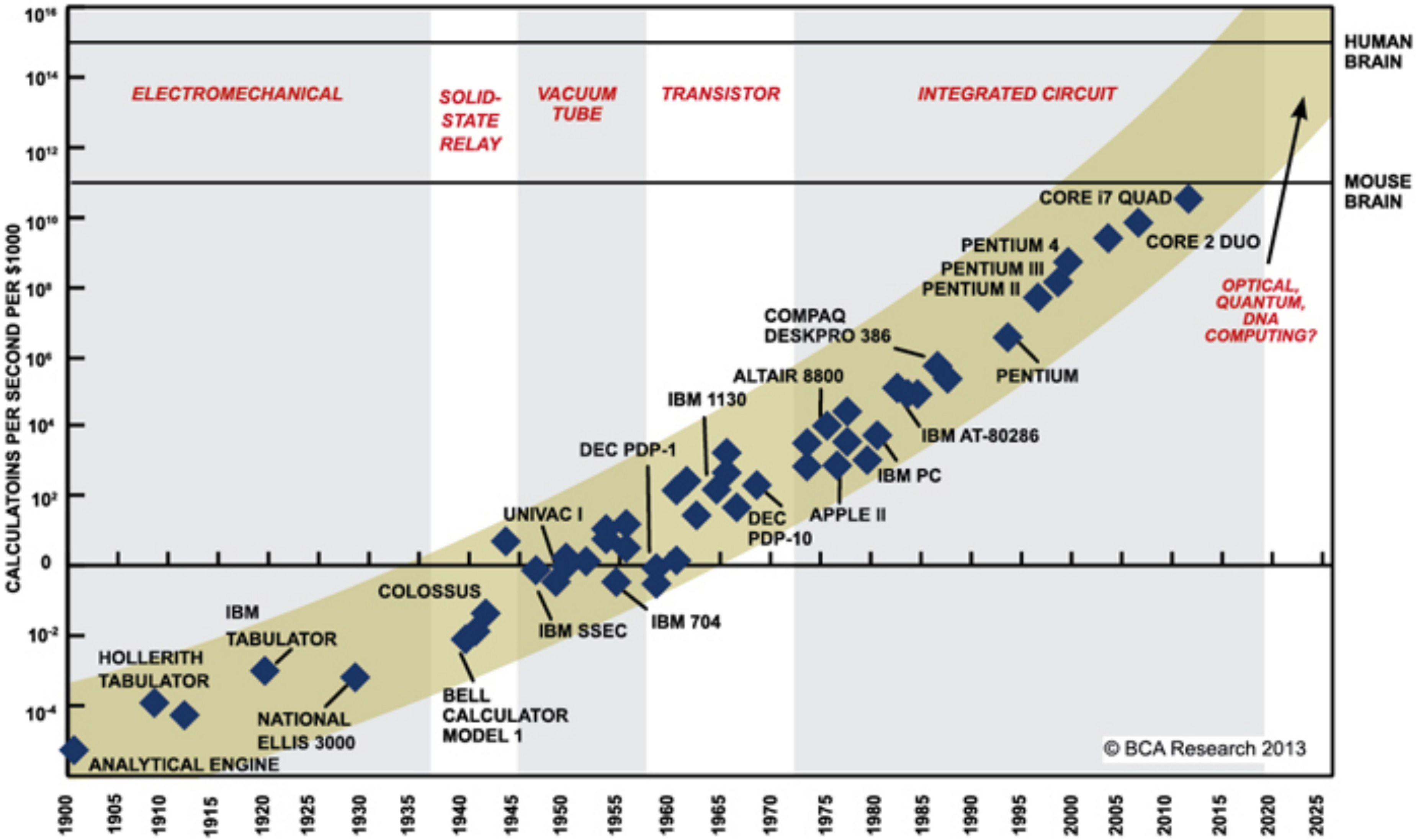
Programming IoT

Rupak Majumdar

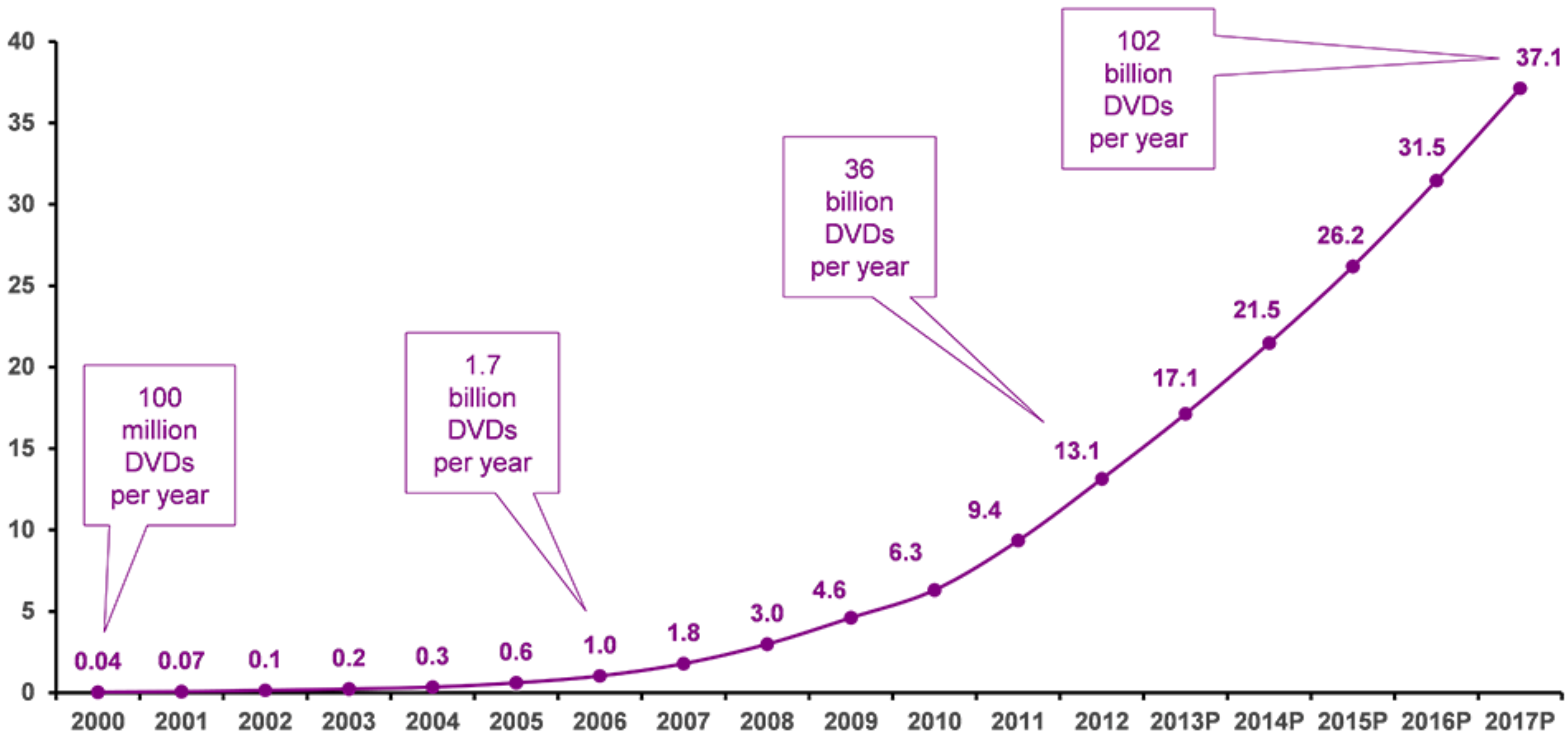
Max Planck Institute for Software Systems
Kaiserslautern, Germany

(Joint work with Jeff Fischer)

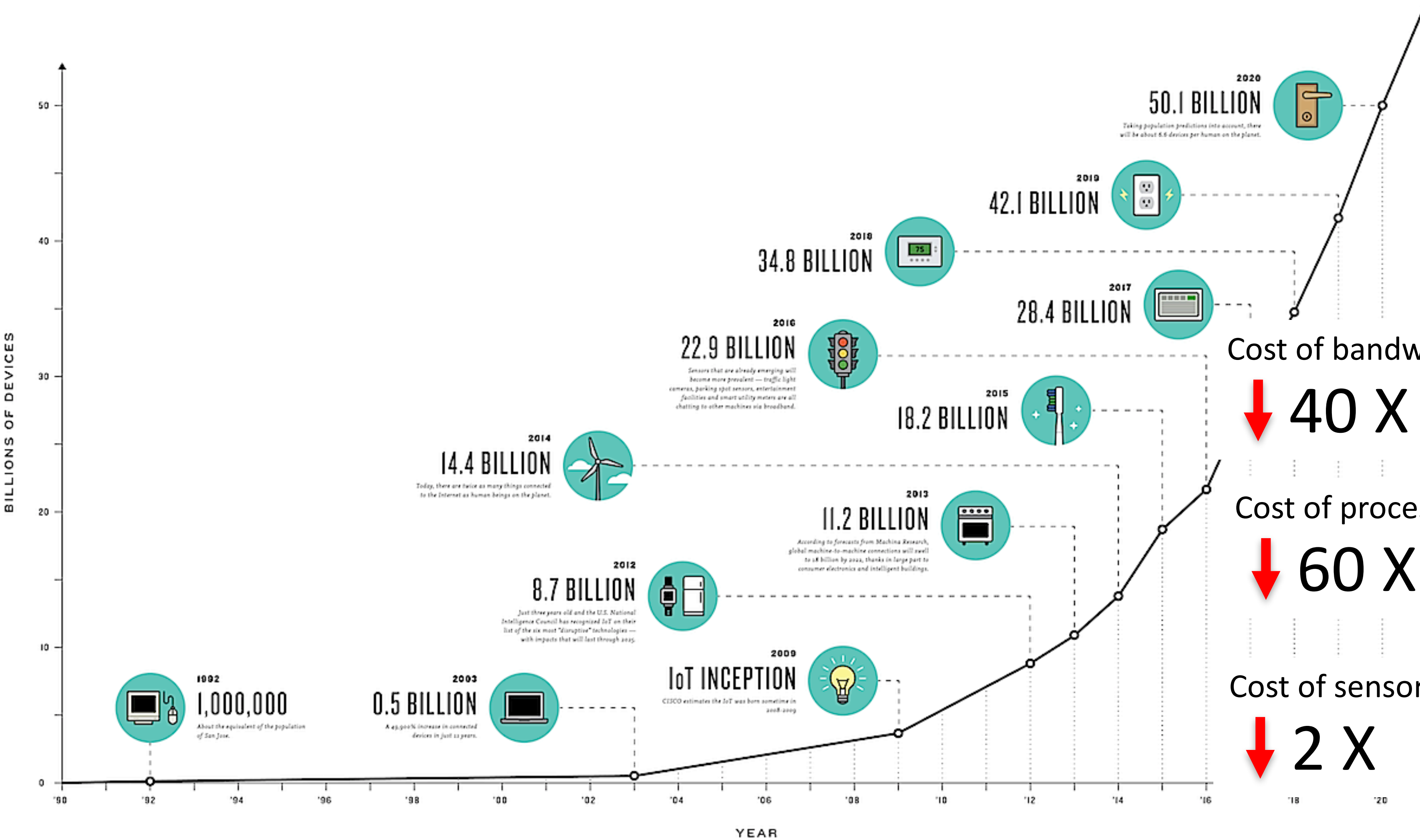


SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Estimated U.S. Internet Protocol Traffic, 2000-2017 (Exabytes per Month)



Source: Cisco Visual Networking Index (VNI) and USTelecom Analysis. A DVD is assumed to store a two-hour movie.



Cost of bandwidth

40 X

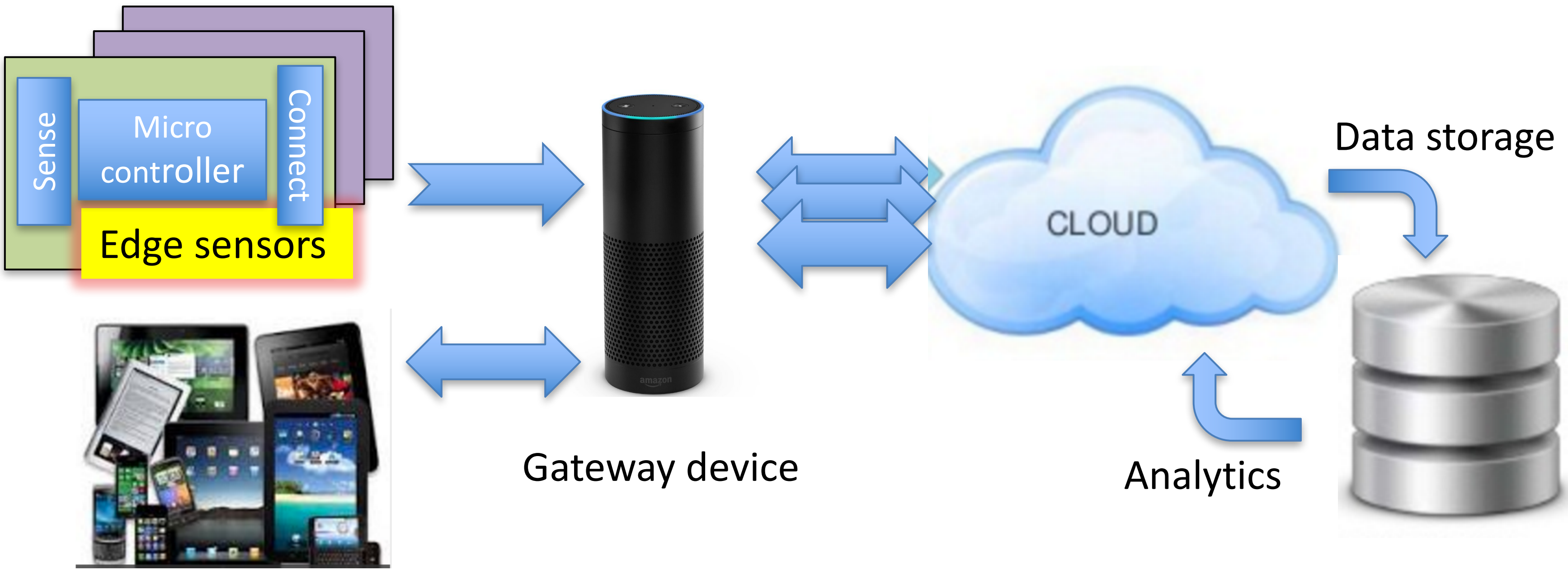
Cost of processing

60 X

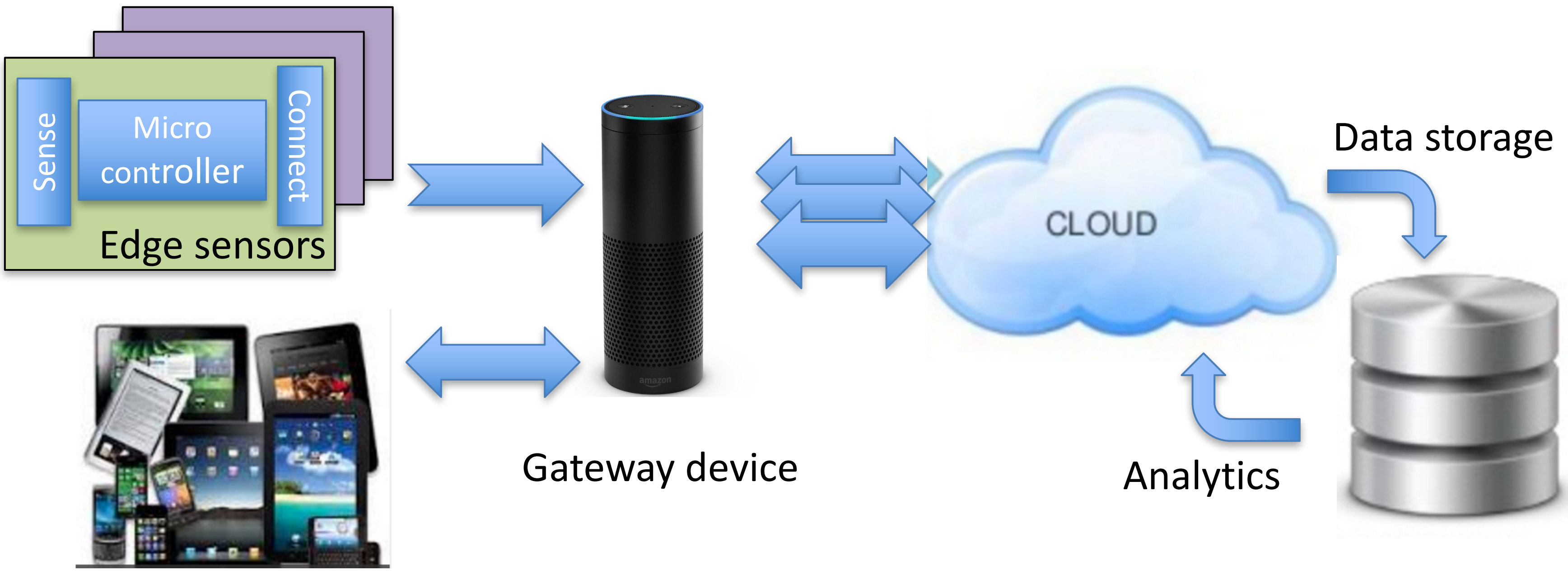
Cost of sensors

2 X

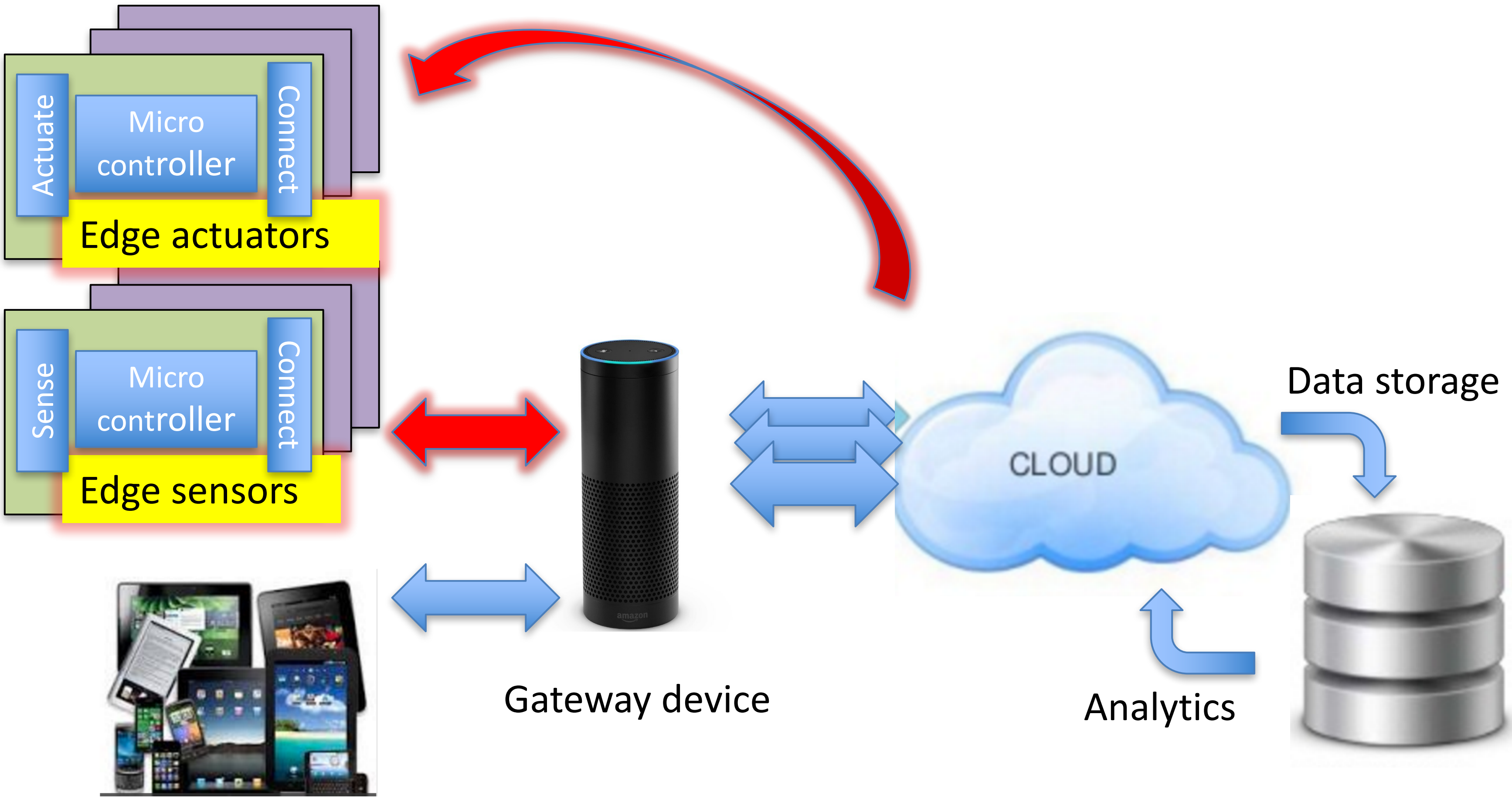
Source: Gartner, IDC



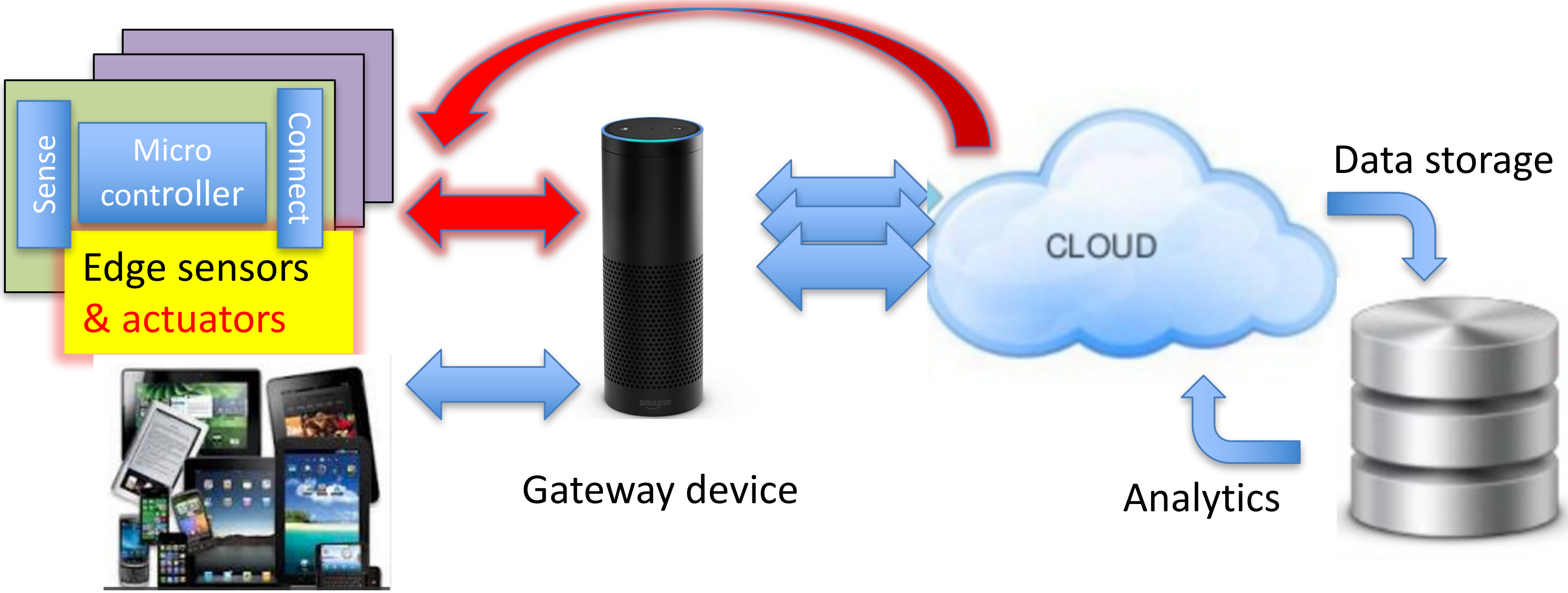
Examples: Consumer analytics, real-time sensing and monitoring



Information and Analytics		
1. Consumer analytics	2. Sensor-driven decision making	3. Real-time monitoring
Monitoring and profiling user behavior on the Internet Learning user models for targeted ads	Analytics for business intelligence	Monitoring the behaviors of persons, things, or data through space and time
Examples: Clickthrough analysis Location-aware recommendations	*Examples:* Smart factories Production trends Computation & storage trends	*Examples:* Inventory and supply chain management Security analytics

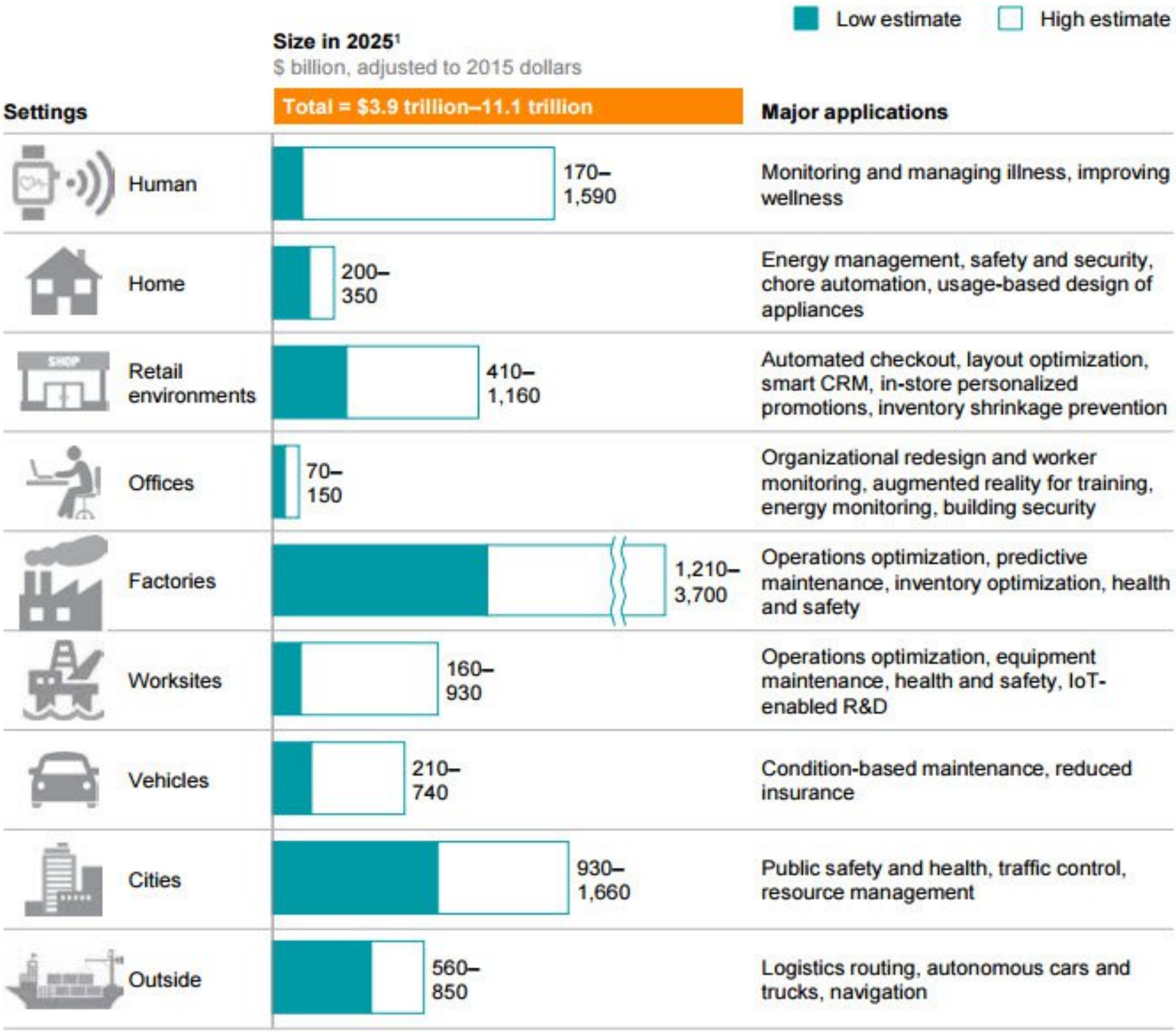


Examples: Process automation, Closed loop decision making, Complex autonomous processes



Automation and control		
1. Process automation	2. Closed-loop decision making	3. Complex autonomous systems
Controlling the behaviors of persons, things, or data through space and time	Feedback control of consumption for resources	Automatic control in open and uncertain environments
Examples: Software-based process control Smart factories	*Examples:* Networked smart energy management Smart buildings Health monitoring	*Examples:* Autonomous cars & traffic networks Robotic swarms, disaster management

Potential economic impact of IoT in 2025, including consumer surplus, is $3.9 trillion to $11.1 trillion



1 Includes sized applications only.
NOTE: Numbers may not sum due to rounding.

SOURCE: McKinsey Global Institute analysis

This Talk:

Programming abstractions, Models, and Analyses for developing Large-scale IoT Systems

Part I: A language abstraction

Part II: Some verification problems

Dynamics Level	Modeling the world: ODEs, Uncertainty	“Classical” control and signal processing: AD converters, PID controllers

Programming Environment

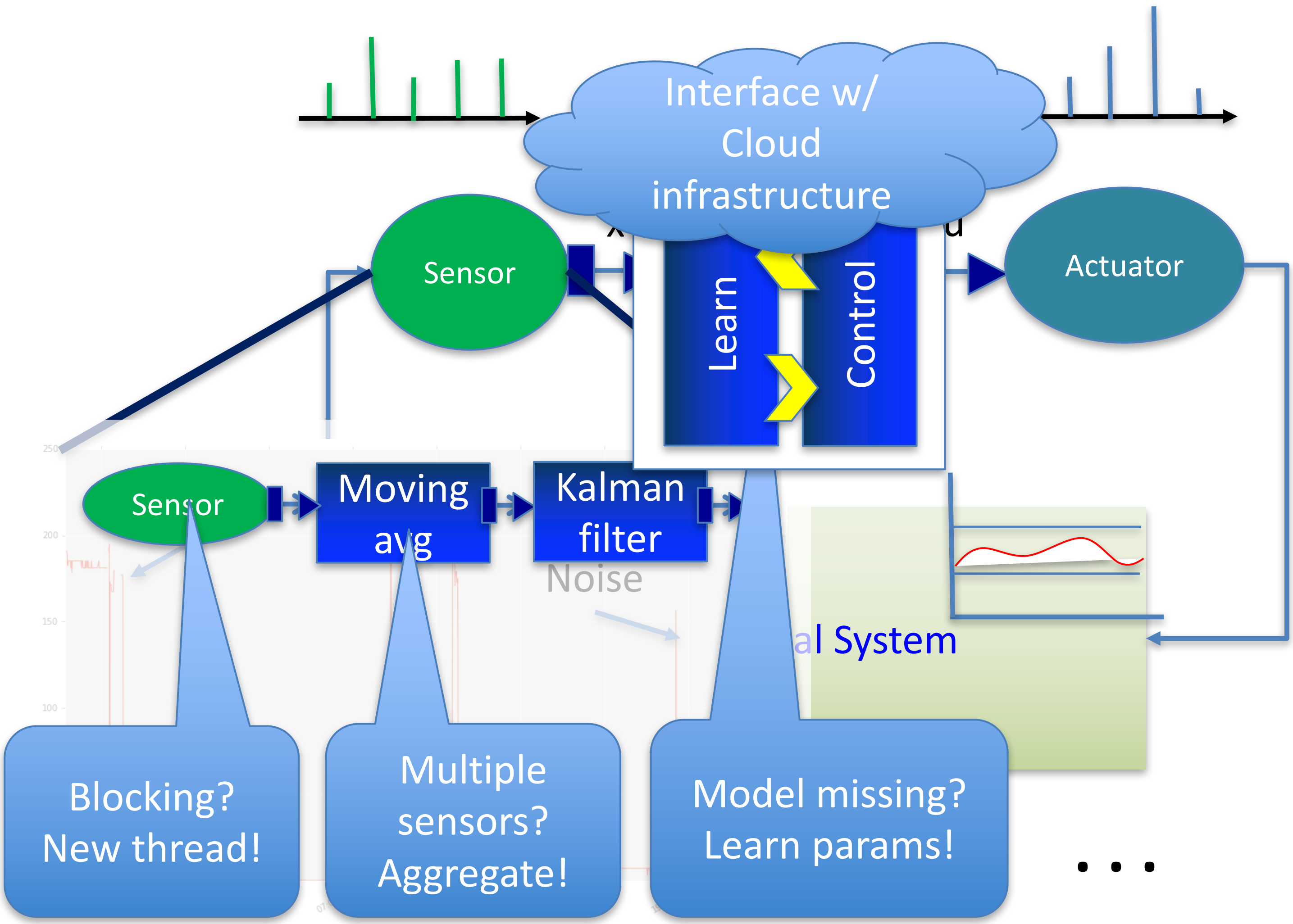
1. *Streams* and *stateful transformations* of streams
2. *Asynchronous* concurrency, real-time
3. *Uncertainty* as “first-class” object
4. *Heterogeneous* computing platforms
5. *Distributed* infrastructure

Domain-Specific Languages

- Control: Simulink/Stateflow
- Synchronous hardware: Esterel/Lustre
- Systems & Networking: Click
- Data processing: Apache Spark Streaming

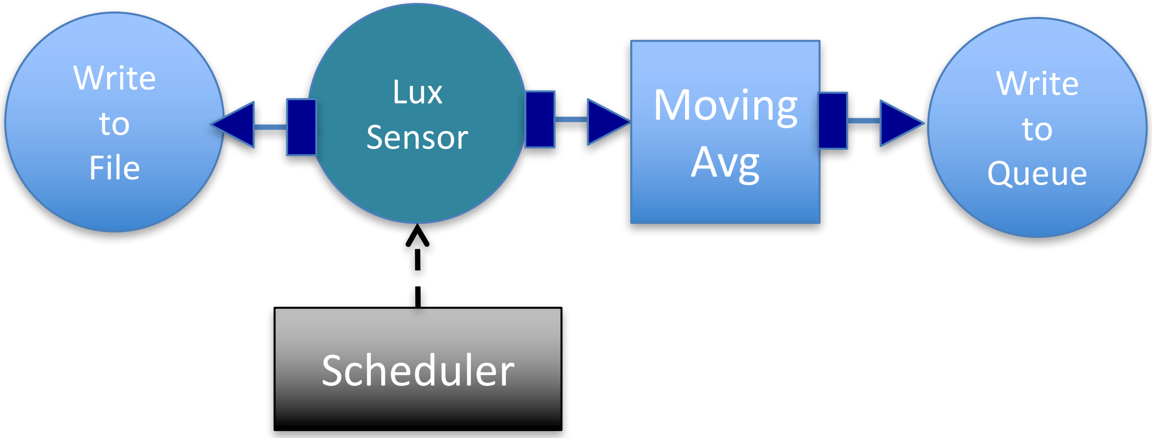
- This Talk: *ThingFlow*, a DSL for IoT

Example: A Temperature Controller



Simple ThingFlow Example

- Periodically sample a light sensor
- Write the sensed value to a file
- Every 5 steps, send the moving average to a message queue




```
sensor.connect(file_writer('file'))
sensor.transduce(MovingAvg(5)).connect(mqtt_writer)
```

“Traditional” Event-driven Style (Callbacks)

```
def sample_and_process(sensor, mqtt, xducer, compcb, errcb):
  try:
    sample = sensor.sample()
  except StopIteration:
    final_event = xducer.complete()
    if final_event:
      mqtt.send(final_event,
                lambda:
                mqtt.disc(lambda: compcb(False), errcb),
                errcb)
    else:
      mqtt.disconnect(lambda: compcb(False), errcb)
    return
  event = SensorEvent(sensor_id=sensor.sensor_id,
                      ts=time.time(), val=sample)
  csv_writer(event)
  median_event = xducer.step(event)
  if median_event:
    mqtt.send(
  else:
    compcb(Tru

def loop(event
  def compcb(m
    if more:
      event_lo
    else:
      print("al
     event_loo
  def errcb(e)
    print("Got
    event_loop
  event_loop.c
    lambda: sa
```

1. Separate connecting streams with handling of runtime situations: distinct control flows for normal, error, and end-of-stream conditions not required
2. Inversion of control avoided: programmer’s view = data flow in the system
3. Scheduling is provided by the infrastructure

With Coroutines

```
async
def sample_and_process(sensor, mqtt, xducer):
  try:
    sample = sensor.sample()
  except StopIteration:
    final_event = xducer.complete()
    if final_event:
        await mqtt.send(final_event)
    await mqtt.disconnect()
    return False
  event = make_event(sensor.sensor_id, sample)
  csv_writer(event)
  median_event = xducer.step(event)
  if median_event:
    await mqtt.send(median_event)
    return T

def loop(eve
  coro = sa
  task = ev
  def done_c
      exc =
      if exc:
        raise
      elif f.
        prin
        even
      else:
        even
  task.add_
```

1. No more callbacks, but interconnection still mixed with control situations

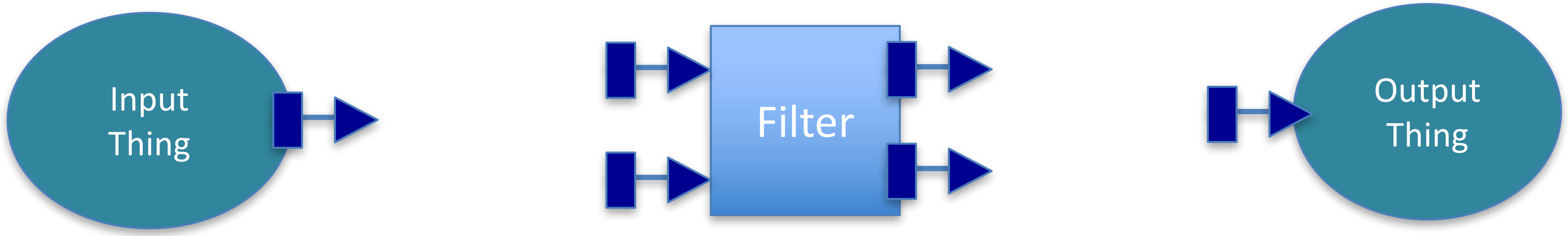
2. Choice to use asynchronous calls propagates through the program: implementation decisions have global effects

ThingFlow Features

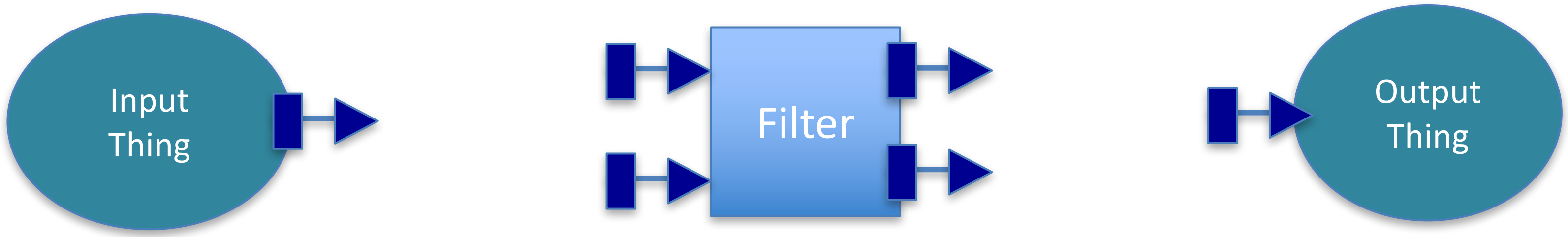
- Streams of “things”
  - Input things introduce streams of events into the system (e.g., sensors)
  - Output things consume streams of events (e.g., actuators)
- Filters =

  Both input and output things =

  Stream transformers

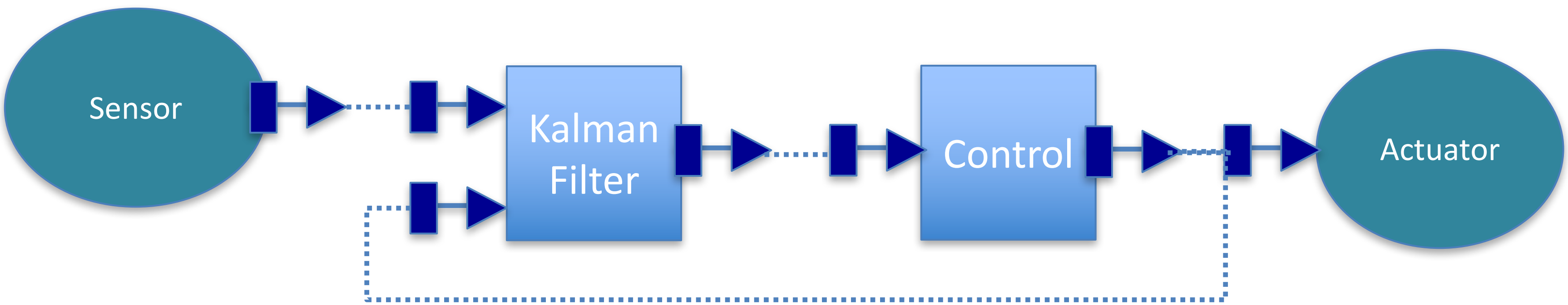


ThingFlow Features



- ThingFlow Programs = Graphs of stream transformers connecting input/output ports
  - Basic construct: A.connect(B, inport=outport)
  - Syntactic sugar: default ports, chaining filters, combinators
    A.map(f) – map the output stream on A using function f
    A.transduce(M) – transduction by machine M
- Asynchronous, push-semantics
  - explicit scheduling

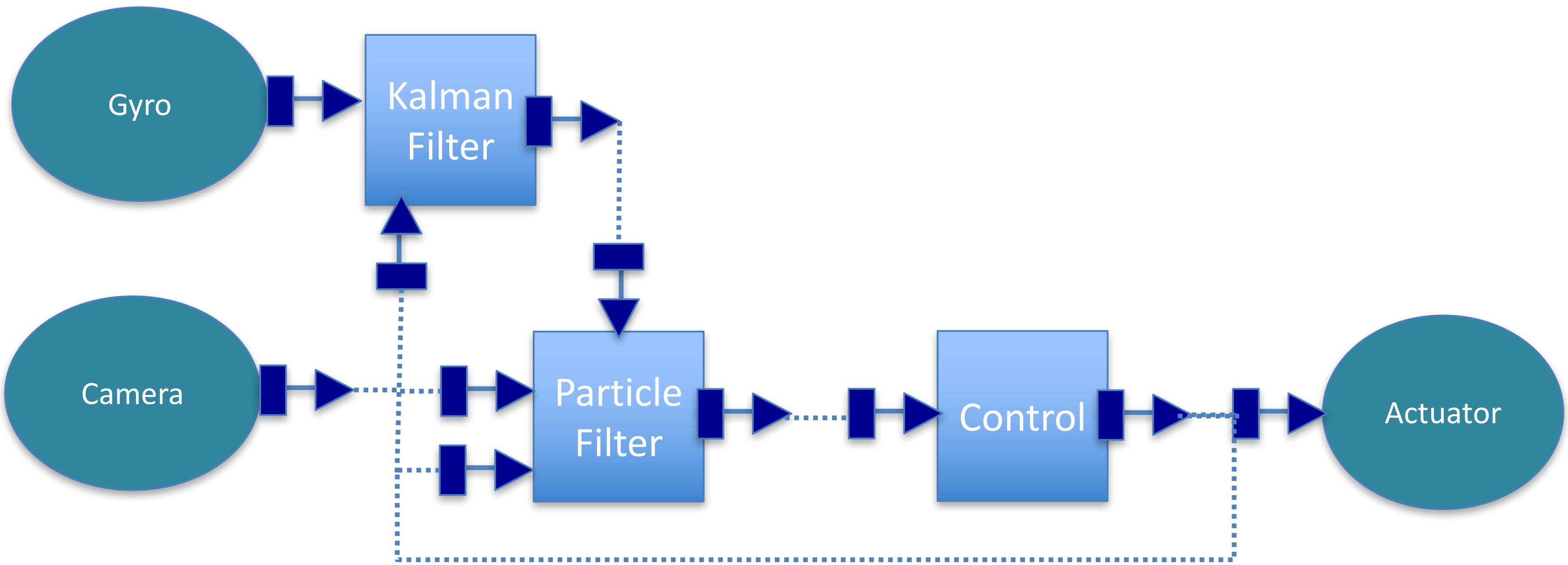
ThingFlow Controller




```
kalman = Kalman(A, B, C, Q, R) # Kalman filter
pid = PID(Kp, Ki, Kd) # PID controller
pid.connect(kalman, port_mapping=('default', 'input'))
_ = Sensor().transduce(kalman)\
            .transduce(pid)\
            .connect(Actuator())
```

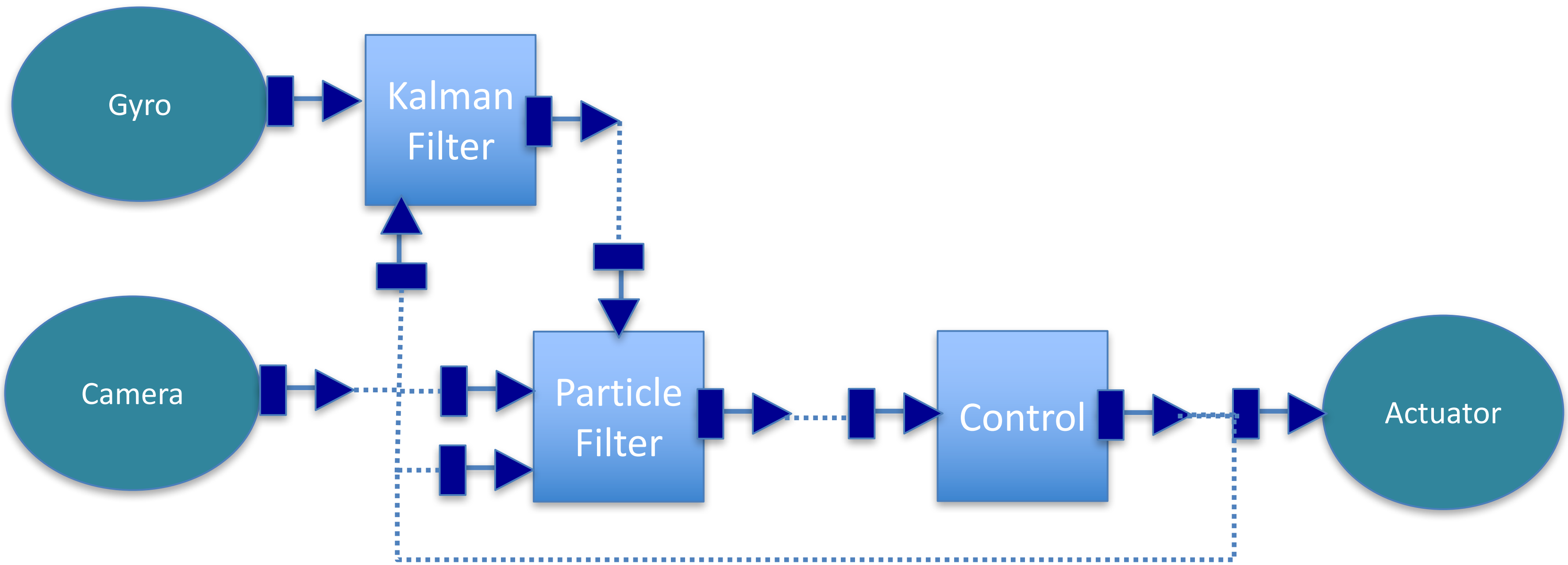
Filter chaining

ThingFlow Controller




```
g = Gyro()
k1 = Kalman(...)
pf = ParticleFilter(I={'gyro', 'camera'})
g.transduce(k1).delay()\
    .connect(pf, port_mapping=('default','gyro'))
c = Camera()
c.connect(pf, port_mapping=('default','camera'))
pf.connect(Controller(...))
```

ThingFlow Controller




```
c = Camera()
c.connect(mqtt_writer)
```

```
g = Gyro()
k1 = Kalman(...)
pf = ParticleFilter(I={'gyro', 'camera'})
g.transduce(k1).delay()\
    .connect(pf, port_mapping=('default','gyro'))
c = Camera()
c.connect(pf, port_mapping=('default','camera'))
pf.connect(Controller(...))
```

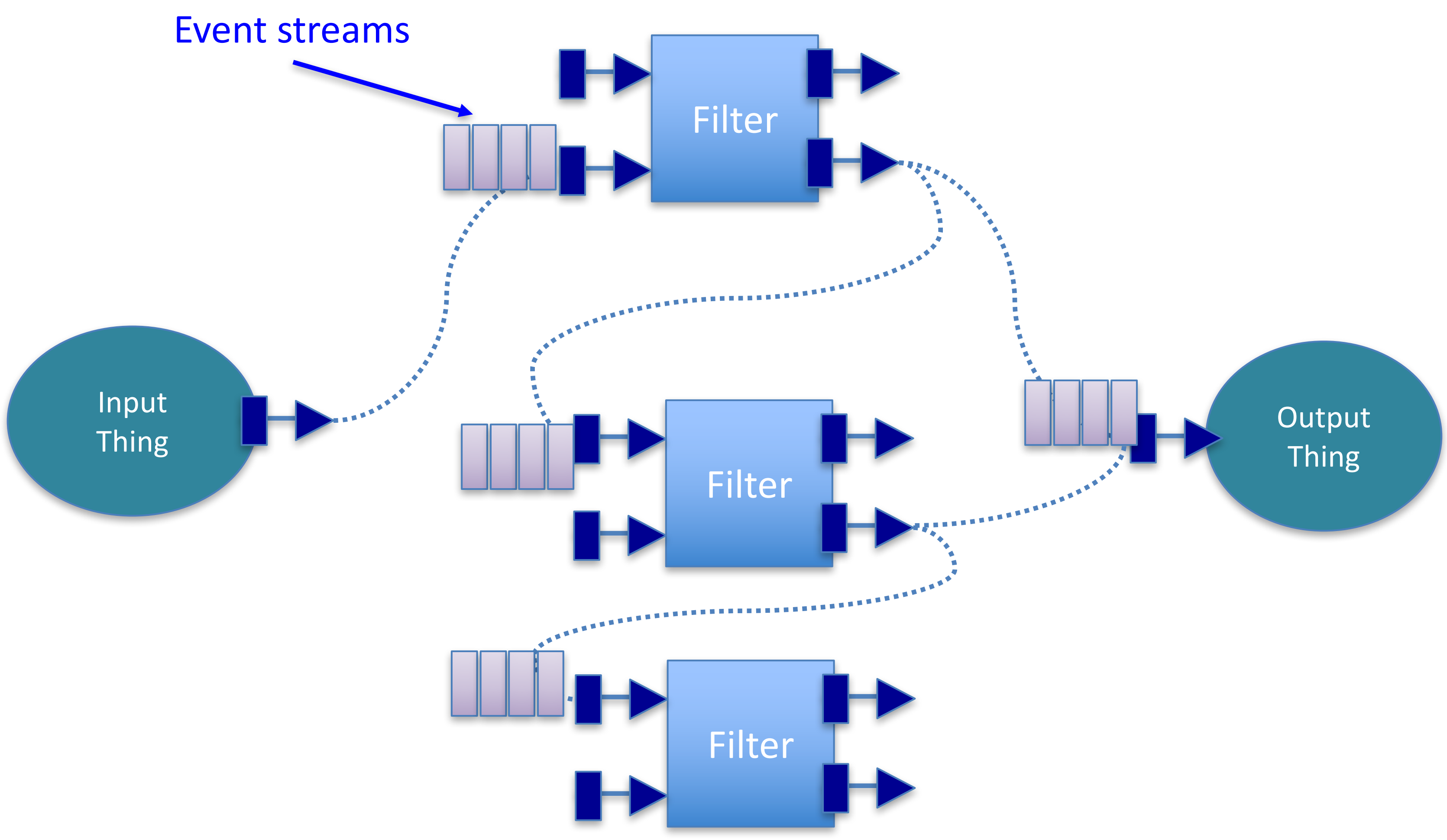
mqtt_reader(...)

ThingFlow Implementation

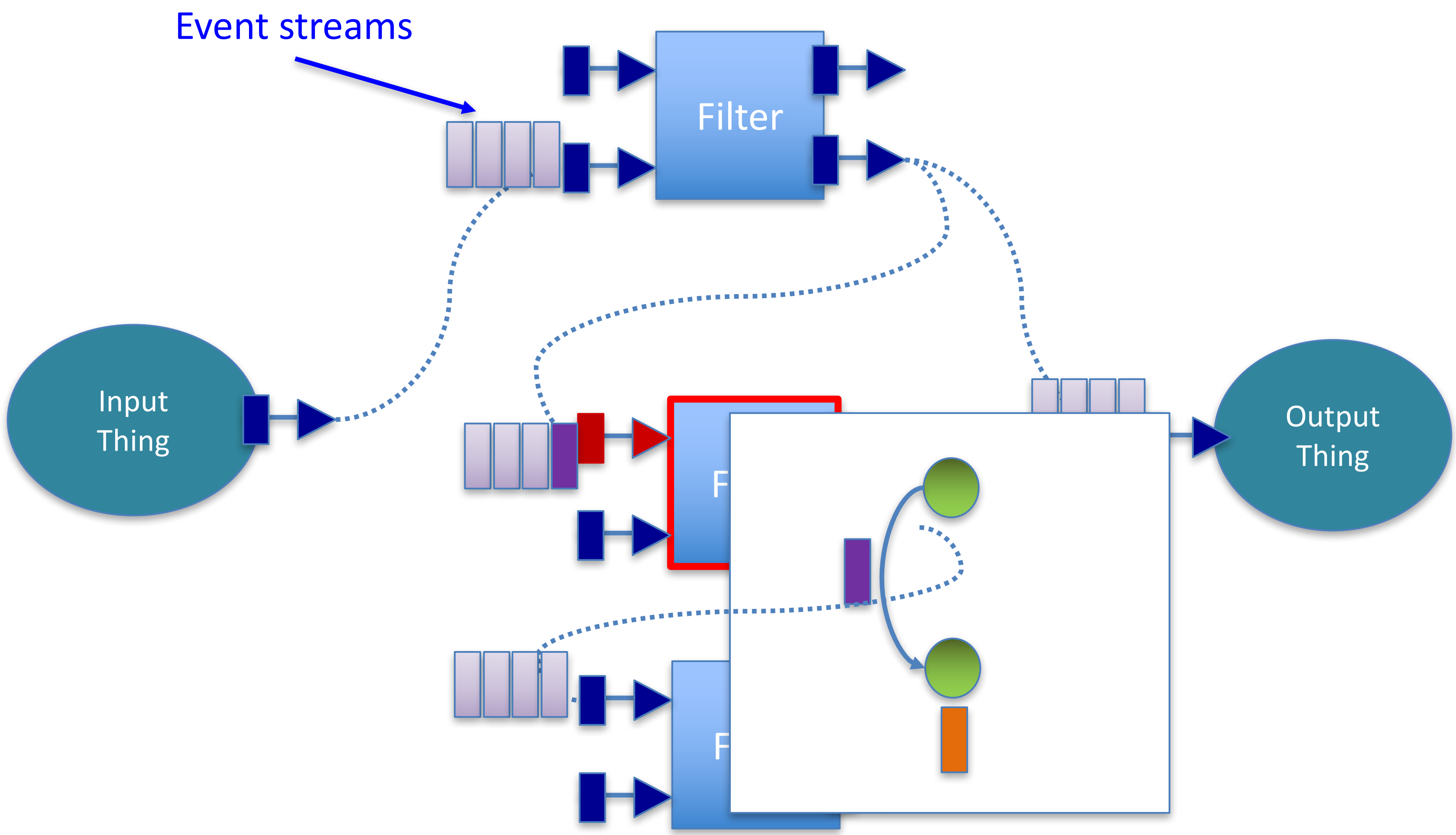
- Python3 library
  - CPython: standard Python implementation
  - MicroPython: “bare metal” implementation for embedded systems

https://github.com/mpi-sws-rse/thingflow-python

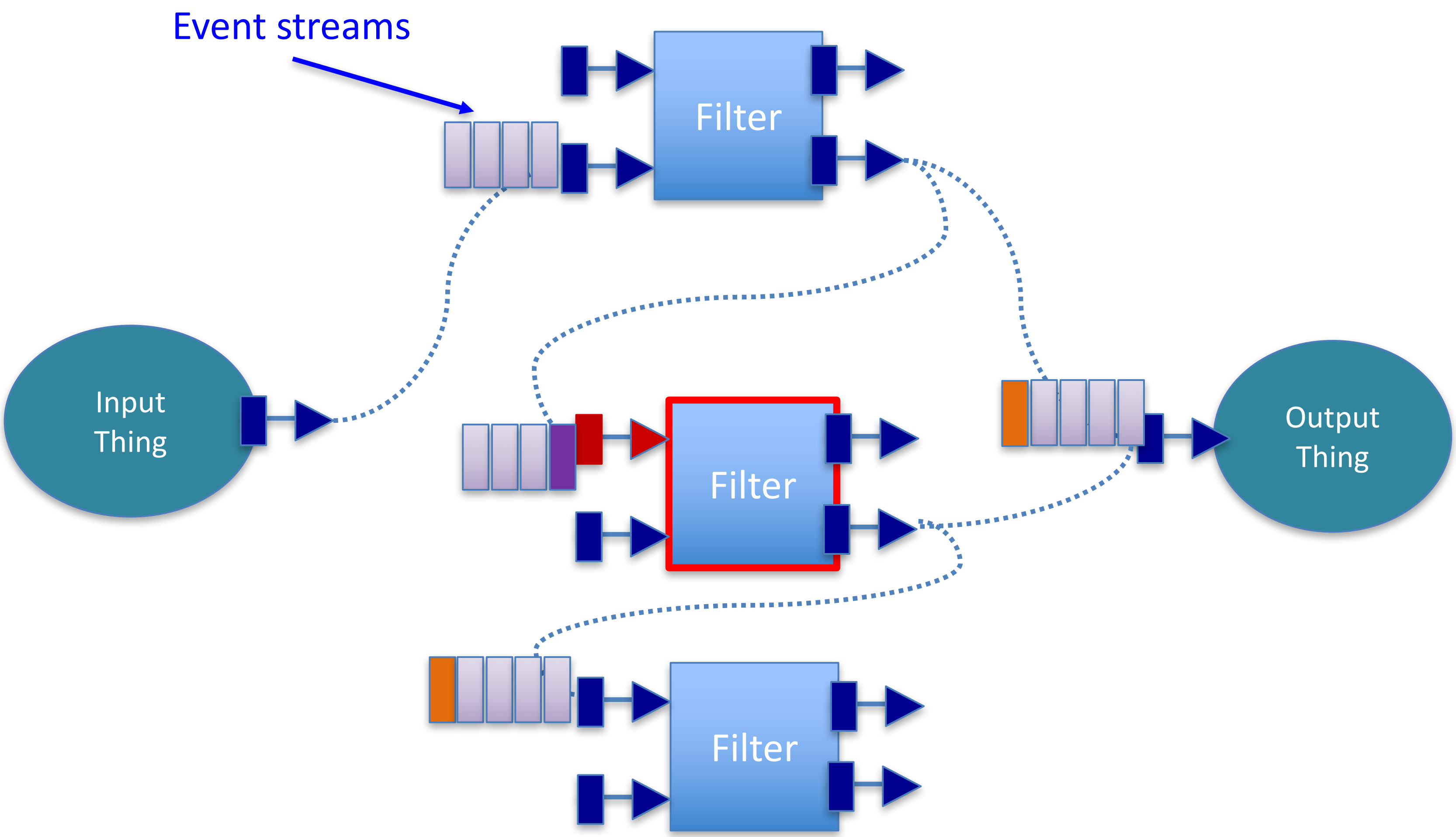
Semantics = Comm. State Machines



In Each Step…



In Each Step…



Event streams
Filter
Input
Thing
Filter
Output
Thing
Filter

In Each Step…

Infinite-state system:

1. Events are infinite-state: events can be chosen from an infinite set (e.g., real-valued signals)
2. Filters are infinite-state: the internal state of filters can be infinite (e.g., a Kalman filter)
3. Queues can be unbounded
4. Probabilistic: The filter transition function can be probabilistic

Semantics: Infinite-state Markov decision process:

- Scheduler picks policy
- State evolves probabilistically based on chosen filter

(under measureability assumptions)

Core language: Prob streams

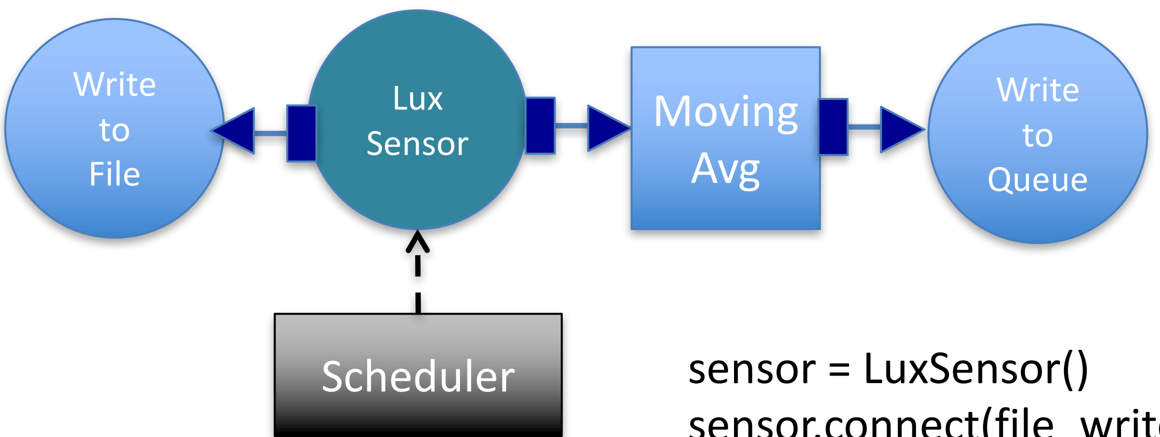
Reading from prob streams = sampling from the distribution

The ThingFlow Scheduler

- Responsible for scheduling “things”
  - Periodic observations (sensor sampling)
  - Non-periodic events (e.g. socket readiness)
  - Inter-thing events
- Abstraction over low level details
  - Threading, Order of scheduling
- Different implementations
  - On top of Python’s asyncio scheduler for Cpython
  - Custom, power-saving implementation for ESP8266
- ThingFlow programs must be explicitly scheduled to perform their tasks!

Simple ThingFlow Example

- Periodically sample a light sensor
- Write the sensed value to a file
- Every 5 steps, send the moving average to a message queue

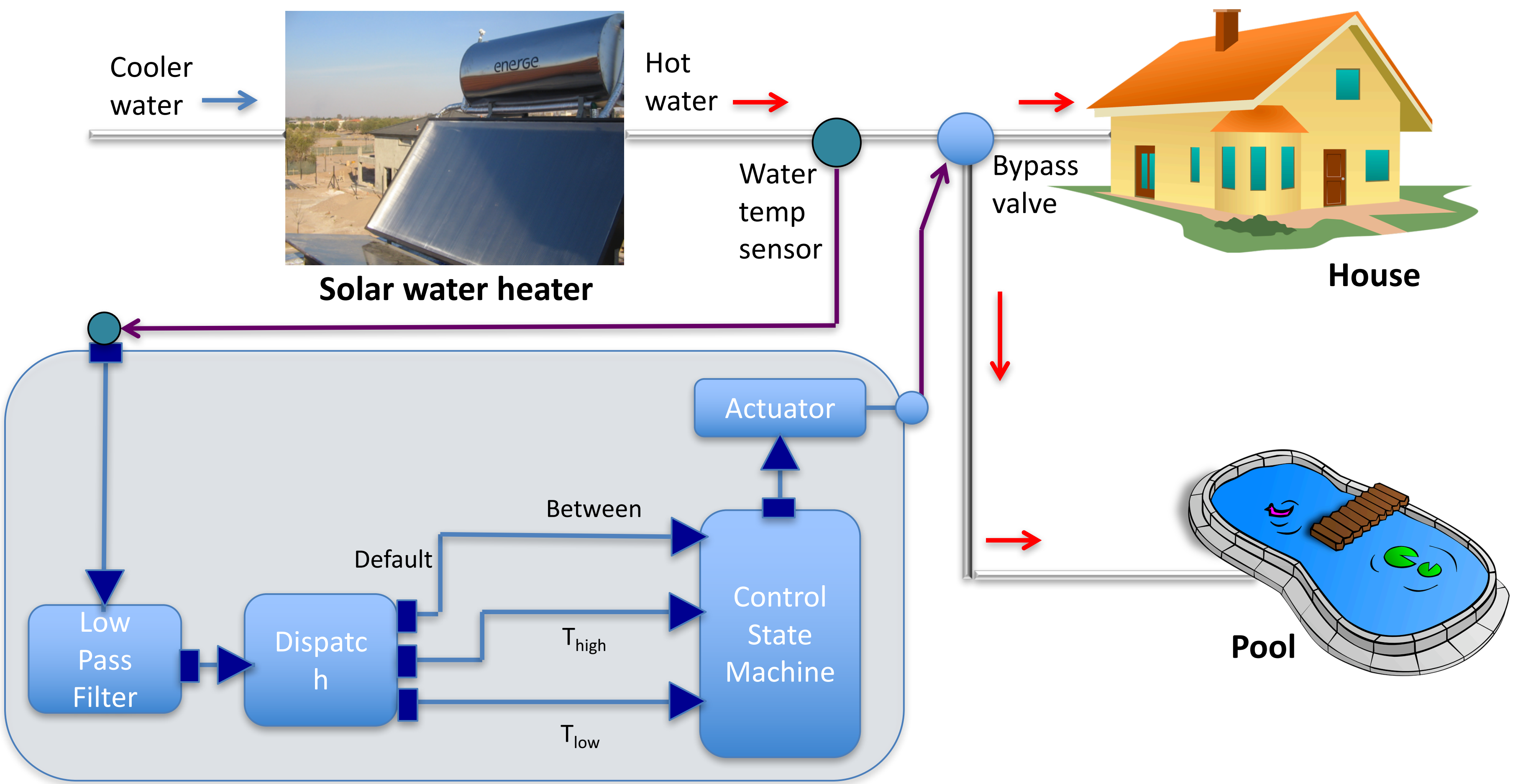



```
sensor = LuxSensor()
sensor.connect(file_writer('file'))
vingAvg(5)).connect(mqtt_writer)

r(asyncio.get_event_loop())
periodic(sensor, 2)
r()
```

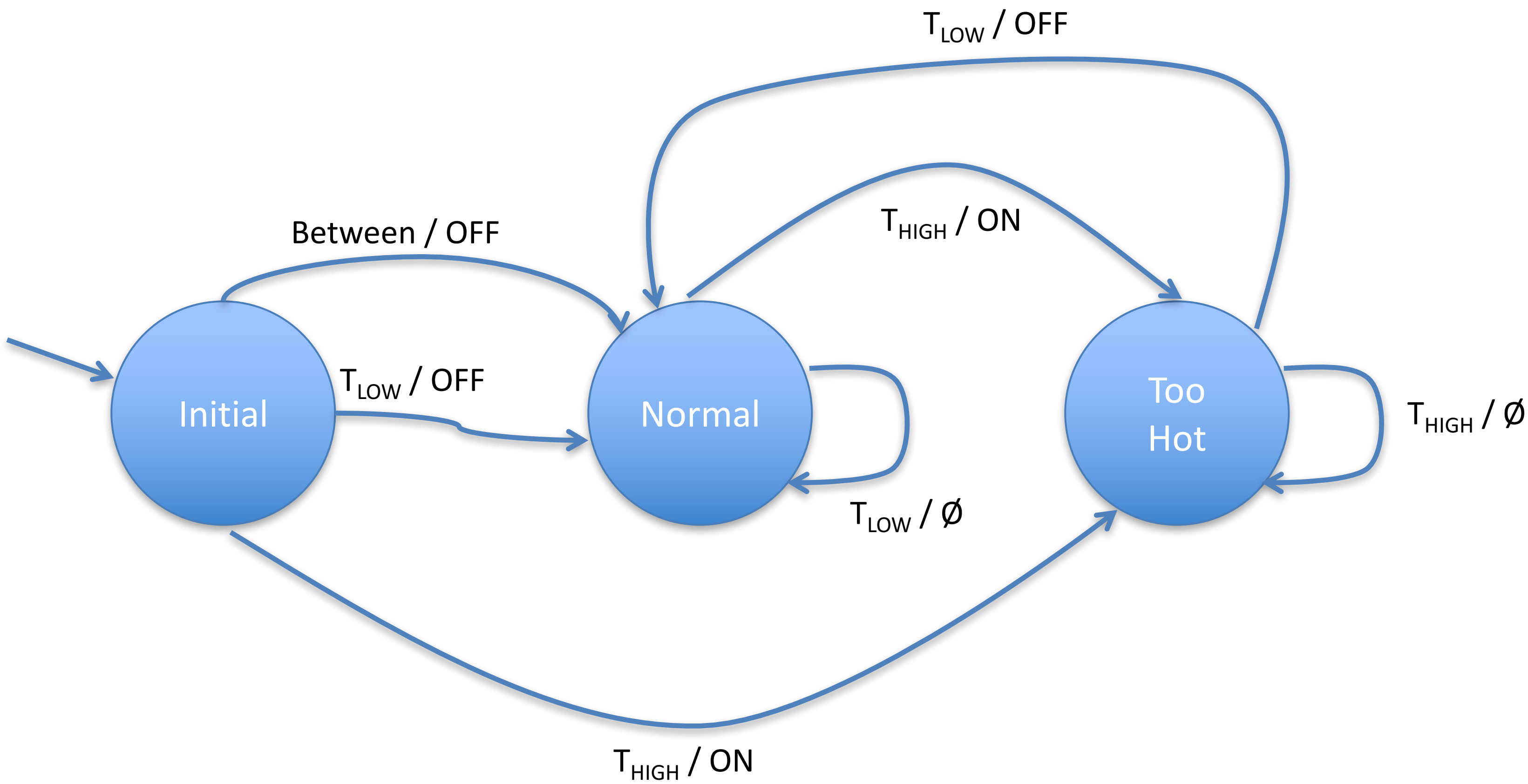
Default scheduler: Push an event entirely through the graph before handling the next input

-- Can replace async calls by sync calls

Solar Heater Example



Solar Heater Example: Controller State Machine

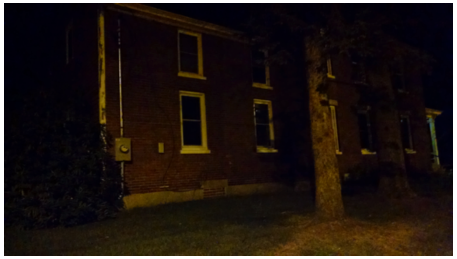


Solar Heater Example: Code

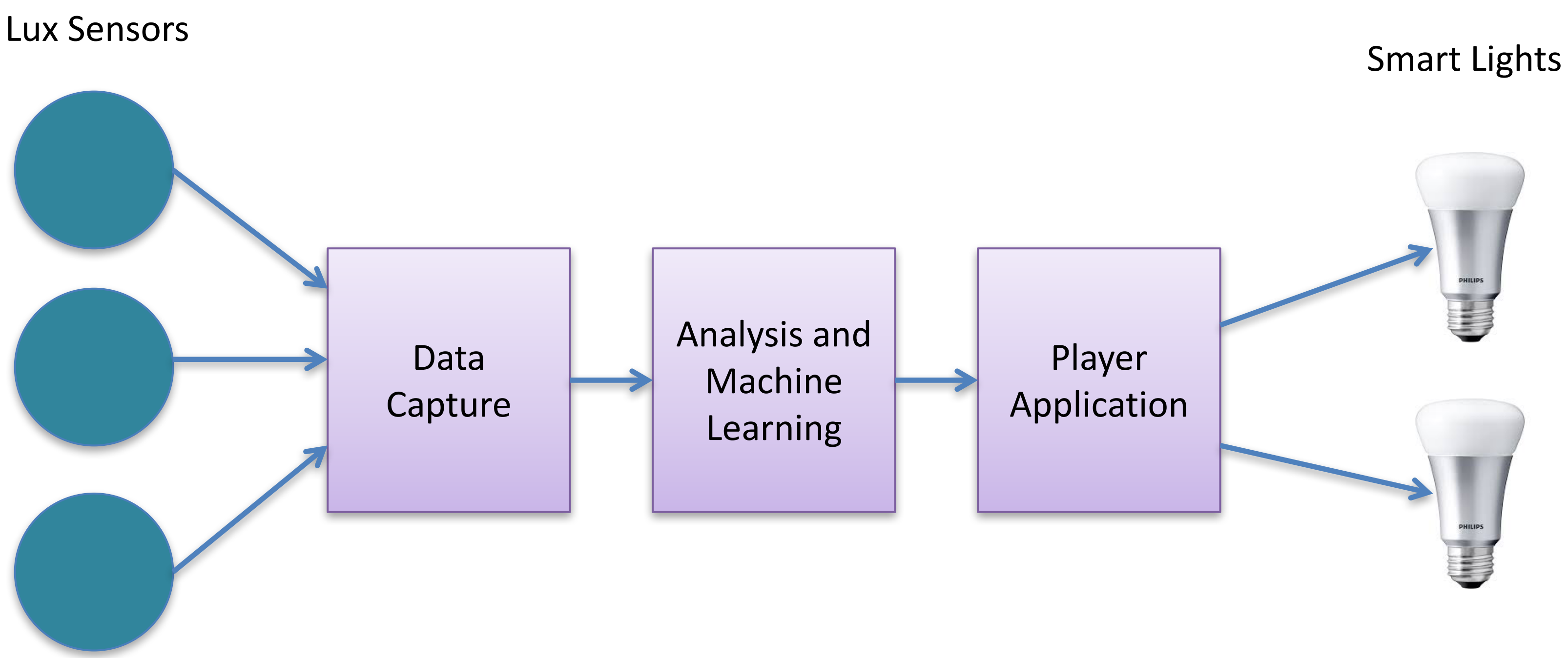
```
T_high = 110 # Upper threshold (degrees fahrenheit)
T_low = 90 # Lower threshold
sensor = TempSensor(gpio_port=1)

# The dispatcher converts a sensor reading into
# threshold events
dispatcher = sensor.transduce(RunningAvg(5)) \
                   .dispatch([(lambda v: v[2]>=T_high, ’t_high'),
                              (lambda v: v[2]<=T_low, ’t_low')])
controller = Controller()
dispatcher.connect(controller, port_mapping=(’t_high’,’t_high'))
dispatcher.connect(controller, port_mapping=(’t_low', ’t_low'))
dispatcher.connect(controller, port_mapping=('default’, 'between'))

actuator = Actuator()
controller.connect(actuator)
```

Lighting Project: Motivation

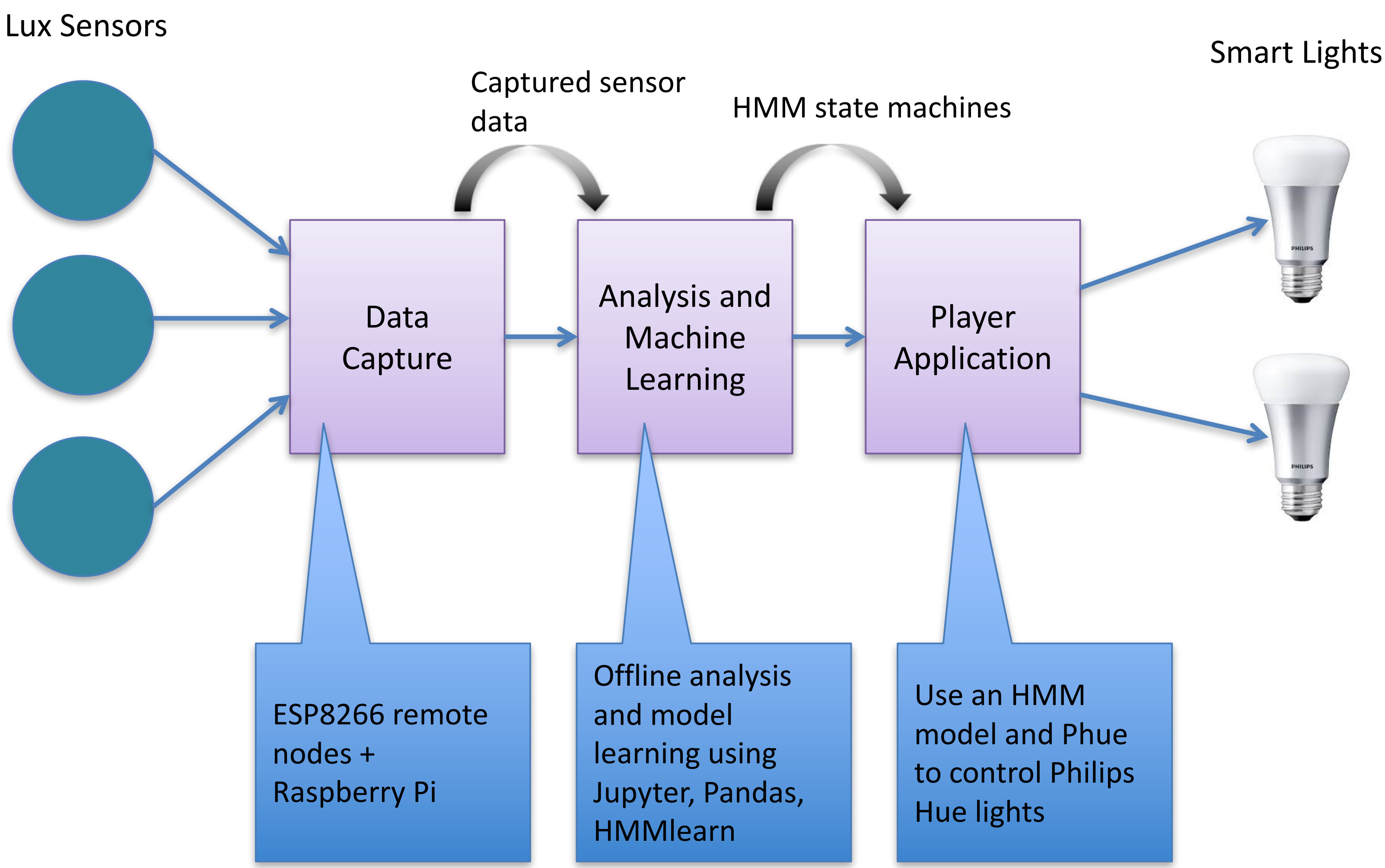
- If out of town for the weekend, don’t want to leave the house dark
- Replay lights “similar” to normal lighting pattern



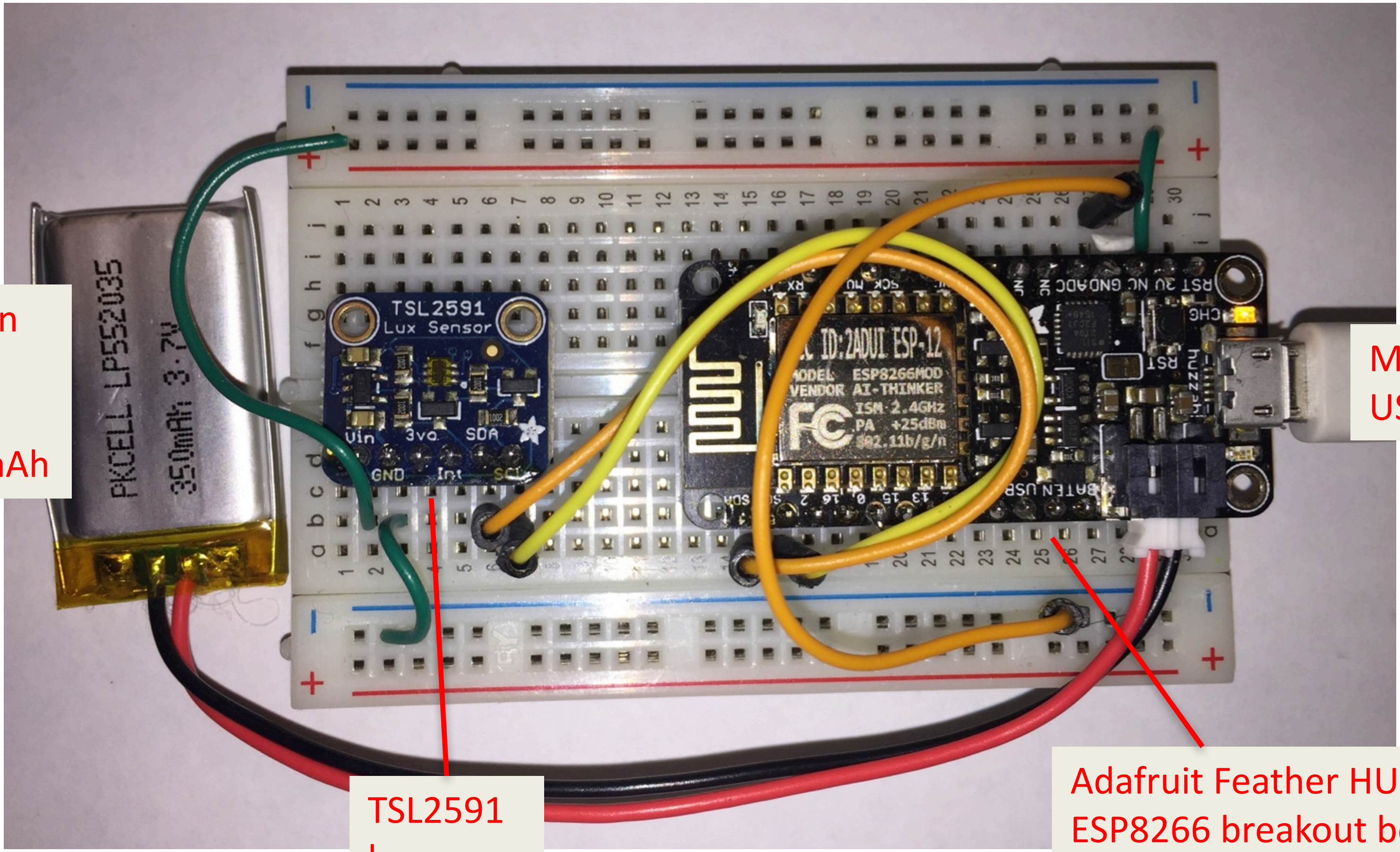
Lighting Replay Application



Lighting Replay Application



ESP8266



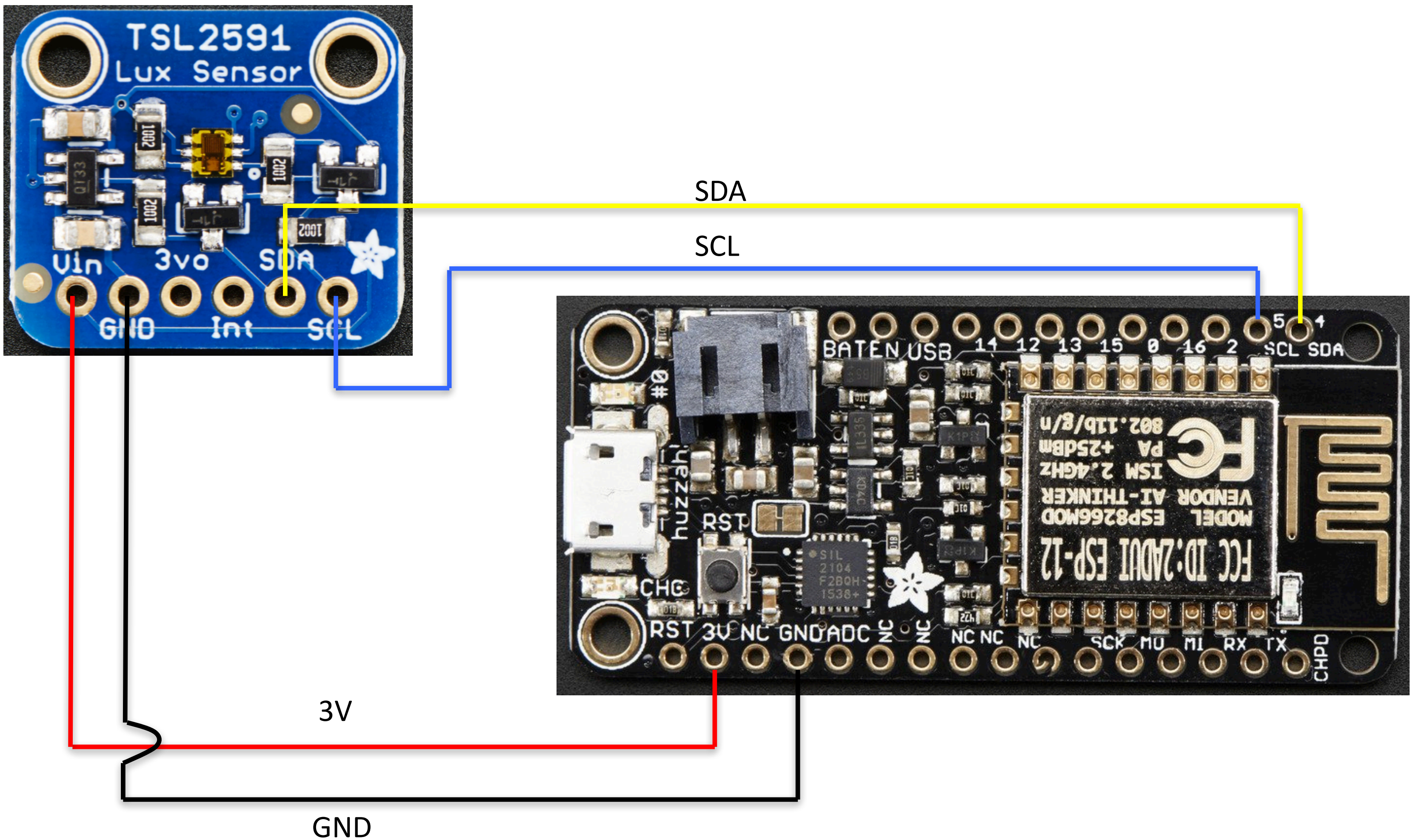
Lithium Ion Polymer Battery 3.7v 350mAh

TSL2591 lux sensor breakout board

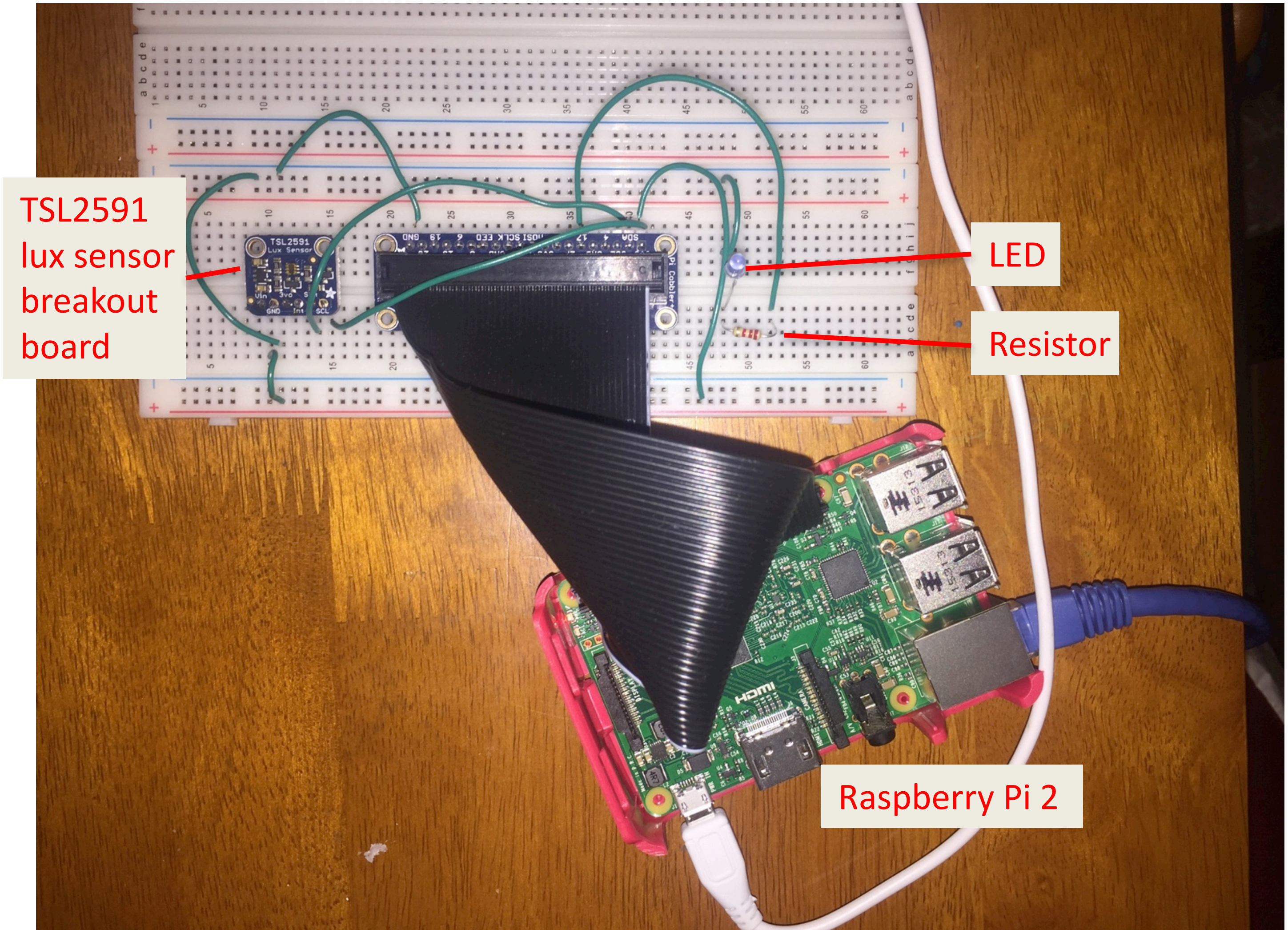
Adafruit Feather HUZZAH ESP8266 breakout board

MicroUSB to USB cable

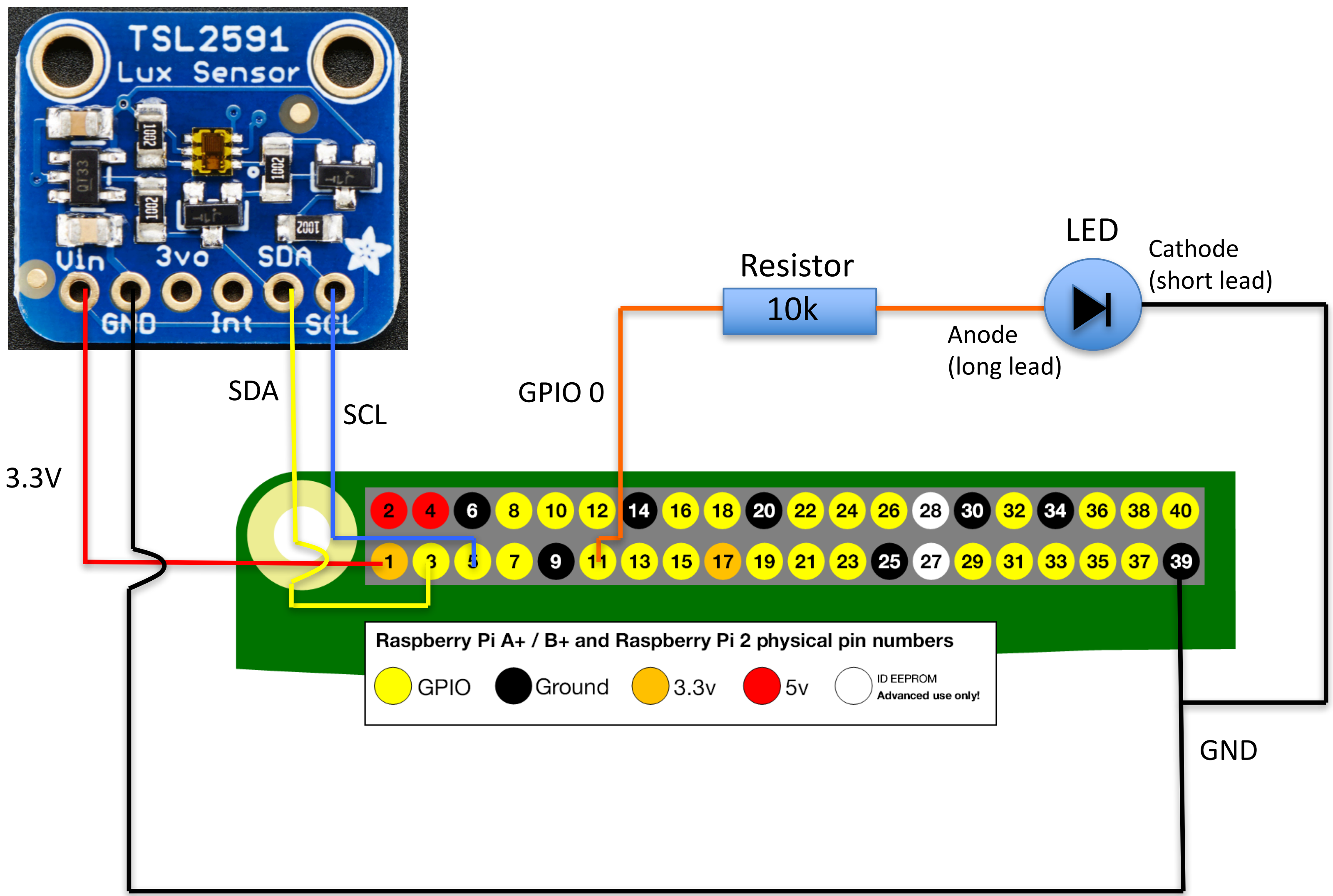
ESP8266: Wiring Diagram



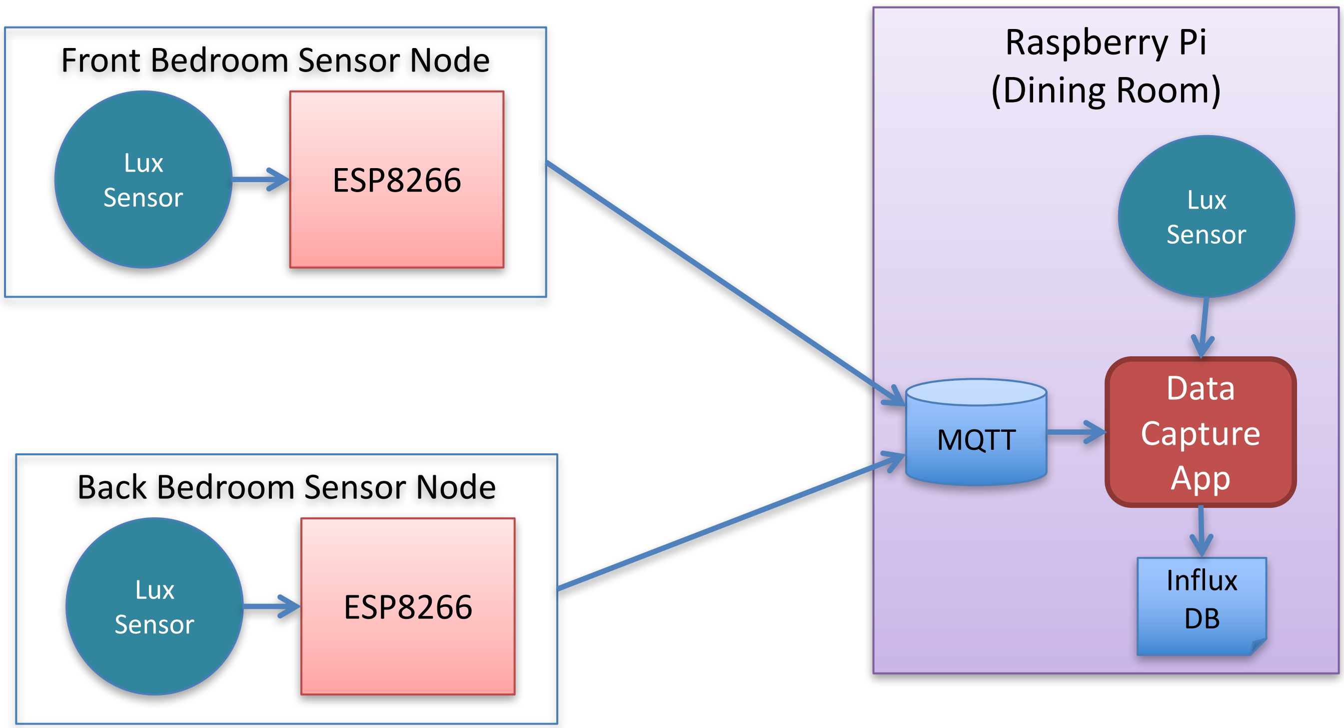
Raspberry Pi



Raspberry Pi: Wiring Diagram



Lighting Replay Application: Capture



ESP8266 Code (ThingFlow)

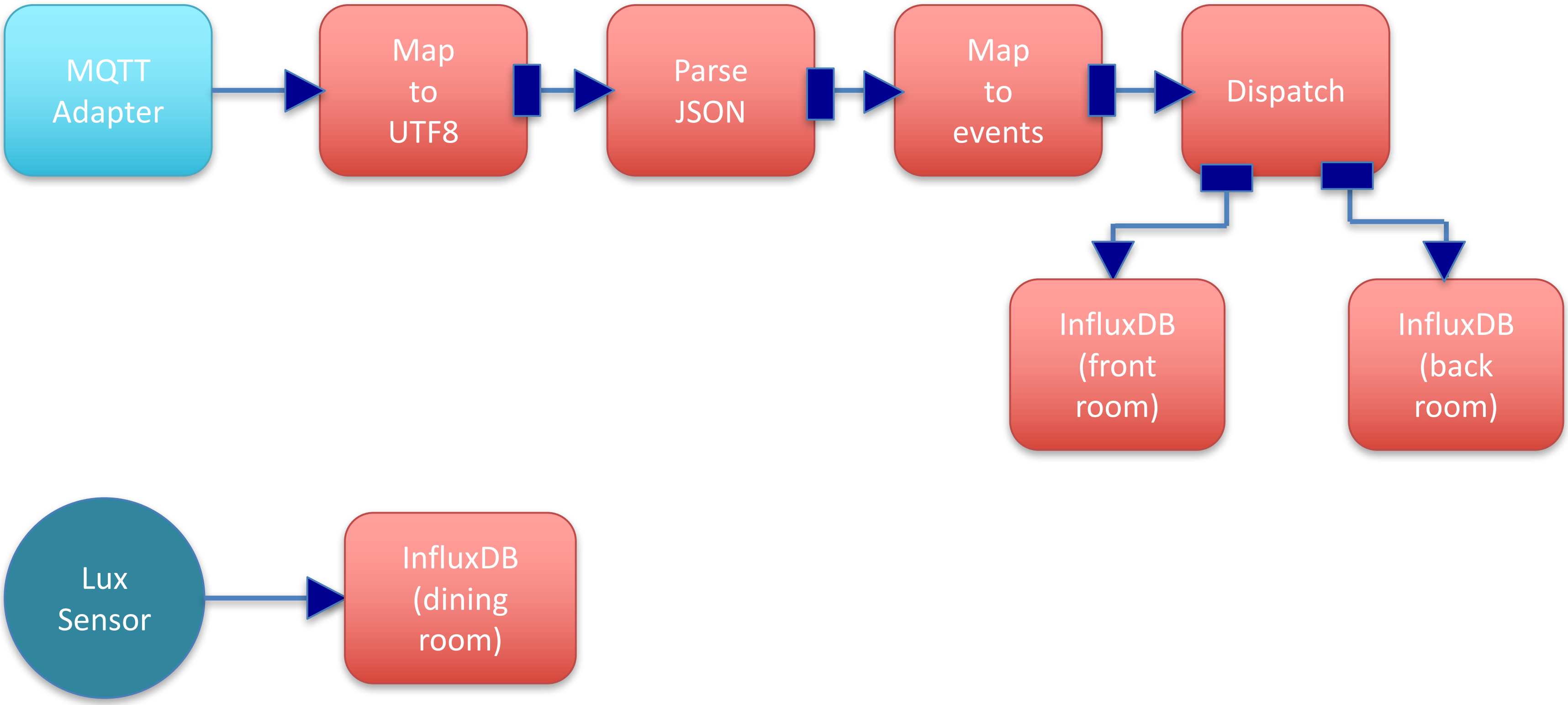
```
from thingflow import Scheduler, SensorAsOutputThing
from tsl2591 import Tsl2591
from mqtt_writer import MQTTWriter
from wifi import wifi_connect
import os

# Params to set
WIFI_SID= …
WIFI_PW= …
SENSOR_ID="front-room"
BROKER='192.168.11.153'

wifi_connect(WIFI_SID, WIFI_PW)
sensor = SensorAsOutputThing(Tsl2591())
writer = MQTTWriter(SENSOR_ID, BROKER, 1883,
                    'remote-sensors')
sensor.connect(writer)

sched = Scheduler()
sched.schedule_periodic(sensor, SENSOR_ID, 60)
sched.run_forever()
```

The MQTT writer is connected to the lux sensor.

Sample at 60 second intervals

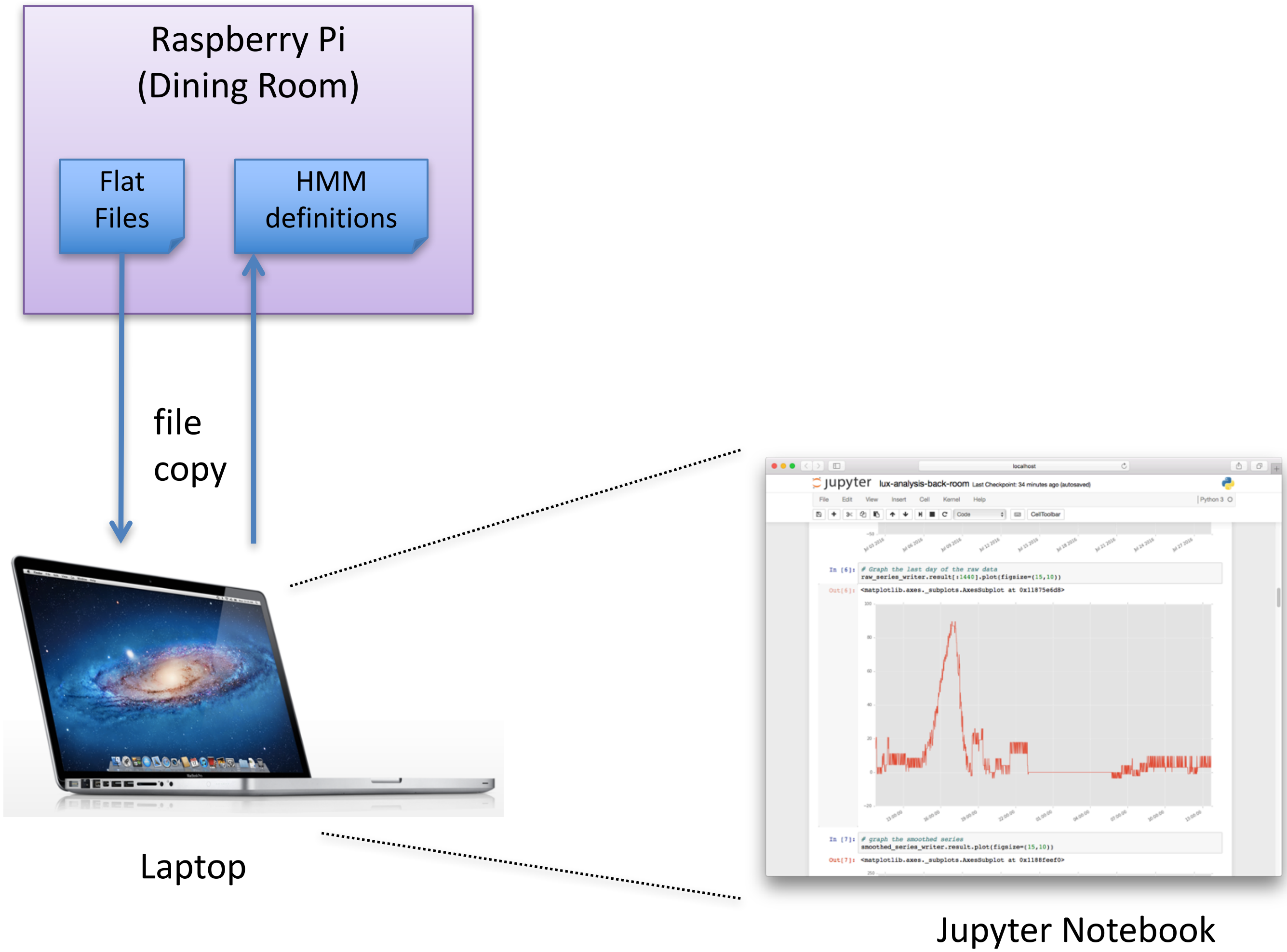
See https://github.com/mpi-sws-rse/thingflow-examples/blob/master/lighting_replay_app/capture/esp8266_main.py

Raspberry Pi Code (ThingFlow)



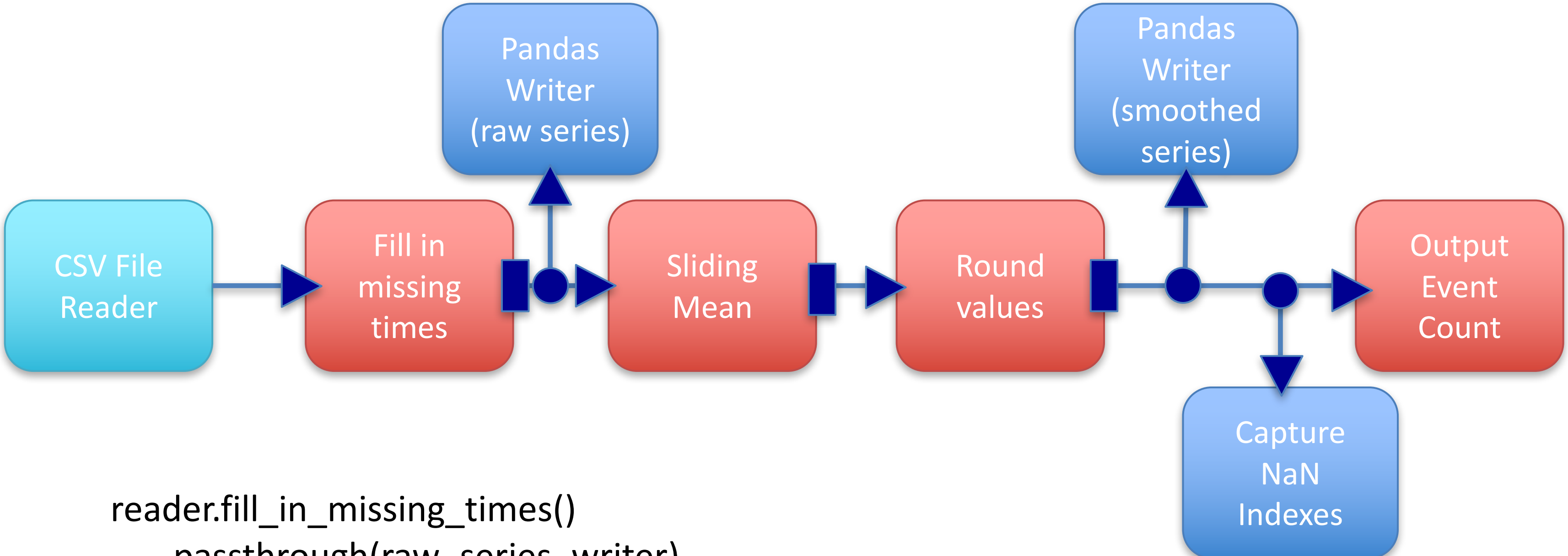
https://github.com/mpi-sws-rse/thingflow-examples/blob/master/lighting_replay_app/capture/sensor_capture.py

Lighting Replay Application: Analysis



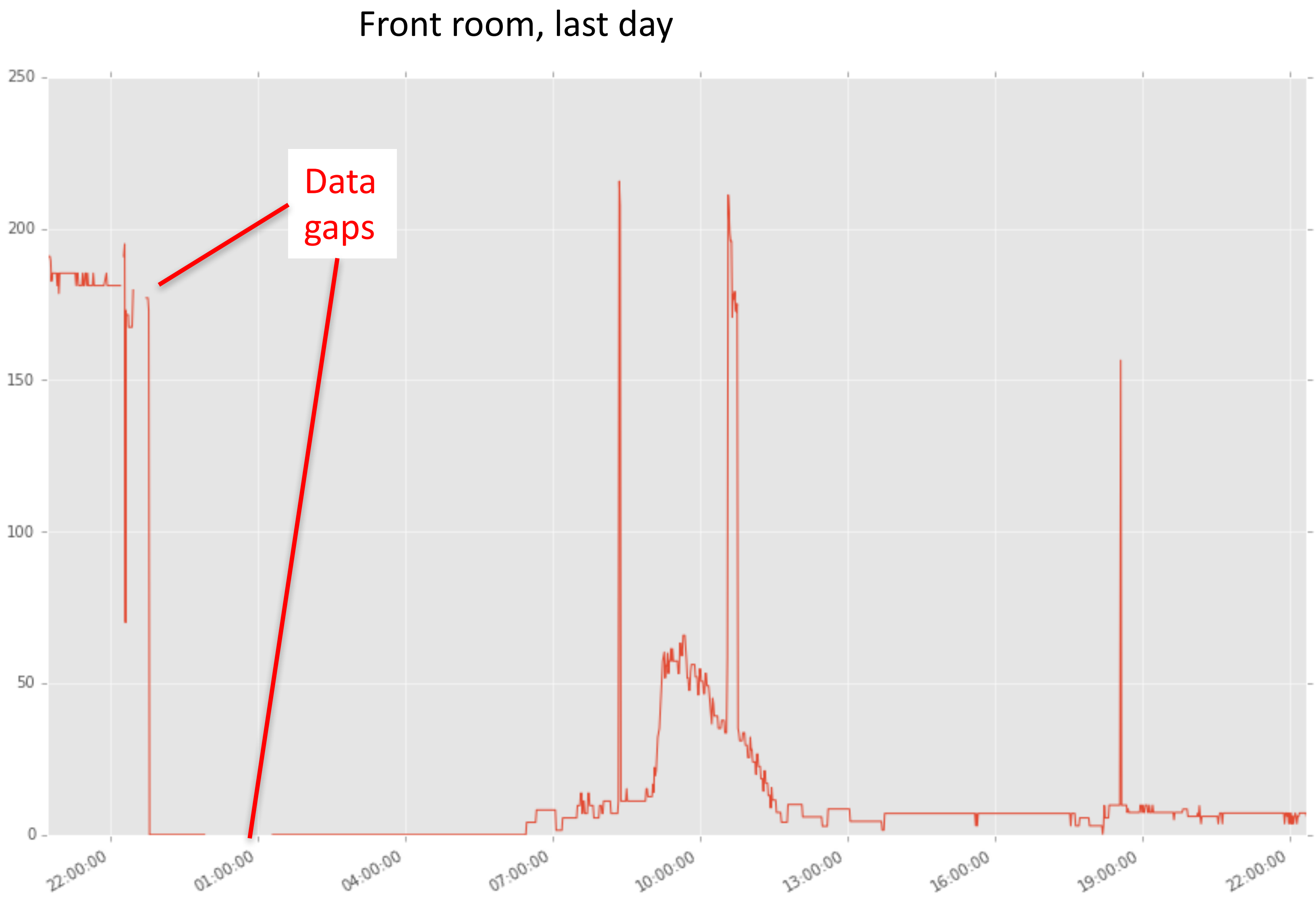
Preprocessing the Data

(ThingFlow running in a Jupyter Notebook)

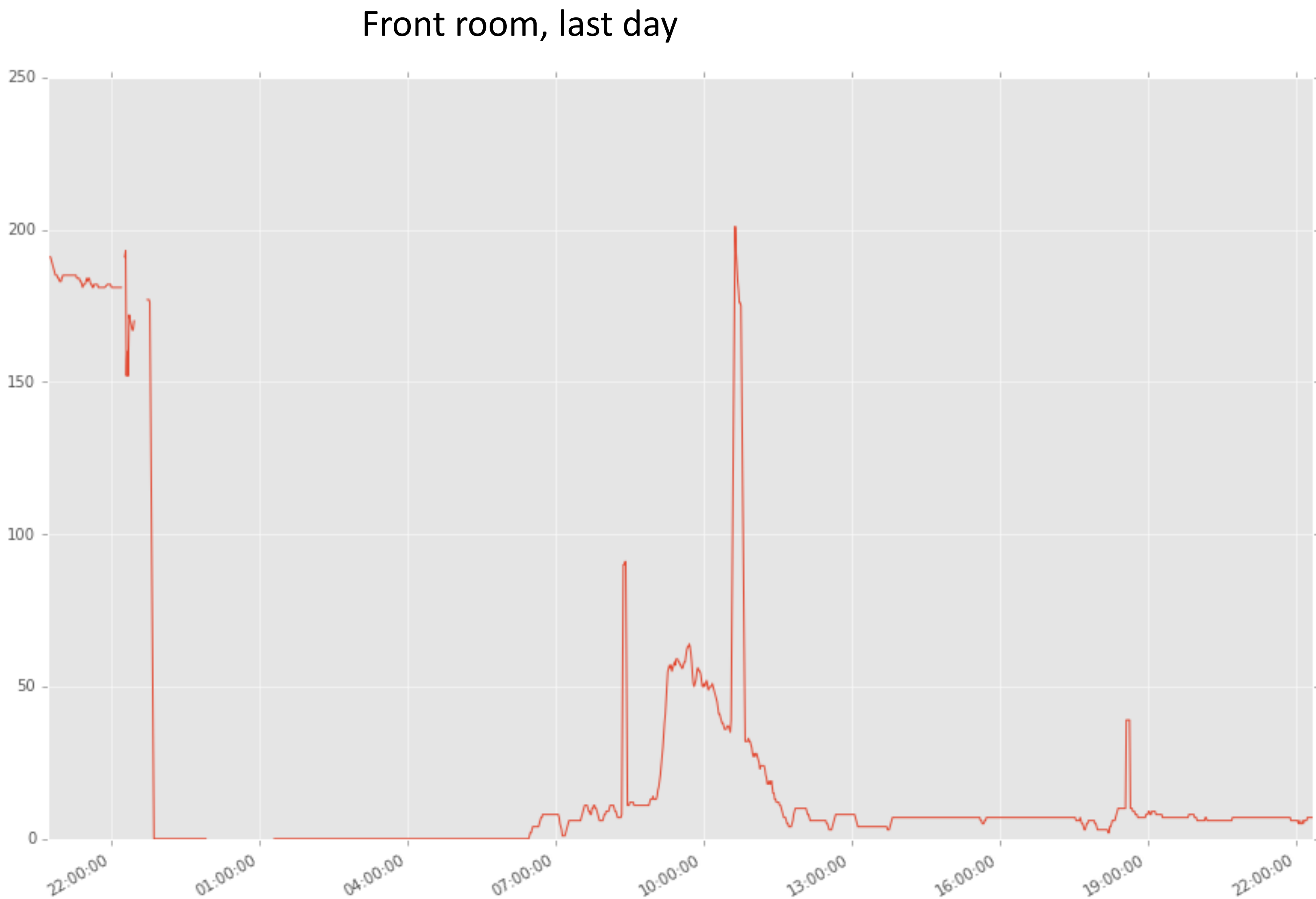



```
reader.fill_in_missing_times()
    .passthrough(raw_series_writer)
    .transduce(SensorSlidingMeanPassNaNs(5))
    .select(round_event_val)
    .passthrough(smoothed_series_writer)
    .passthrough(capture_nan_indexes)
    .output_count()
```

Data Processing: Raw Data

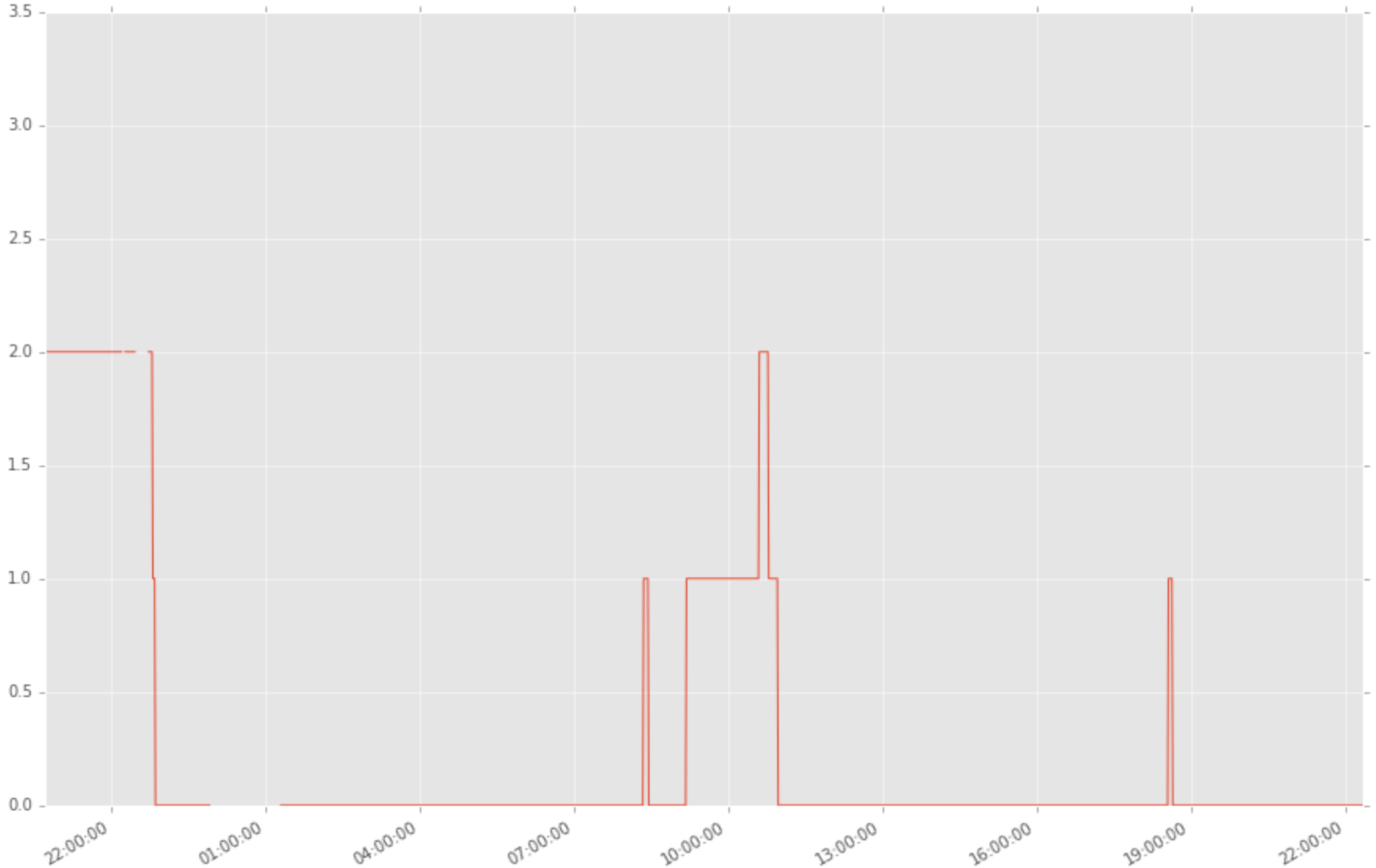


Data Processing: Smoothed Data

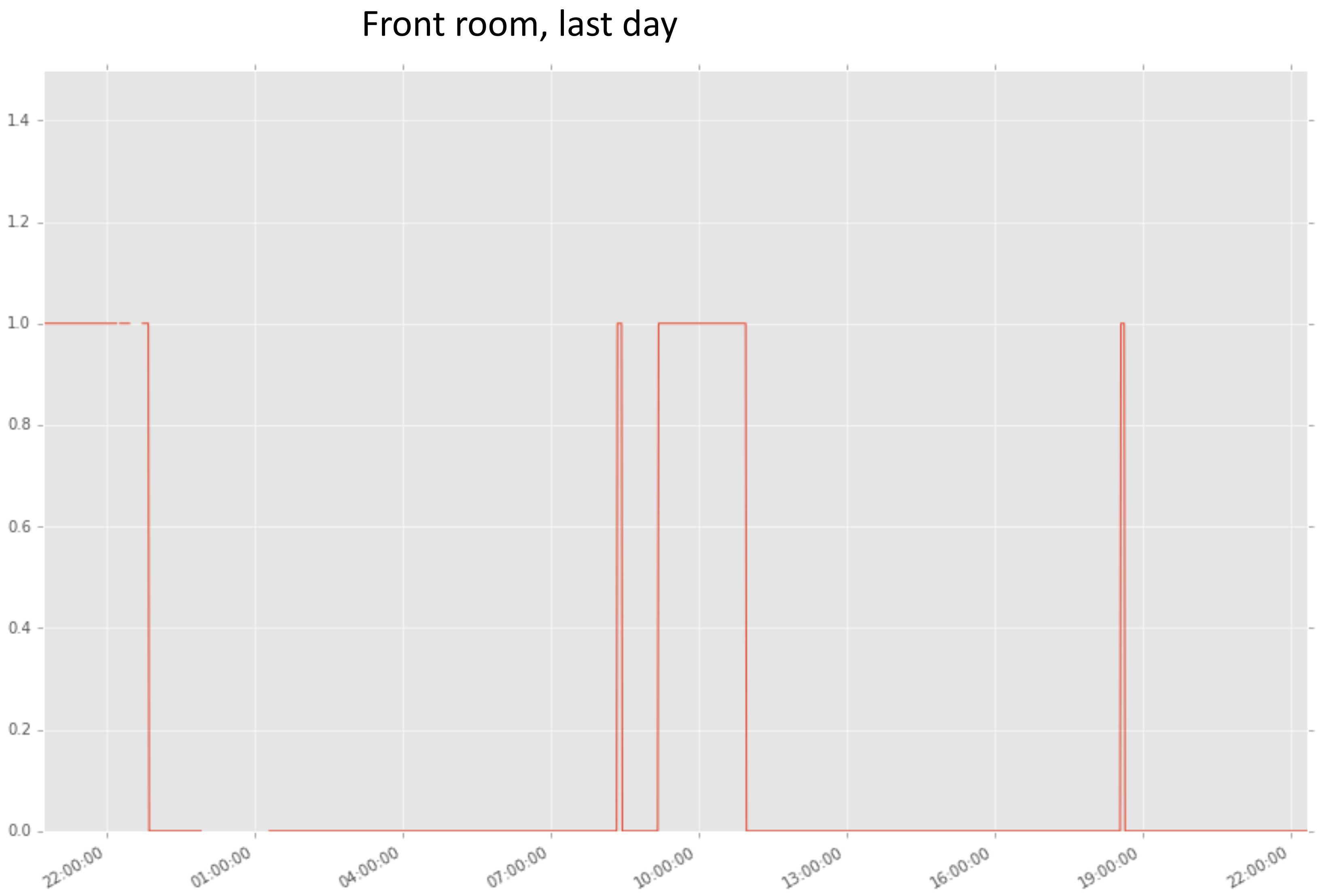


Data Processing: K-Means Clustering

Front room, last day

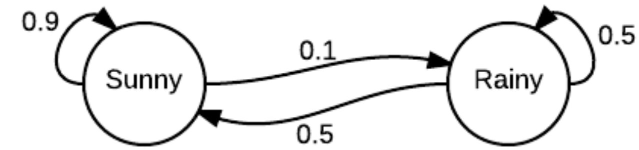


Data Processing: Mapping to on-off values



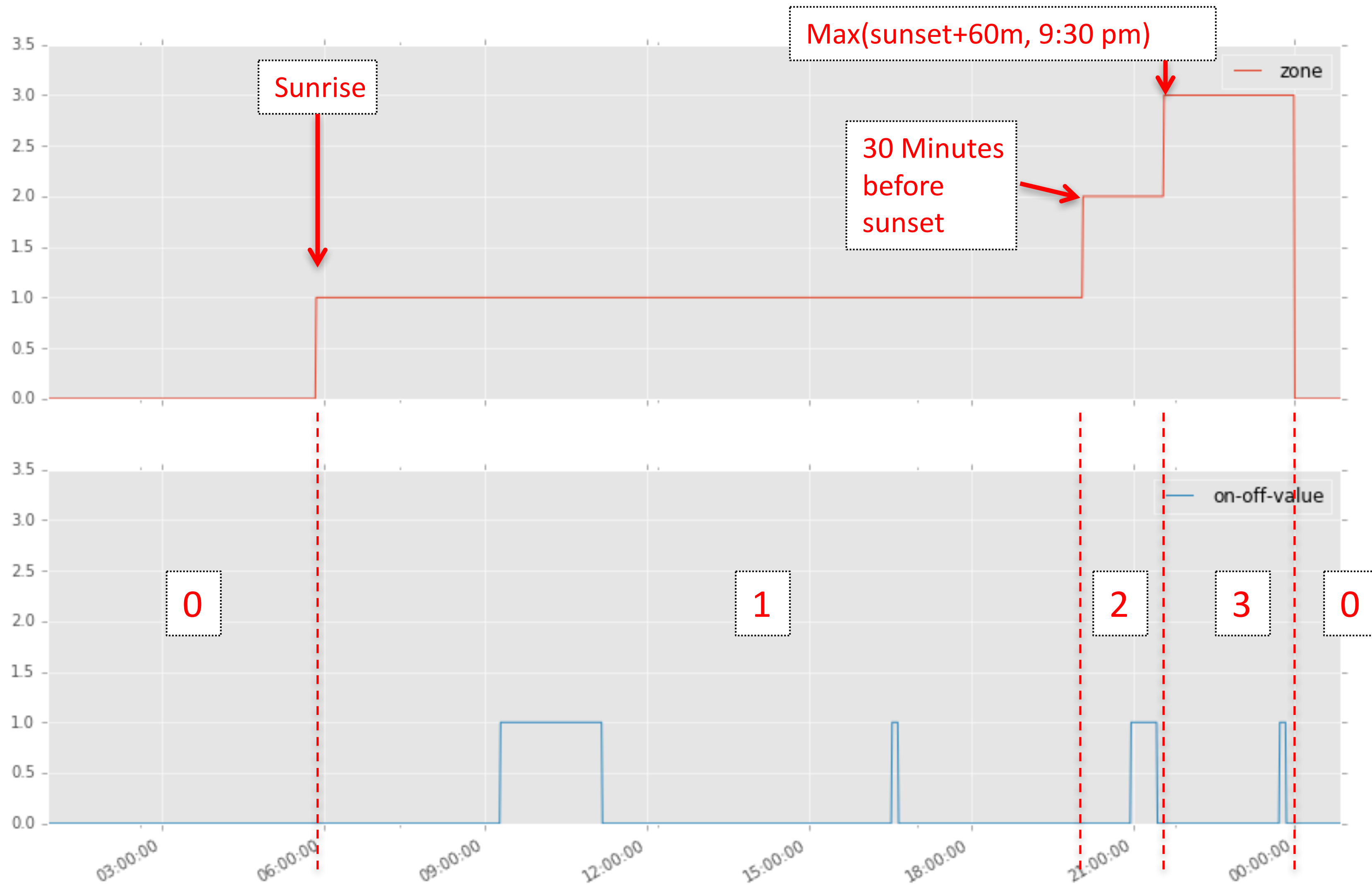
Hidden Markov Models (HMMs)

- *Markov process*
  - State machine with probability associated with each outgoing transition
  - Probabilities determined only by the current state, not on history
- Hidden Markov Model
  - The states are not visible to the observer, only the outputs ("emissions").
- In a machine learning context:
  - (Sequence of emissions, # states) => inferred HMM
- The `hmmlearn` library will do this for us.
  - https://github.com/hmmlearn/hmmlearn



Example Markov process (from Wikipedia)

Slicing Data into Time-based "Zones"



HMM Training and Prediction Process

Training

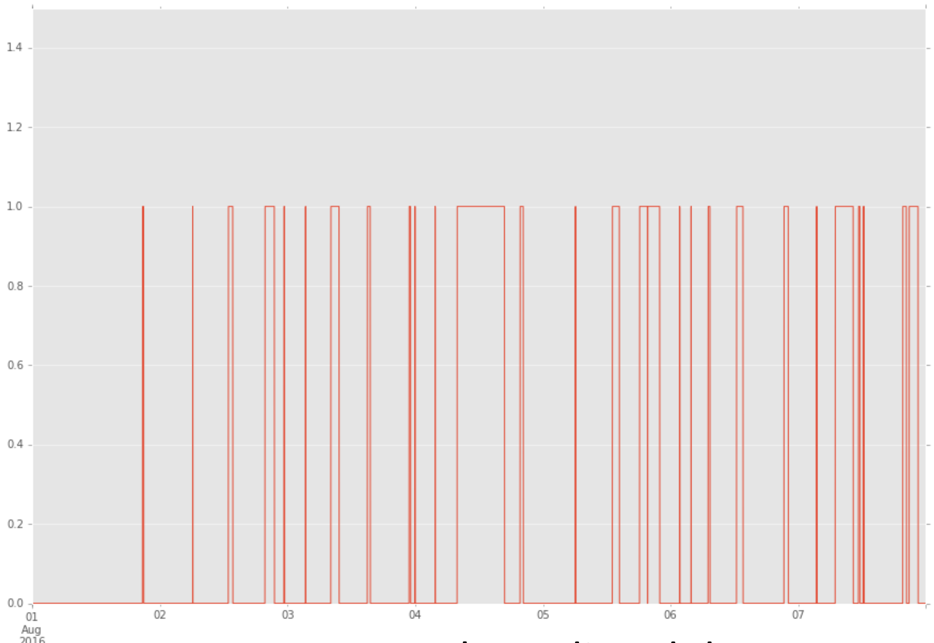
1. Build a list of sample subsequences for each zone
2. Guess a number of states (e.g. 5)
3. For each zone, create an HMM and call `fit()` with the subsequences

Prediction

For each zone of a given day:

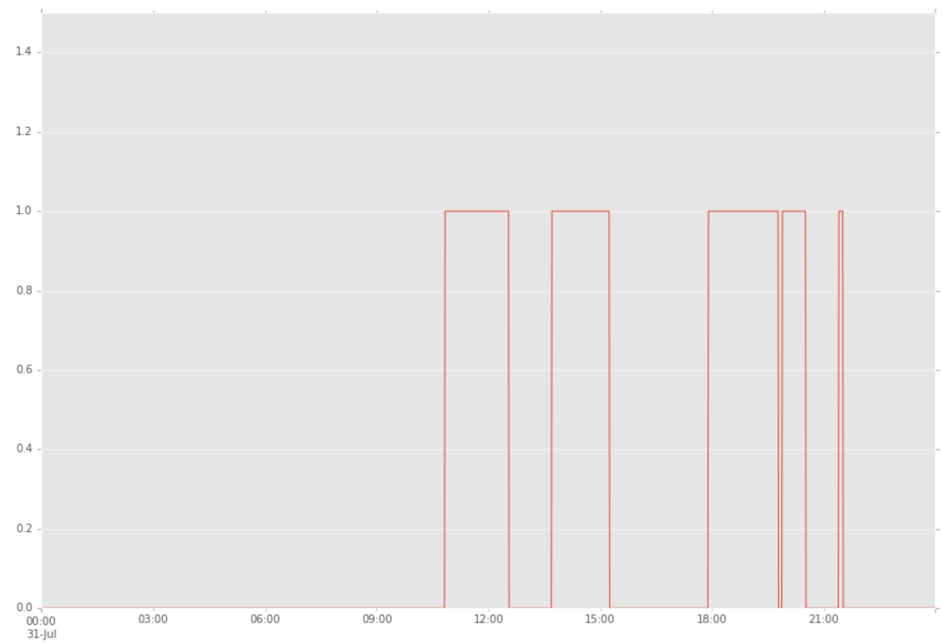
- Run the associated HMM to generate N samples for an N minute zone duration
- Associated a computed timestamp with each sample

HMM Predicted Data

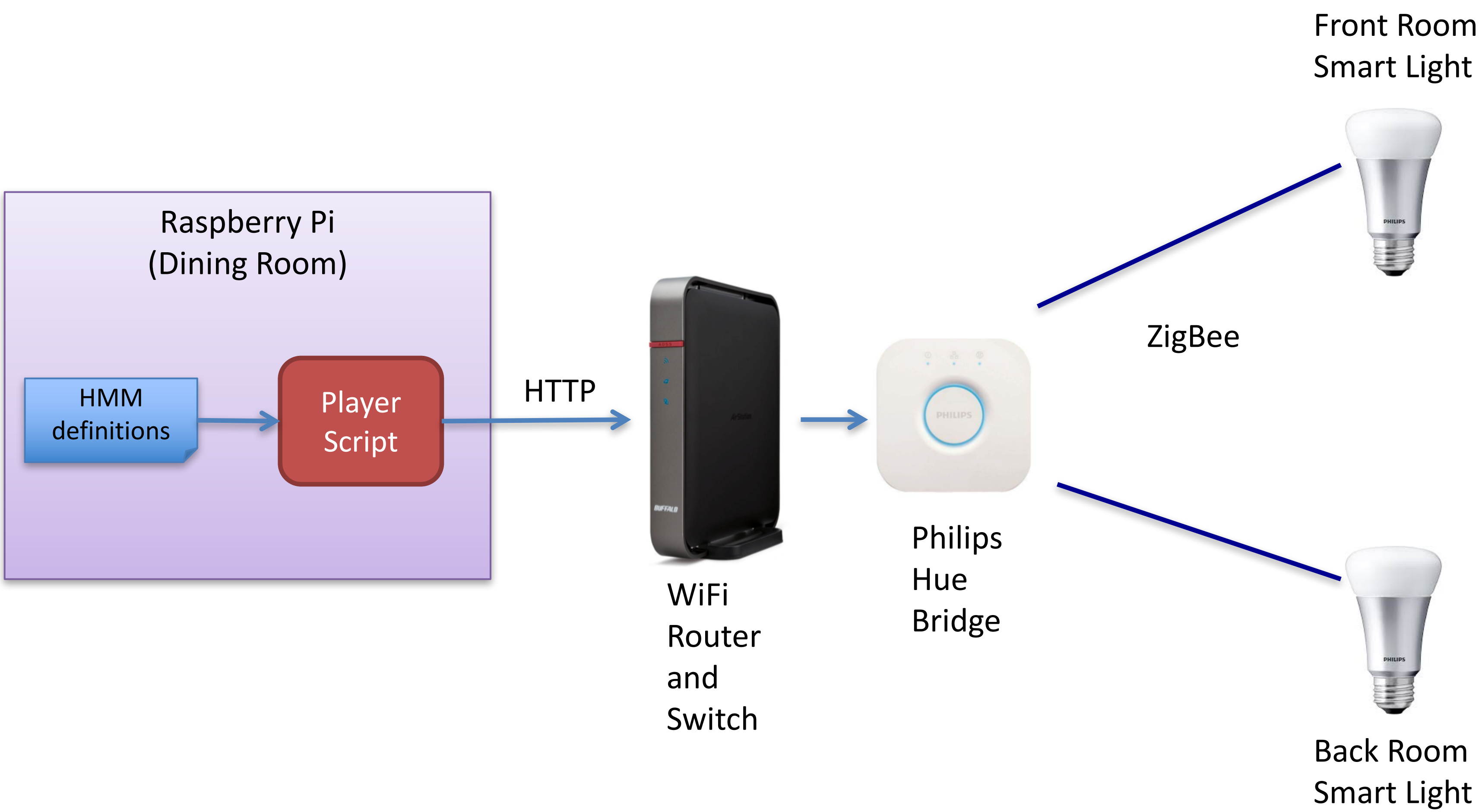


Front room, one week predicted data

Front room, one day predicted data



Lighting Replay Application: Replay



Logic of the Replay Script

- Use phue library to control lights
- Reuse time zone logic and HMMs from analysis
- Pseudo-code:

```
Initial testing of lights
while True:
    compute predicted values for rest of day
    organize predictions into a time-sorted list of on/off events
    for each event:
        sleep until event time
        send control message for event
    wait until next day
```

https://github.com/mpi-sws-rse/thingflow-examples/blob/master/lighting_replay_app/player/lux_player.py

ThingFlow: Analysis

- Bad news: Communicating finite-state machines + FIFO queues = everything is undecidable!

- Decidable verification in special cases: finite-state events & filters, ordering of messages ignored

- Analyzing a filter: Abstraction & approximation of infinite-state probabilistic processes
  - algorithms with guaranteed error bounds

- *Open: Tools and analyses for Thingflow programs*
  - *Asynchrony, Hybrid systems, Uncertainty, Distribution*

Analysis of ThingFlow

Two example analyses for subcases:

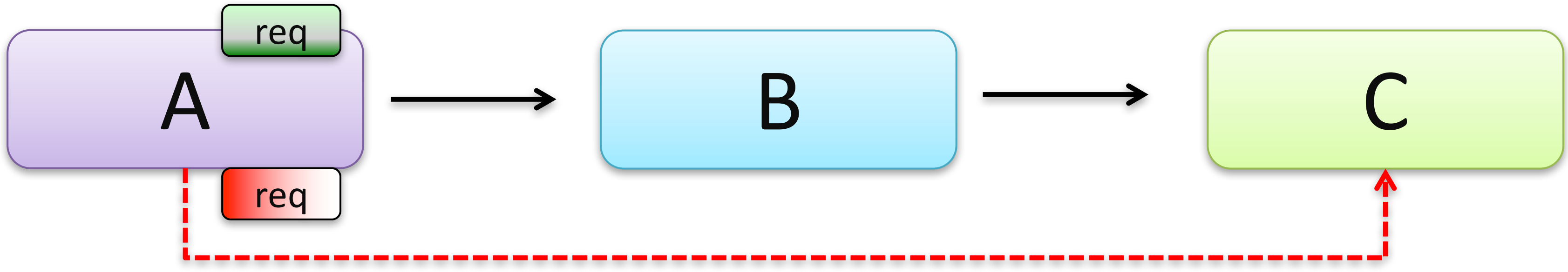
1. Analyzing event flows: Provenance Analysis
   [Joint work with Roland Meyer & Zilong Wang]

2. Analyzing a filter: Abstracting infinite-state Markov processes
   [Joint work with Sadegh Soudjani and Alessandro Abate]

Provenance

Information about the *source* and *access history* of an object

"All inputs to controller are sanitized"



Provenance for ThingFlow

- Associate principals with filters
- Provenance of a message =
  Principals who have sent the message chronologically
- Provenance domain =
  Strings over principal names

Provenance Verification Problem

Given a Thingflow program *P*, a stream *x*, and a regular set *R* of provenances,

are the provenances of all events in *x* always in the set *R* along all executions of *P*?

Assumptions: Finitely many events, finite-state filters, and ordering of events in queues ignored

Note: The system is still infinite-state!

Example: All inputs to controller have passed through a sanitizer and then a state estimator

Provenance Verification Problem

Given a Thingflow program *P*, a stream *x*, and a regular set *R* of provenances,

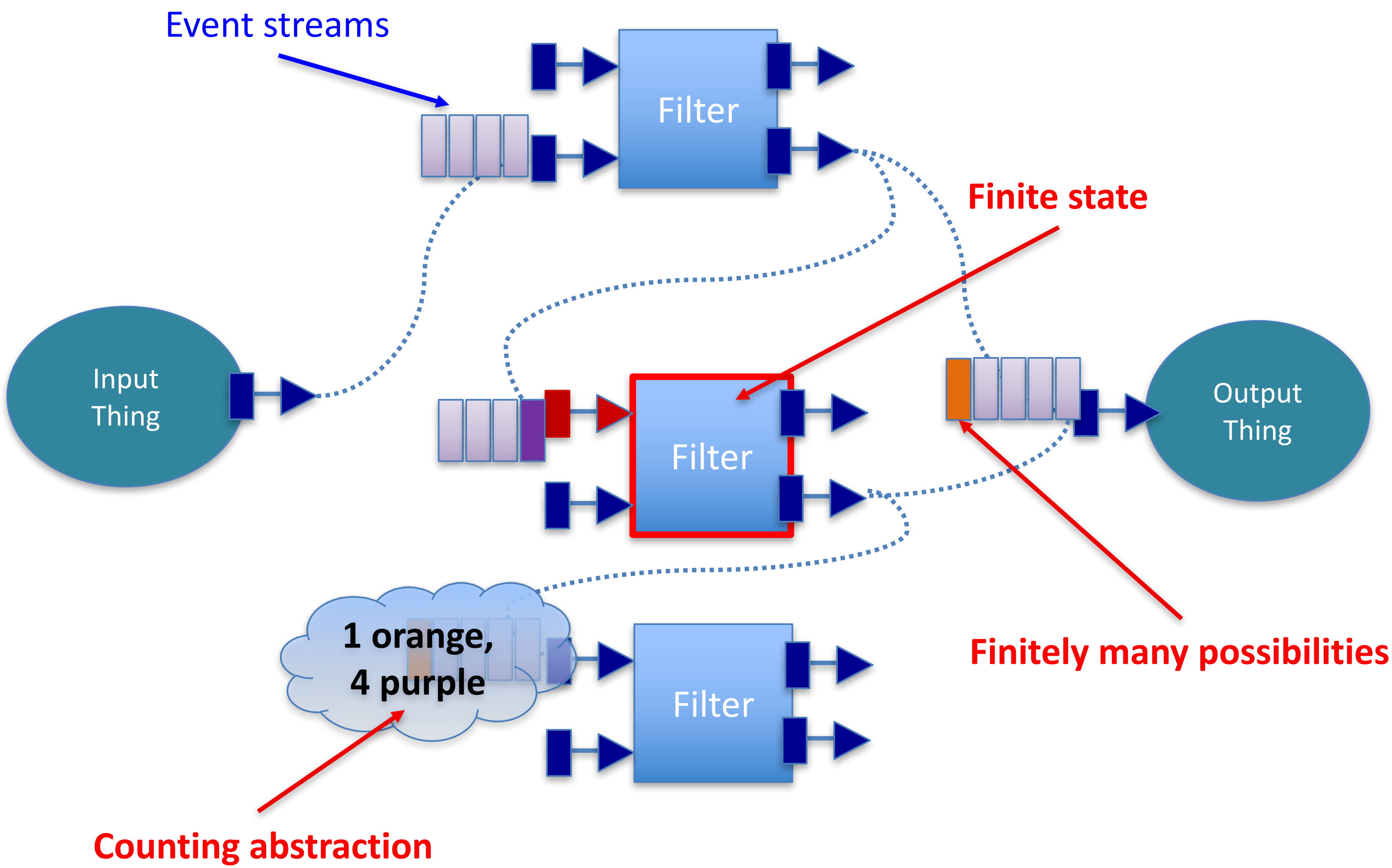
are the provenances of all events in *x* always in the set *R* along all executions of *P*?

Assumptions: Finitely many events, finite-state filters, and ordering of events in queues ignored

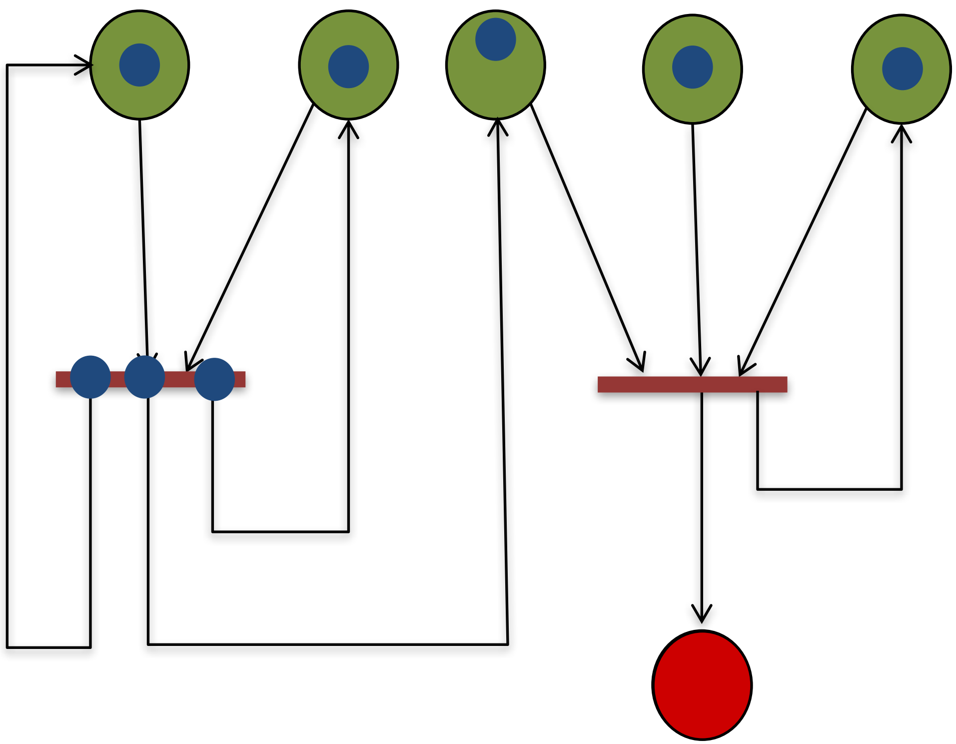
Note: The system is still infinite-state!

Basic abstraction: For each stream, each kind of event, count how many events are currently in the stream

In Each Step…


Event streams
Filter
Finite state
Input
Thing
Filter
Output
Thing
1 orange,
4 purple
Filter
Finitely many possibilities
Counting abstraction

Unbounded Events: *Petri Net*



- Finite set of places
- Finite set of transitions
- Places marked with tokens
- State: Marking
- Step: consume tokens from sources, put tokens into targets of a transition
- Defines an infinite state system

The Benefits of Petrification

Petri nets have nice decidable properties:

Coverability problem (is some place markable?) is decidable

Theorem [Rackoff,Lipton] The coverability problem for Petri nets is EXPSPACE-complete.

From ThingFlow to Nets

- A place for each filter state

- A place for each queue and each event type
  - Count how many events of each type in a queue

With provenances, we do not get a Petri net:

Unboundedly many provenances ➔

unboundedly many places

Unbounded Provenances: Automata

- Define equivalence classes w.r.t. the states of DFA for the regular set of provenances.

  The validity of the provenance property depends on states of the spec automaton, not concrete provenances.

- Define a counter for each queue, event, and state of the spec

Reduction

Program + Provenance DFA $\rightarrow_{\text{poly}}$ Petri net

- Control flow can be modeled by Petri net
- Each counter is a place in the Petri net

Provenance verification problem =
Coverability problem of Petri nets

Main Theorem

Provenance verification problem for finite-state ThingFlow programs (when ordering is ignored) is **EXPSPACE-complete**.

Linear Temporal Logic

- Provenance verification = Invariants
- Provenance linear temporal logic:

  "Whenever event in x has provenance R, eventually an event in y has provenance S"

Theorem: ProvLTL decidable for finite-state Thingflow programs (when ordering is ignored)

Analysis of ThingFlow

Two examples of decidability in special cases:

1. Provenance Analysis
   [Joint work with Roland Meyer & Zilong Wang]

2. Analyzing a single filter: Abstracting infinite-state Markov processes
   [Joint work with Sadegh Soudjani and Alessandro Abate]

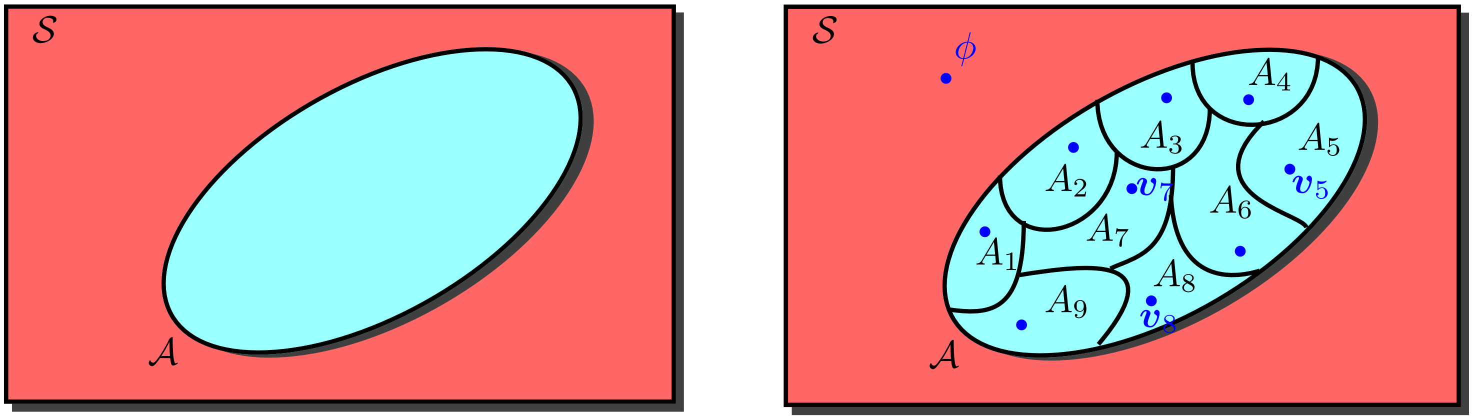
Discrete-Time Markov Process

- State space S
- Transition kernel $T(ds' \mid s) = t(s' \mid s)\, ds'$

$$T(C \mid s) = Pr[s' \in C \mid s]$$

- N-step safety problem: Given s_0, T, and a set A, find the probability that the system stays in A up to N steps
  - Can formulate as a Bellman iteration (but without any closed form)

Markov Chain Abstraction



- Finite-state Markov chain = Representatives from a partition of the infinite-state space
- Transitions:

$$P(v_i, v_j) = \int_{A_j} t(s' \mid v_i) ds'$$

Main Result

If *t(. | s)* is Lipschitz continuous with constant *h*, one can bound the probability of error between the original model and the finite-state abstraction:

$$|p_{s_0}(\mathcal{A}) - p_{v_0}(A_\delta)| \leq NLh\delta$$

Prob of staying in *A* for *N* steps

Prob of staying in abstraction of *A* for *N* steps in abstraction

N = Number of steps
L = Volume of *A*
h = Lipschitz const
δ = Diameter of abstraction

Infinite to Finite MDPs

- Bounds are *very* weak!
  - Compared to Monte Carlo simulation
- Open: Better bounds?
- Open: Verification for MDP + asynchronous concurrency?

Analysis of ThingFlow

Two examples of decidability in special cases:

1. Provenance Analysis
   [Joint work with Roland Meyer & Zilong Wang]

2. Analyzing a single filter: Abstracting infinite-state Markov processes
   [Joint work with Sadegh Soudjani and Alessandro Abate]

Open: Analysis of a Thingflow program (combining asynchrony, filters, and probabilities)

Other Open Problems

1. Parameterized reasoning
2. Real-time control
3. Fault tolerance and distribution
4. Deployment
5. Security, privacy, accountability

Conclusion

- ThingFlow = DSL for stream-processing applications for IoT systems
  - Streams & stream transformations
  - Filters & filter composition
  - Uncertainty & infinite-state
  - Asynchrony & explicit scheduling

- Many verification/analysis/tool aspects are open!

Thank You

http://www.mpi-sws.org/~rupak

ThingFlow:

https://github.com/mpi-sws-rse/thingflow-python

ThingFlow Examples:

https://github.com/mpi-sws-rse/thingflow-examples