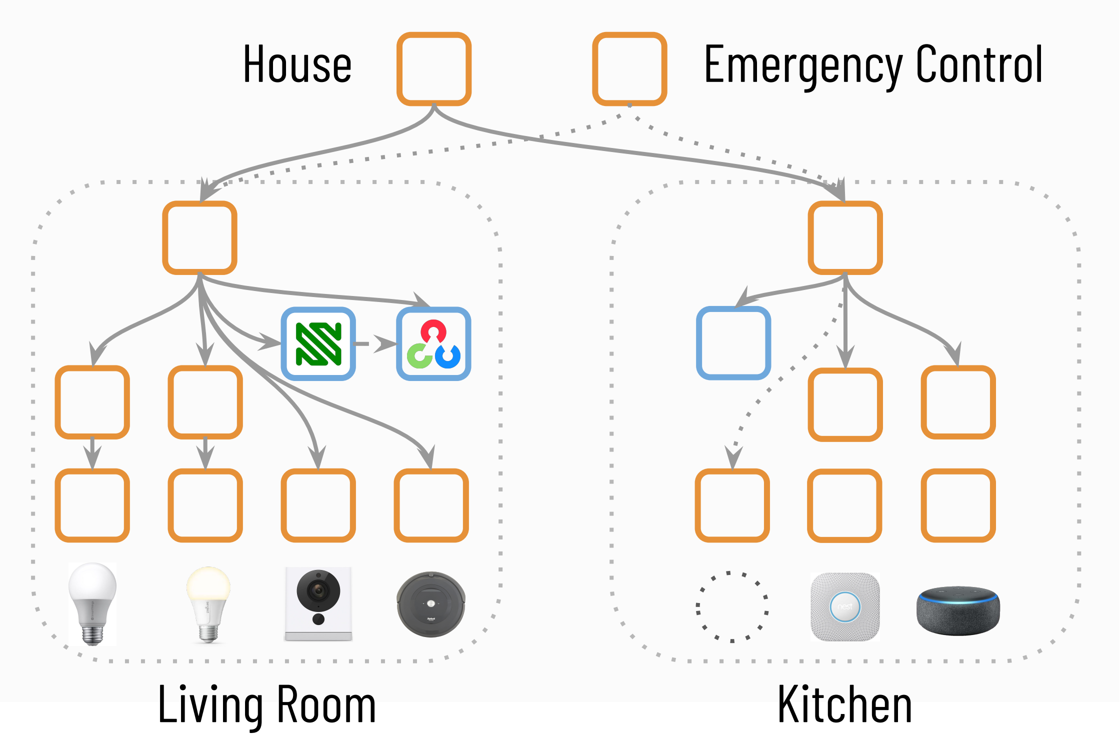
Mount

Pipe

Yield

Policies

Delegation



Device moves across rooms

Delegate access to third-party

Allow multiple parents, but only one can have write-access and the other(s) are **yielded**

Must preserve **multi-tree** topology to avoid loops

dSpace:

Modules

Digivice

Digidata

Composition

Mount

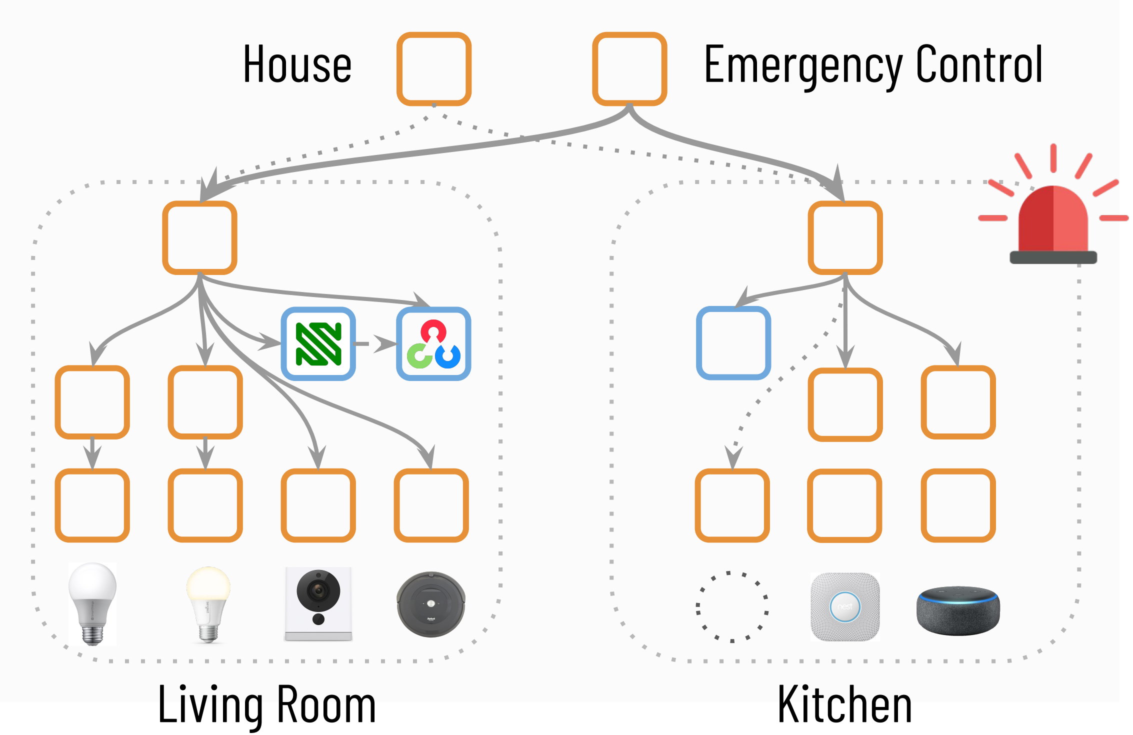
Pipe

Yield

Policies

Delegation

Intent reconc.



Device moves across rooms

Delegate access to third-party

Handle intent conflicts due to physical events

Allow multiple parents, but only one can have write-access and the other(s) are **yielded**

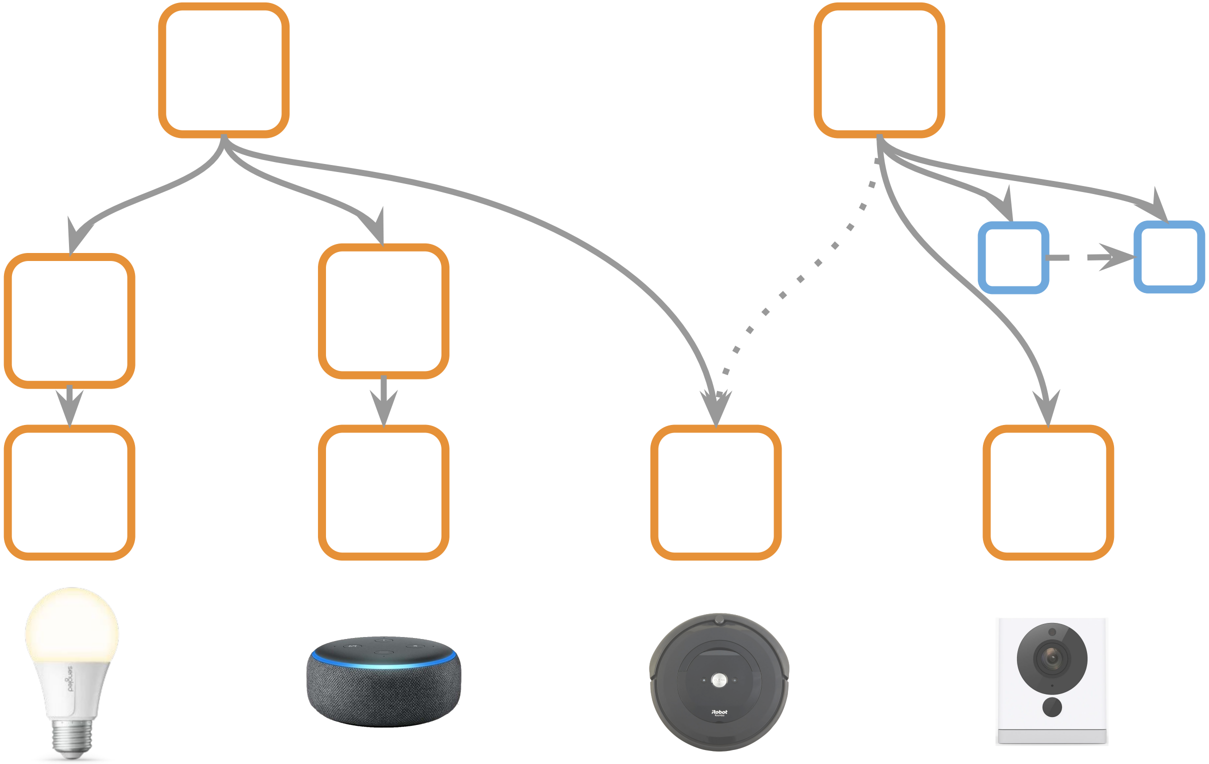
Must preserve **multi-tree** topology to avoid loops

More details about policies in the paper!

Recap: Programming Smart Spaces

What makes dSpace simple?

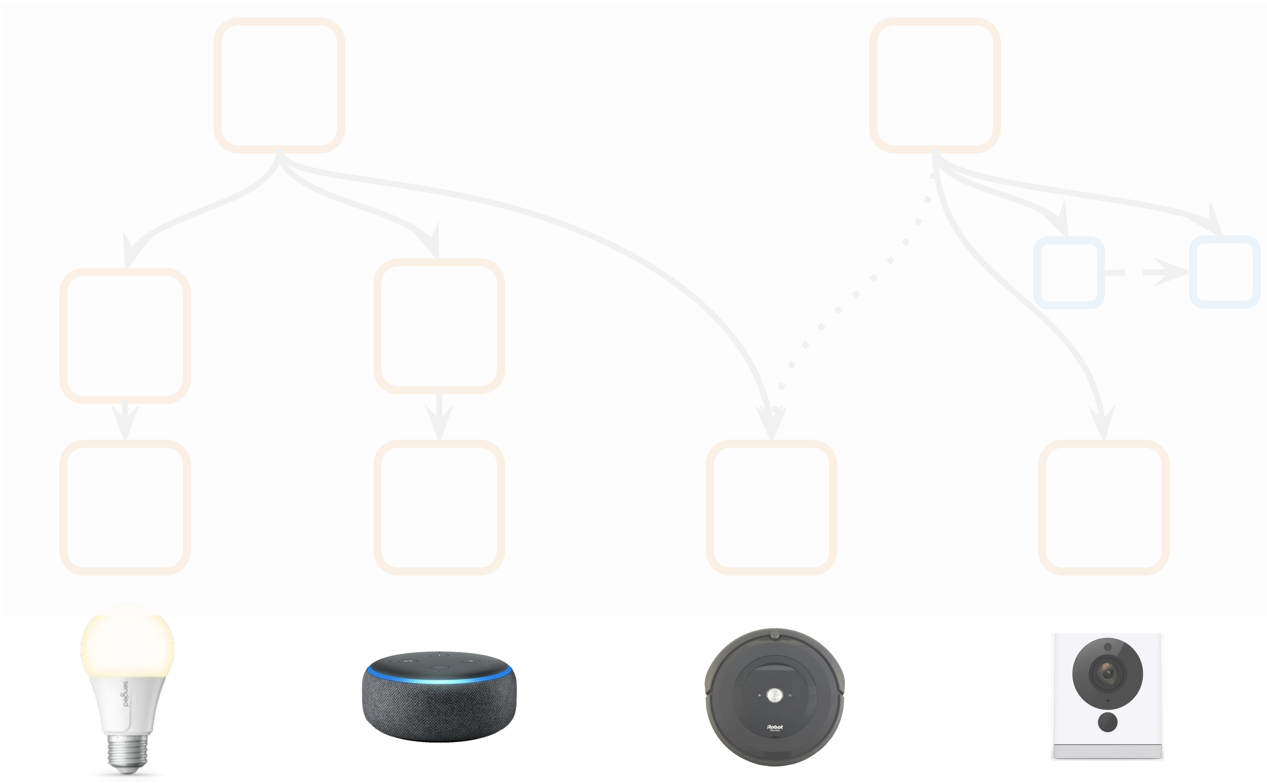
Today:	dSpace:
Device-centric abstractions	Composable digivice/data
Ad-hoc composition: **If-then-that**	First-class primitives: **Mount Pipe Yield**
Limited HL abstraction and policies	Rich policies: **delegation** **intent reconc.**
Monolithic architecture	



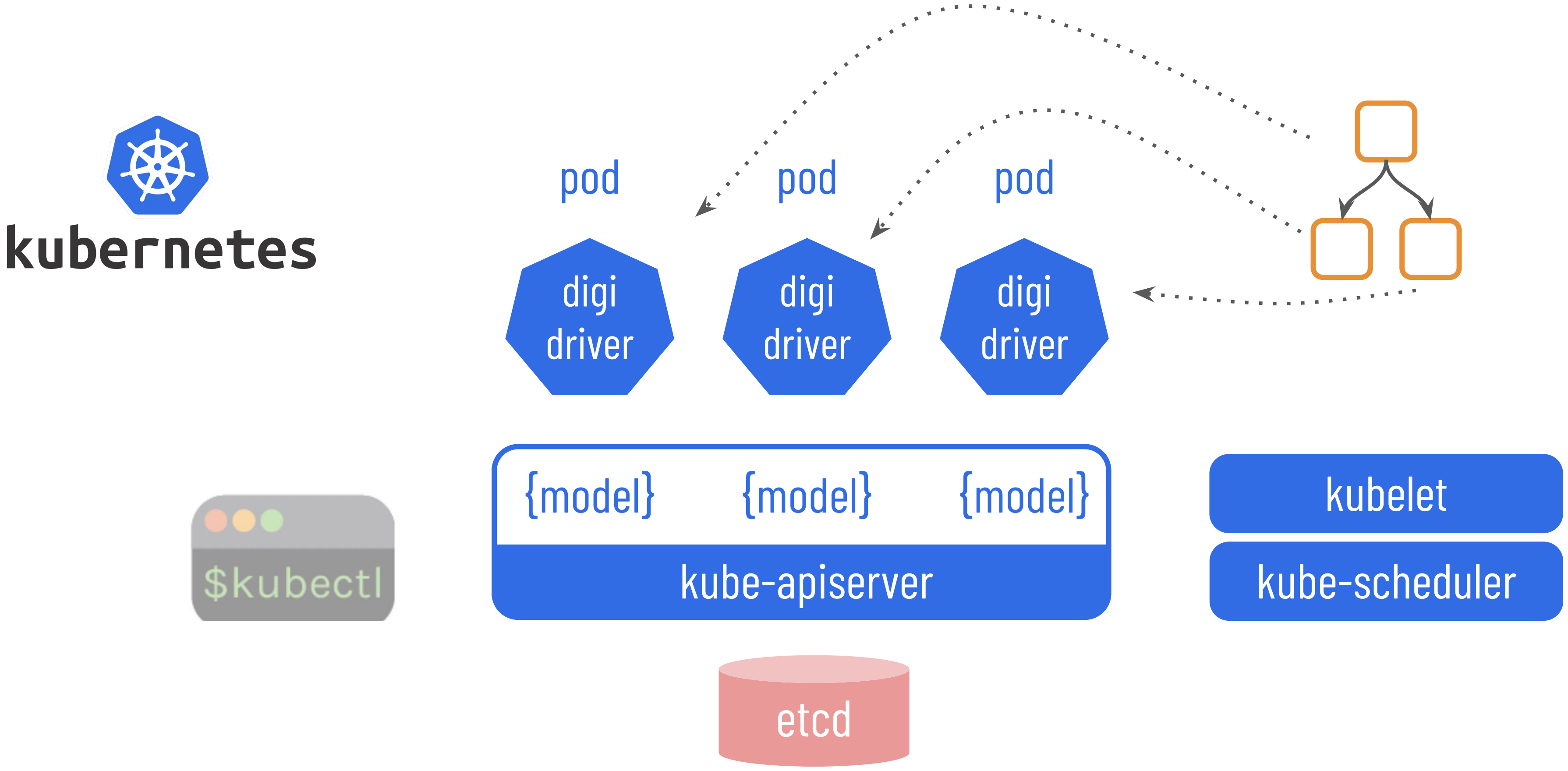
Recap: Programming Smart Spaces

What makes dSpace simple?

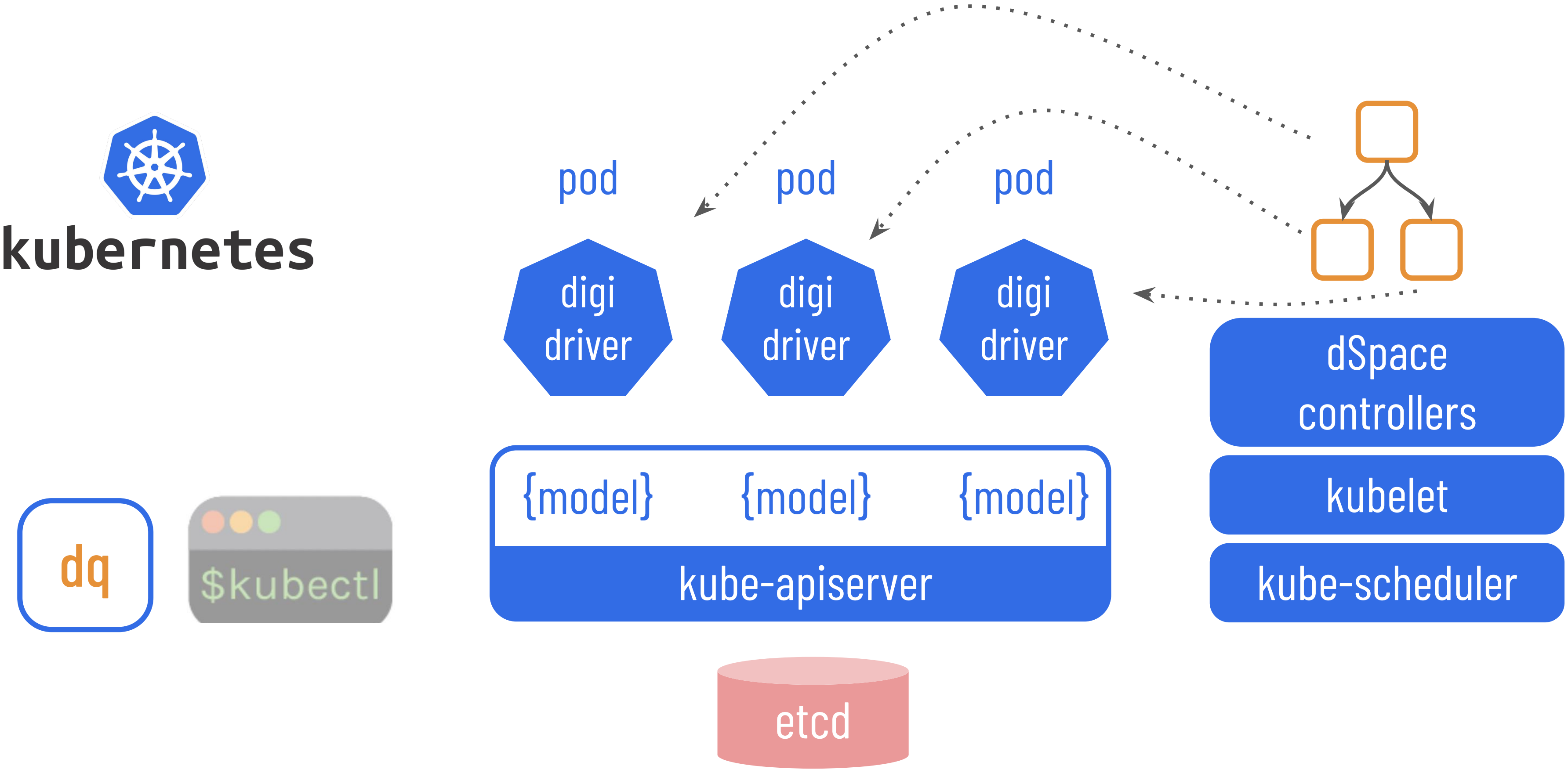
Today:	dSpace:
Device-centric abstractions	Composable digivice/data
Ad-hoc composition: **If-then-that**	First-class primitives: **Mount Pipe Yield**
Limited HL abstraction and policies	Rich policies: **delegation intent reconc.**
Monolithic architecture	Digis run as **microservices**



Implement dSpace with Microservices



Implement dSpace with Microservices



See the paper for details:

Design and Implementation

Design Principles, Driver Programming, Runtime Arch., Security etc.

Evaluation

10 scenarios in smart home with 9 devices

< *300* lines of code (LoC; *+15%*) for all scenarios

vs. existing frameworks

***4/10* scenarios are dSpace only, the rest more (4x) LoC**

User study and performance benchmarks

***4.41* MOS (0-5); runtime adds *<20%* latency overhead**

Goal: simplify development of smart space apps

Manage complexity through the (right) modularity:

Digivice, Digidata + Mount, Pipe, Yield

Thank you!

github.com/NetSys/dspace