does Task-Oriented Programming reduce the IoT development grief?

Pieter Koopman

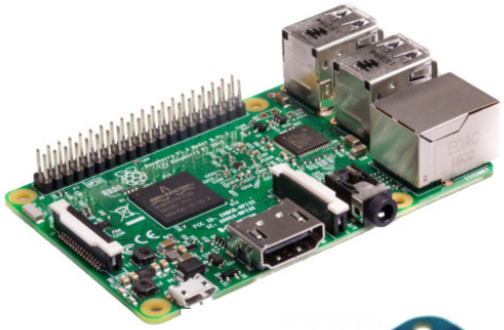
Mart Lubbers, Adrian Ramsingh, Jeremy Singer, Phil Trinder
and many others

based on ACM TIOT https://doi.org/10.1145/3572901

Radboud University

IoT devices: single-board computers vs. microcontrollers

	Raspberry Pi 3	Wemos D1 mini
price	60 €	6 €
energy	4 W	0.4 W
volatile fast memory	1,000 MB	0.05 MB
flash memory (wears)	16,000 MB	4 MB
CPU speed	1,400 MHz	80-160 MHz
word	64 bits	32 bits
WiFi	✗	✓
operating system	✓ Pi OS	✗ *



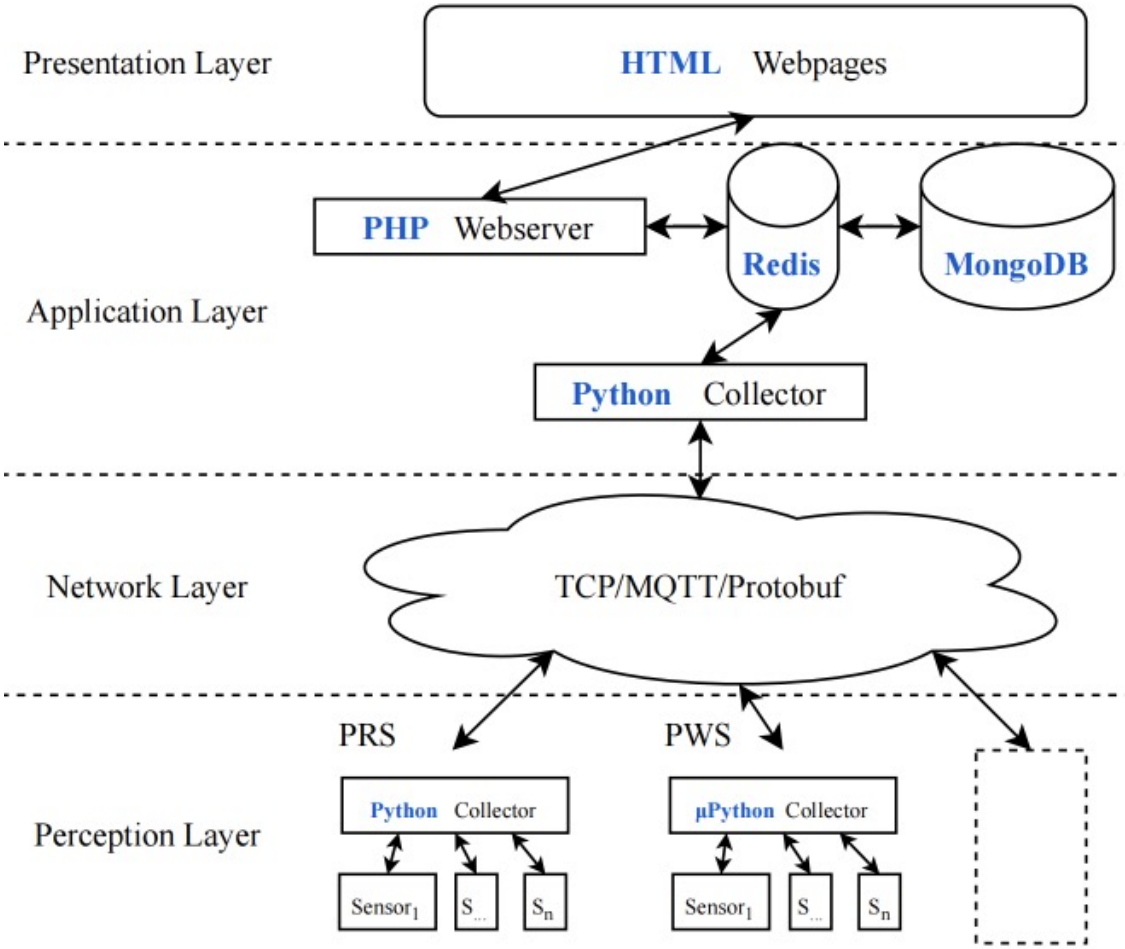
* we can use FreeRTOS

- microcontrollers are fine IoT edge devices

+ price and energy consumption are excellent, WiFi included

- memory and speed are limited, which has an impact on the software

the IoT Development Grief



- **distributed heterogenous** system
- many languages and systems involved
  - Python, MicroPython
  - TCP, MQTT, Protobuf
  - HTML, PHP, Redis, JSON,
  - MongoDB

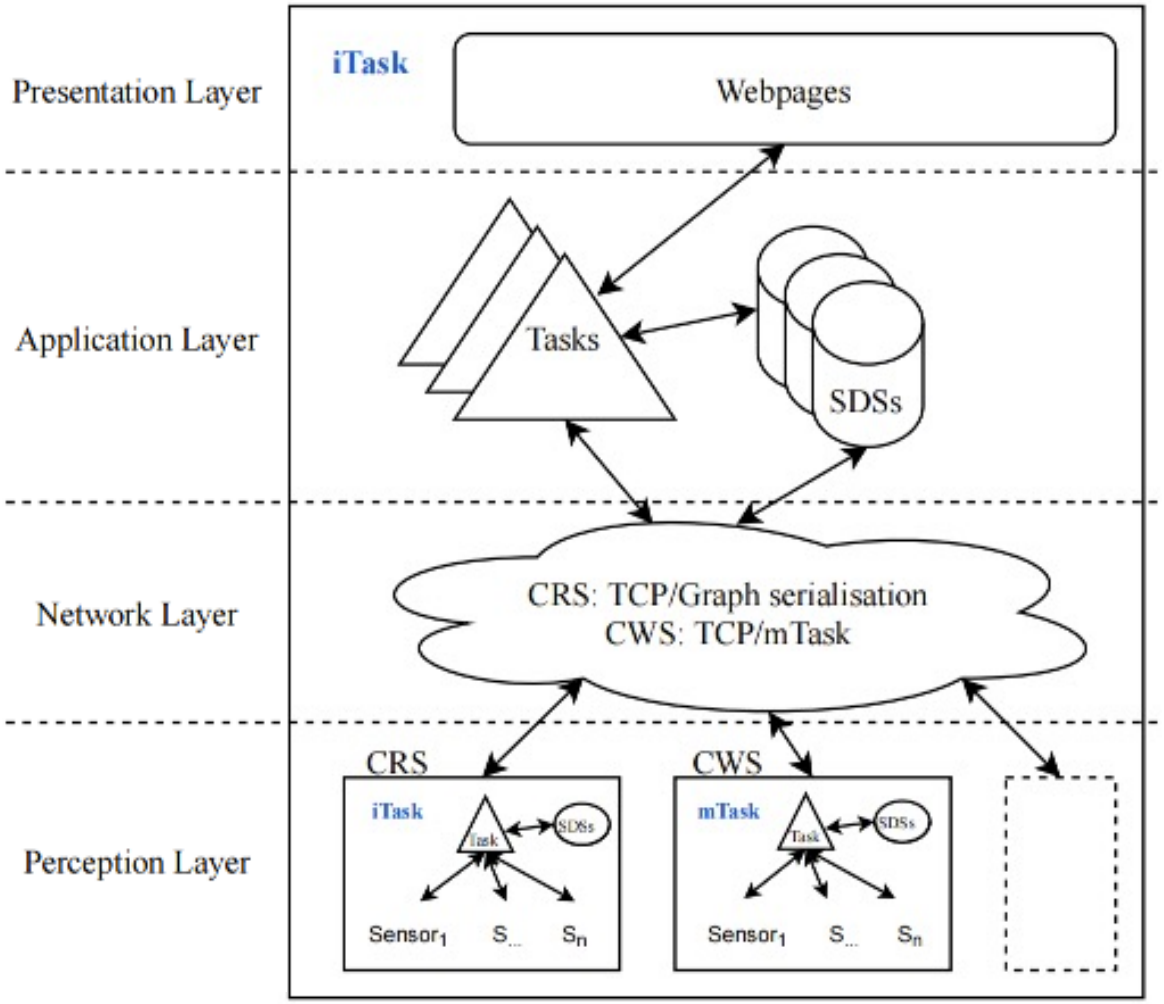
+ flexible

- semantic friction
- problems detected at runtime
- maintenance is a challenge

the tierless approach

- **tierless = use a single source to define the entire application**
  - prevents semantic friction and version problems
  - a single type system checks the entire application (for typed languages)
- tierless Task-Oriented Programming, TOP
  - focussed on tasks to be executed by machines and humans
  - **iTask** for web-page, server, database and devices, same code runs everywhere
  - **mTask** for resource-restricted devices (embedded in iTask)
- tierless web-applications
  - generate web-page, server and database from a single source
  - e.g. Links, Hop: focussed on the web-page
- tierless IoT-applications
  - generate code for network of devices from a single source
  - e.g. Potato: reactive programming for network of Raspberry Pi's

Task-Oriented Programming for the IoT



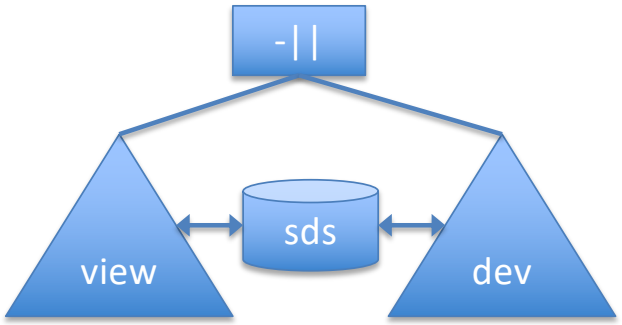
Tierless

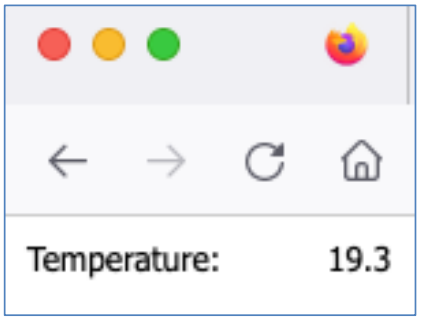
- single source for all code
- full code runs everywhere
  - except on restricted devices

+ no semantic friction
+ strong type system prevents runtime problems
+ communication and storage is generated
+ less code
+ more reliable

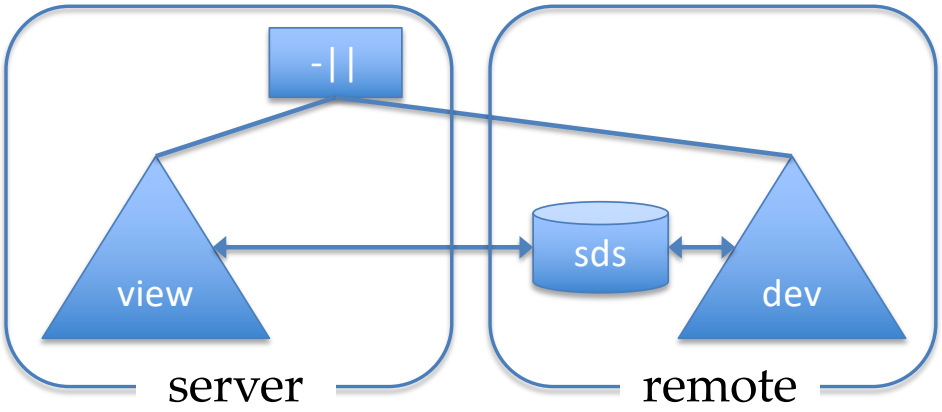
TOP by example: temperature sensor at server

```
tempSDS :: SimpleSDSLens Real
tempSDS = sharedStore "tempSDS" -273.15 // brrr

localSensor :: Task Real
localSensor =
    withDHT TempID \dht ->
    get tempSDS >>- \old ->
        devTask dht old
    -|| viewSharedInformation [] tempSDS <<@ Label "temperature"

devTask :: DHT Real -> Task Real
devTask dht old =
    temperature dht >>~ \new ->
        if (old <> new)
            (set new tempSDS >-| devTask dht new)
            (waitForTimer False 5 >-| devTask dht old)
```

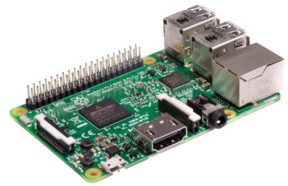




TOP by example: temperature sensor remote RPi

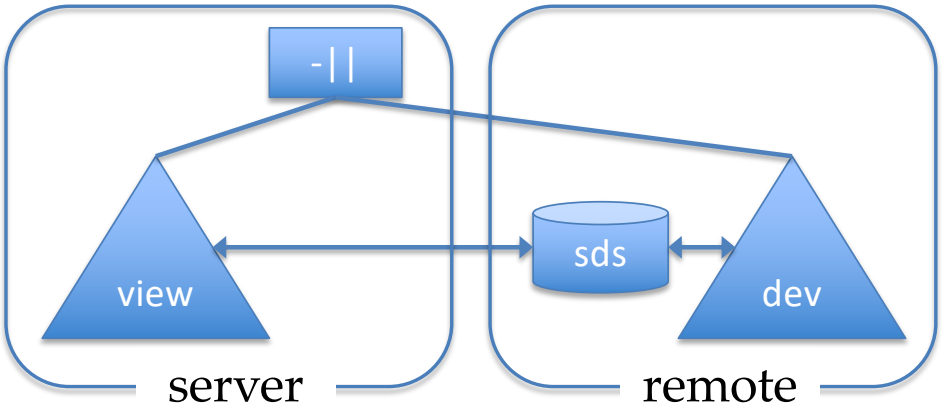



```
remoteSensor :: Task Real
remoteSensor =
    withDHT TempID \dht ->
    get tempSDS >>- \old ->
    enterInformation [] <<@ Title "device" >>? \dev ->
        asyncTask dev.domain dev.port (devTask dht old)
    -|| viewSharedInformation [] (remoteShare tempSDS dev)
        <<@ Label "temperature"
```



- remote node executes the same iTask `devTask` code

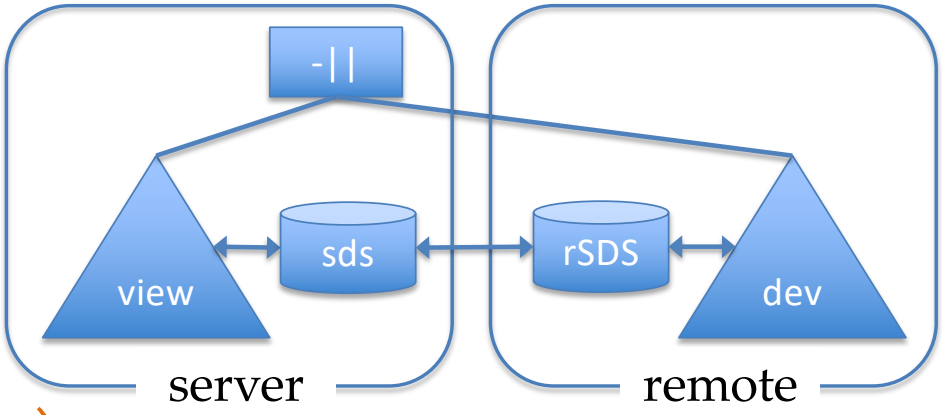
the need for mTask



- we would like to run the remote task on a microprocessor like the Wemos D1
  - more sustainable, cheaper
- challenges
  - processor is too slow
  - memory is too small (4 MiB flash and 50 KiB RAM)
  - tasks are too dynamic to store in flash (wear)
- solution
  - mTask: restricted version of iTask - first order, simple types, strict
  - compile mTask dynamically to bytecode and send it to the device
  - device runs bytecode interpreter - featherlight domain-specific OS
  - tasks are stored in RAM, with no wear of flash memory

TOP by example: mTask temperature sensor

```
mTaskSensor :: Task Real
mTaskSensor =
    enterInformation [] <<@ Title "device" >>? \dev ->
        withDevice dev (liftmTask devTask)
    -|| viewSharedInformation [] tempSDS <<@ Label "temperature"

devTask tempSDS =
  liftsds \rSDS -> tempSDS In
  DHT DHT_I2C \dht ->
  fun \measure = (\old ->
    temperature dht >>~. \new ->
      If (new  !=. old)
        (setSds rSDS new >>|. measure new)
        (measure old) In
  {main = getSds tSDS >>~. measure}
```

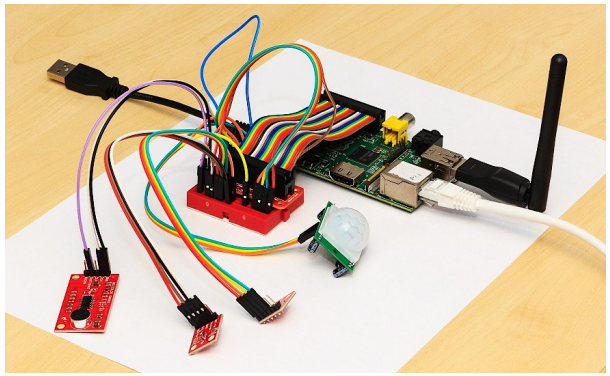


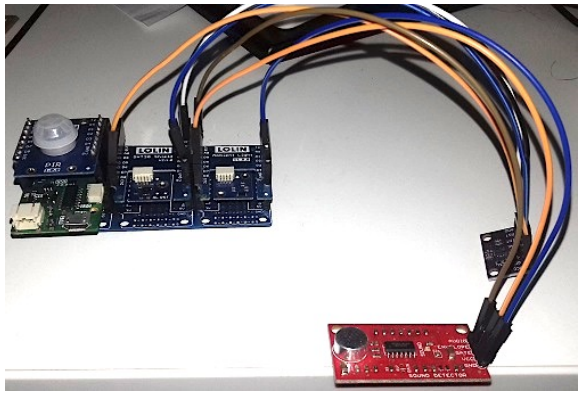
- orange code is dynamically compiled
- shipped to the selected device
- run by the mTask OS

Radboud University

University of Glasgow - smart campus sensor

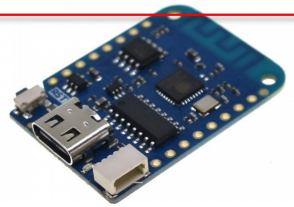
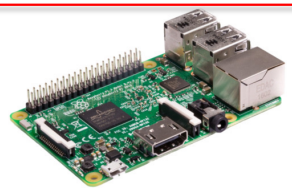
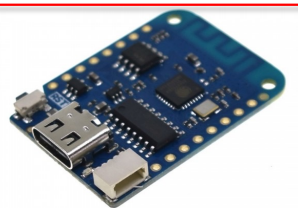
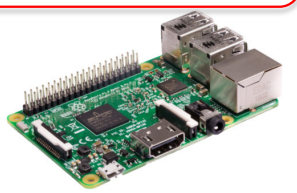
- real-world example to compare tiered and TOP code
- sensor in each room should make campus smart
  - existing prototype in Python on Raspberry Pi https://ieeexplore.ieee.org/document/7575844
  - UoG ten-year campus upgrade programme
  - apps to monitor campus use, room temperature, …
- functional requirements:
  - measures temperature, humidity and light
  - scales to 10 sensors per node
  - communication with server
  - centralised database server
  - web interface to data
  - managing and monitoring sensor nodes



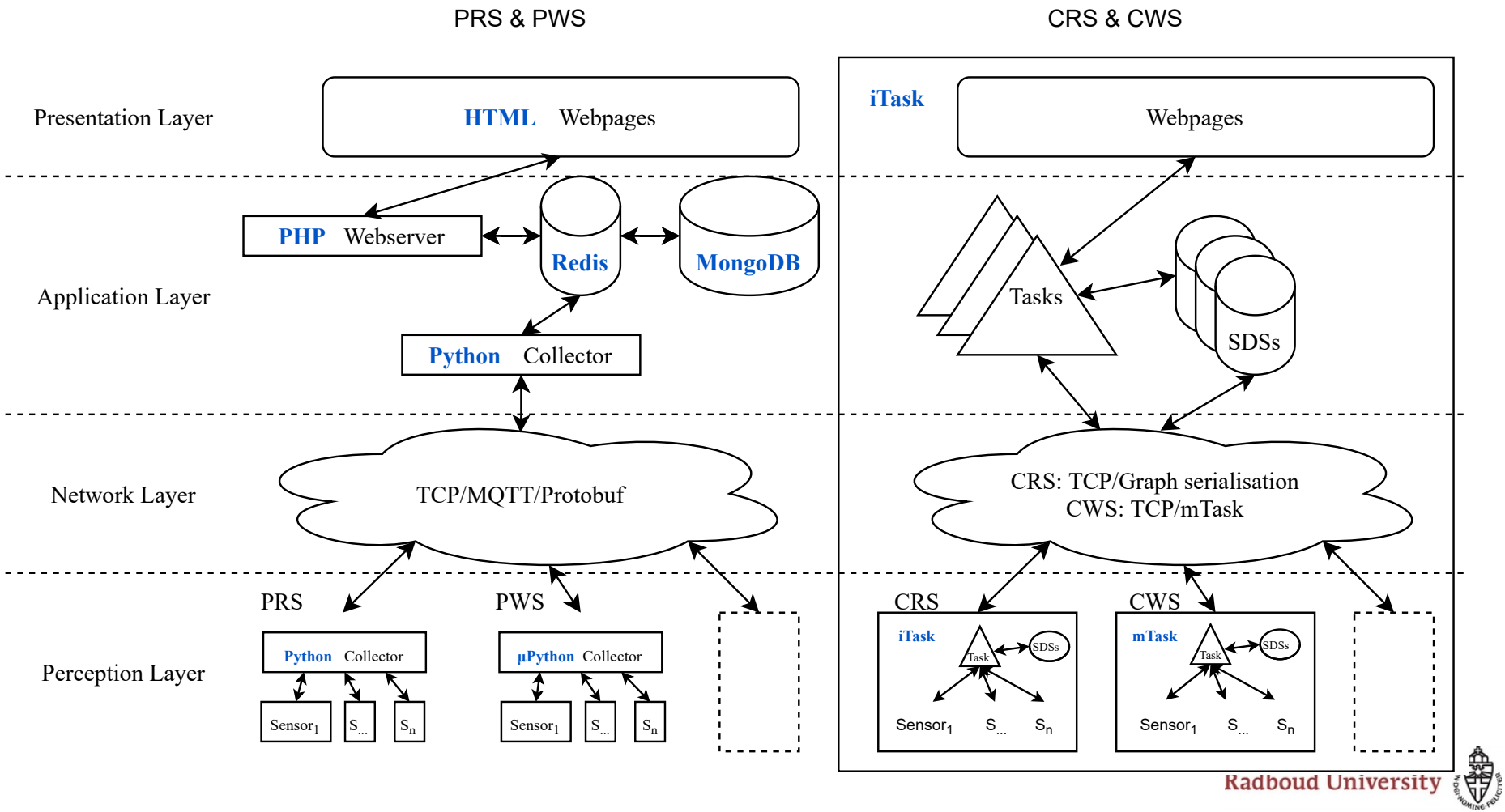


4 implementations of smart campus sensor

	tiered		tierless	
	PWS	PRS	CWS	CPS
location				
sensor node	MicroPython	Python	mTask	iTask
server	Python, JSON, HTML, PHP, Redis	Python, JSON, HTML, PHP, Redis	iTask	iTask
communication	MicroPython	Python	iTask, mTask	iTask
languages used	7	6	2	1

implementations mapped to 4-tier architecture



some results: memory residence & power consumption

	PWS	PRS	CWS	CRS
memory (KB)	20	3557	0.9	2726

	PWS	PRS	CWS	CRS
power (W)	0.2	1-2	0.2	1-2

P = Python

C = Clean

W = Wemos D1 mini

R = Raspberry Pi

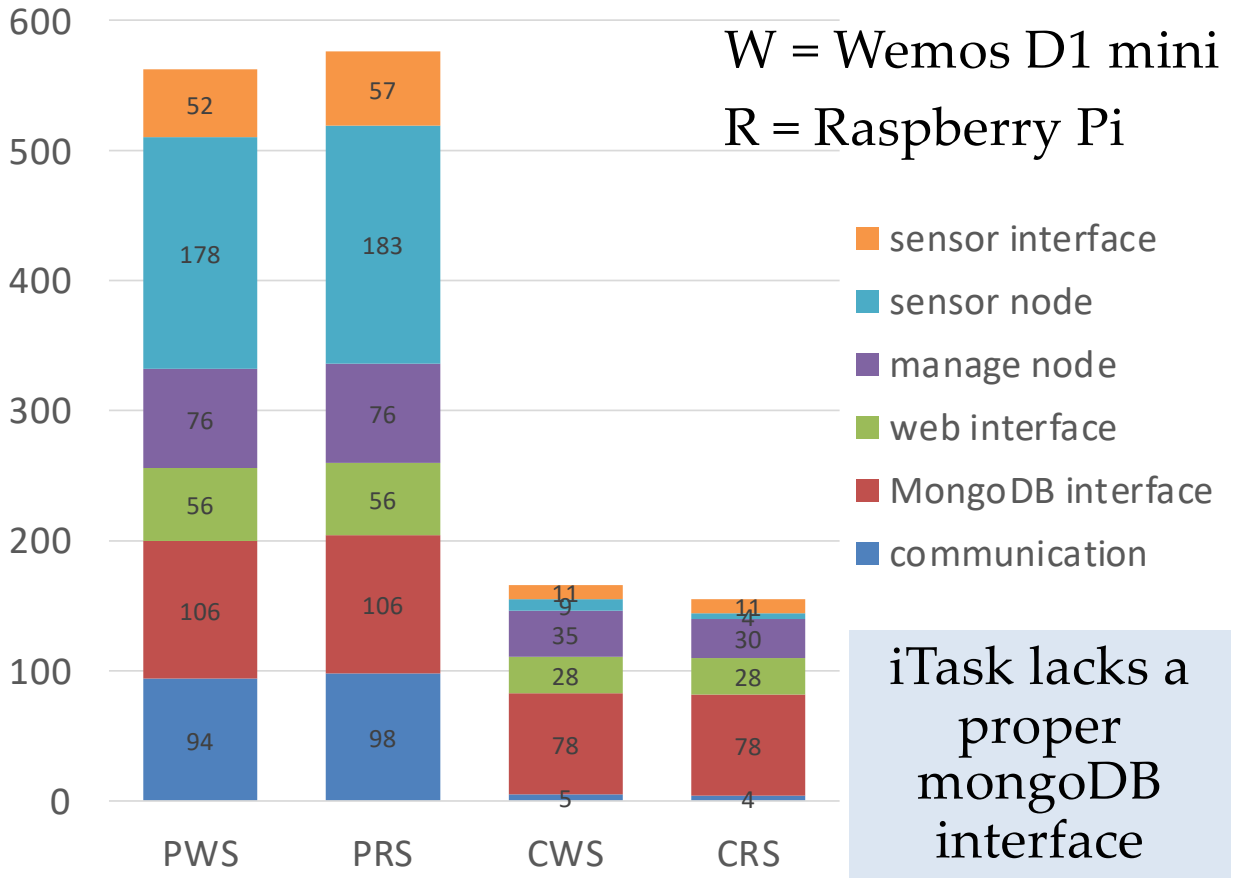
- Raspberry Pi's use much more memory, but they have more
- Clean versions have less memory residence, despite the high abstraction level
- Wemos D1's use an order of magnitude less power

some results: code size

	PWS	PRS	CWS	CRS
SLOC	562	576	166	155
files	35	38	3	3
languages	7	6	2	1

- tierless requires 70% less code
  - even less with SDS instead of DB
- using constrained nodes has limited influence on code size, increments the language count

P = Python
C = Clean
W = Wemos D1 mini
R = Raspberry Pi



iTask lacks a proper mongoDB interface

could tierless IoT programming be more reliable than tiered?

- tierless requires less programming languages = less semantic friction
- tierless requires less interoperation = less errors
- communication is generated in the tierless system = no code needed
- TOP offers convenient high-level abstractions for tierless programming

advantages

- the tierless approach offers type-safety
- failure management is much easier in the tierless approach
- maintainability of the tierless systems is higher

disadvantages

- harder to find skilled programmers and support for tierless IoT programming

resource-rich/constrained sensor nodes

- large parts of the code (server + web-interface) are unaffected by this choice
- constrained sensor nodes have advantages
  - more sustainable (less energy and resources)
  - cheaper
  - bare metal control (GPIO control, timing, energy use, memory use)
- constrained sensor nodes come with a cost
  - tailored programming language (microPython, mTask, BIT, uLisp, Arduino C++, …)
  - one additional language in the system
  - this language has restrictions w.r.t. general-purpose programming languages
  - requires proper design choices
  - limited amount of extra code (less than 10% in our example)

does TOP reduce the IoT development grief?

observed advantages of the tierless TOP approach for IoT programming

- type checking of the entire system
- generation boilerplate code
- single paradigm
- significantly less code
- easy integration of restricted nodes (microprocessors): mTask
- this real-life example is too simple to see the full benefits of the TOP approach

YES

challenge

- finding support and programmers is harder (like every new software)
- warning N = 1 for code measurements
- no reason why advantages are not generally valid