Sustainable IoT programming

IoT = Internet of Things

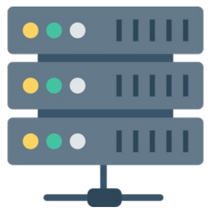
Mart Lubbers - **Pieter Koopman**

Radboud University

layered IoT architecture

presentation layer

application layer



sometimes split in an application and a business layer

network layer

edge computing

perception layer

the smart sensors on the mills

vital software challenges
also holds for Internet of Things programming

- **reliable**
  - functionality, performance, security, usability, ..
- **maintainable & evolvable**
  - maintainable: fix problem and small adaptations to changing environment
  - evolvable: ability to easily accommodate future changes gradually
- **efficient engineering**
  - effective tools and how to use them properly
- **sustainable**
  - energy-efficiency of IoT system (previous SusTrainable summer school)
  - efficiency of construction, maintenance and evolution

up to 90% of project costs

adapted from the versen.nl manifesto

first laws of software quality

we need concise high quality code !

static typing spots error before they occur

$$E = mc^2$$

$$\textbf{E}\text{rrors} = (\textbf{m}\text{ore } \textbf{c}\text{ode})^2$$

Chet Haase (Google)

If something can go wrong, it will

Edward A. Murphy

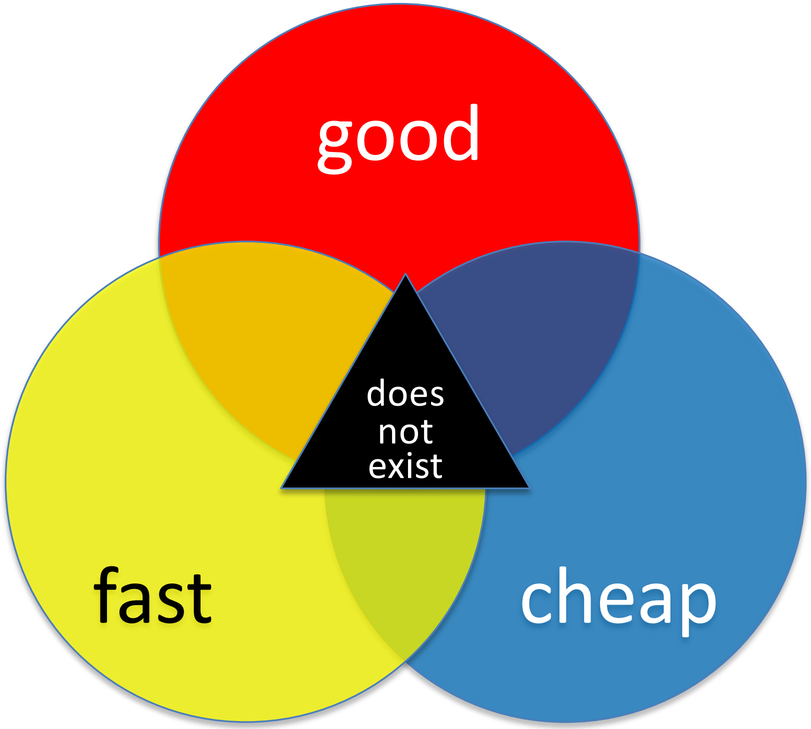
lowering quality lengthens development time

Ward Cunningham

Programs must be written for people to read,
and only incidentally for machines to execute.

Abelson and Sussman

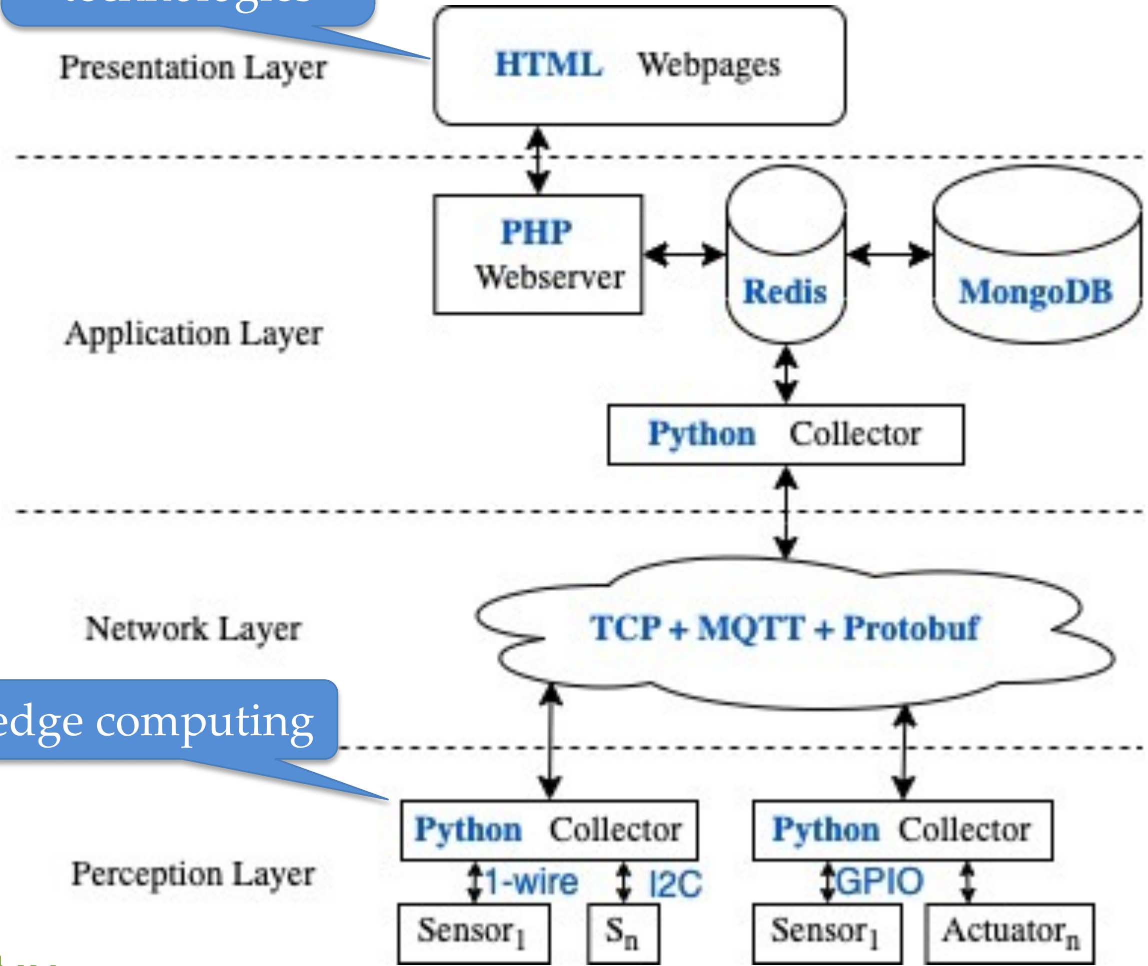
project management triangle



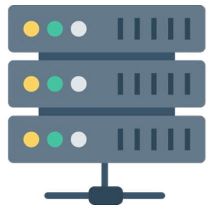
good, fast and cheap; pick any two you like

the IoT Development Grief

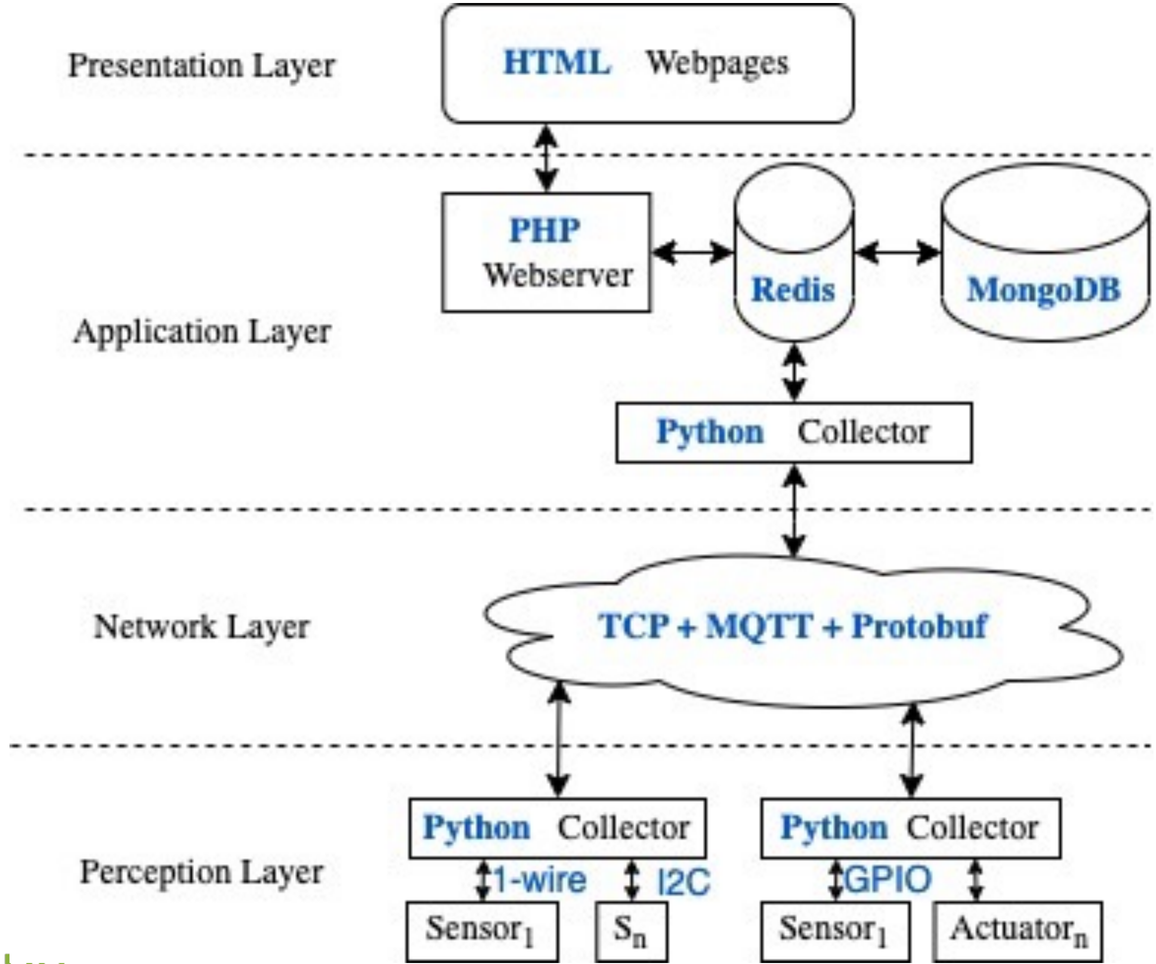
example technologies



edge computing



the IoT Development Grief



- **distributed heterogeneous** system
- many languages and protocols
  - Python, PHP
  - TCP, MQTT, Protobuf
  - HTML, JSON,
  - Redis, MongoDB
  - I2C, 1-Wire, GPIO

+ flexible

- complex
- semantic friction
- problems detected at runtime
- maintenance is very hard

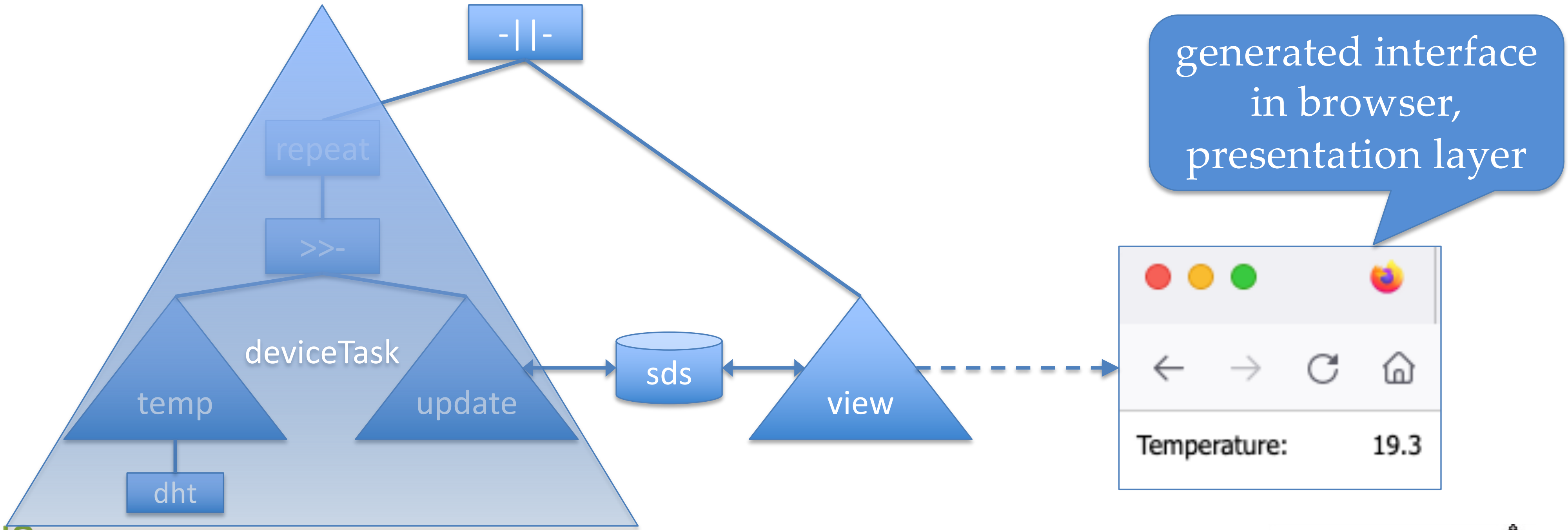
the tierless approach

- **tierless = use a single source to define the entire application**
  - software for all components and their interaction generated from this single source
  - type-system checks the entire application (prevents run-time errors)
  - prevents semantic friction and version problems
  - also used for websites: Hop, Links, ScalaLoci, .., Potato

- tierless Task-Oriented Programming, TOP
  - focussed on **tasks** to be executed by machines and humans
    web-pages and other interactions are determined by the current tasks to do
  - **iTask** for web-pages, server, and database,
    the same code is executed in the browser and on the server and clients
    - ➢ built on top of and inheriting all advantages of **functional programming**
  - **mTask** for small IoT devices

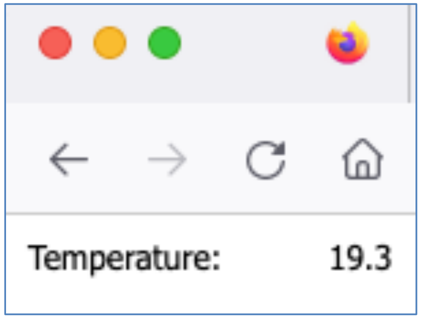
tasks by example

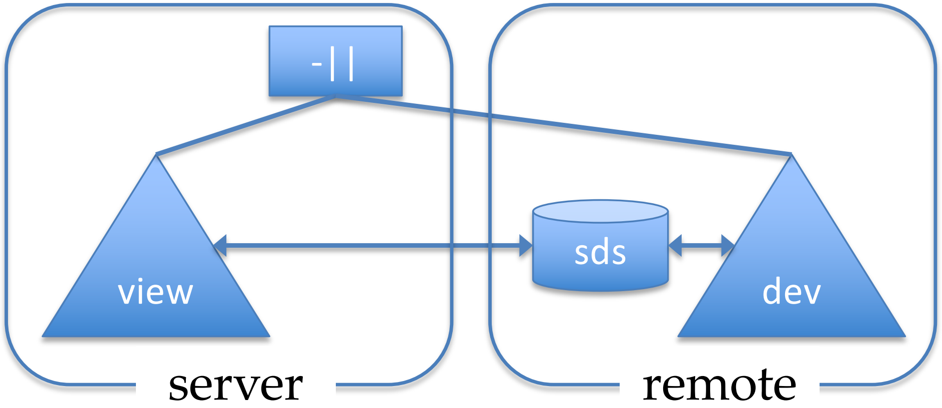
- basic tasks: read temperature sensor, update SDS, view/edit SDS
- Shared Data Source, SDS
- task composition: sequence, repeat, parallel (combinators)



SUS Trainable

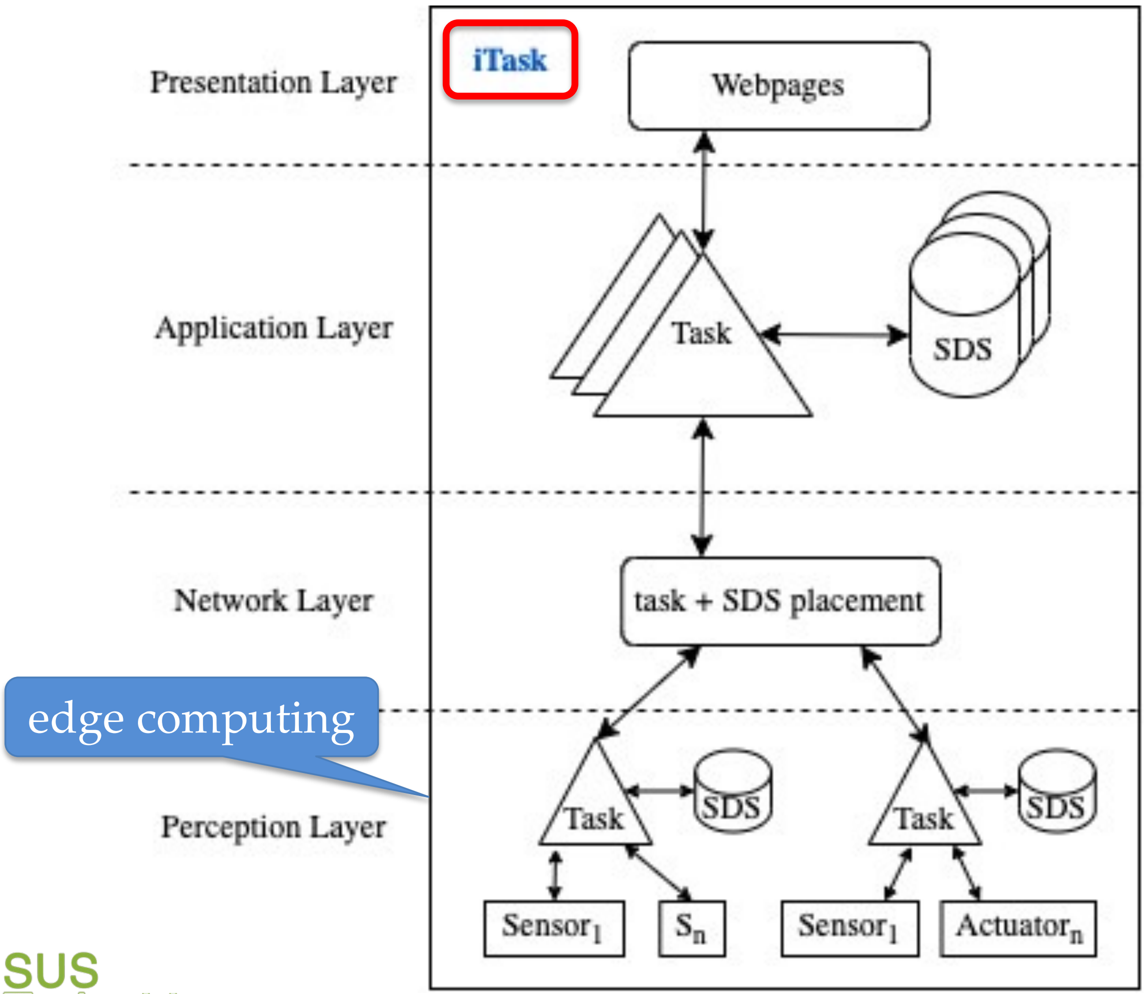
TOP by example: temperature sensor remote RPi





- we only indicate where tasks are executed
- code for remote task is unaffected
- an easy way to select the remote computer is via an iTask editor, **dynamic** placing of the device task

Task-Oriented Programming for the IoT



Tierless

- single source for all code

+type system prevents errors

+no semantic friction

+**communication and storage are generated**

+more reliable

+easier to maintain

+less code

- monolithic

good

microcontrollers require less energy and resources

tasks on restricted hardware

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	2,000 MB	0.05 MB
flash memory (wears)	32,000 MB	4 MB
CPU speed	1,400 MHz	80 MHz
Word size	64 bits	32 bits
WiFi	✗	✓
operating system	✓ Pi OS	✗ *

* we can use FreeRTOS

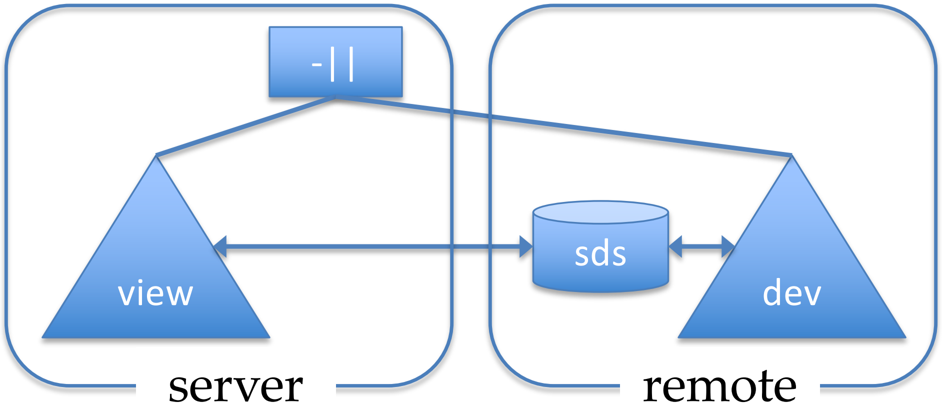
- microcontrollers are fine IoT edge devices

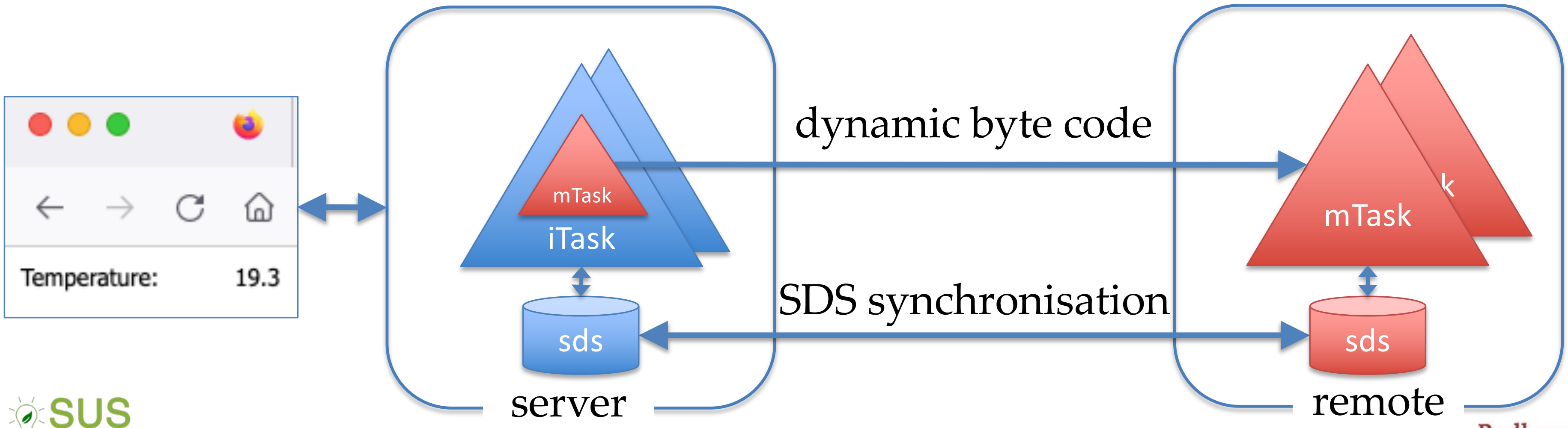
+ price and energy consumption are excellent, Wi-Fi included

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

the need for mTask

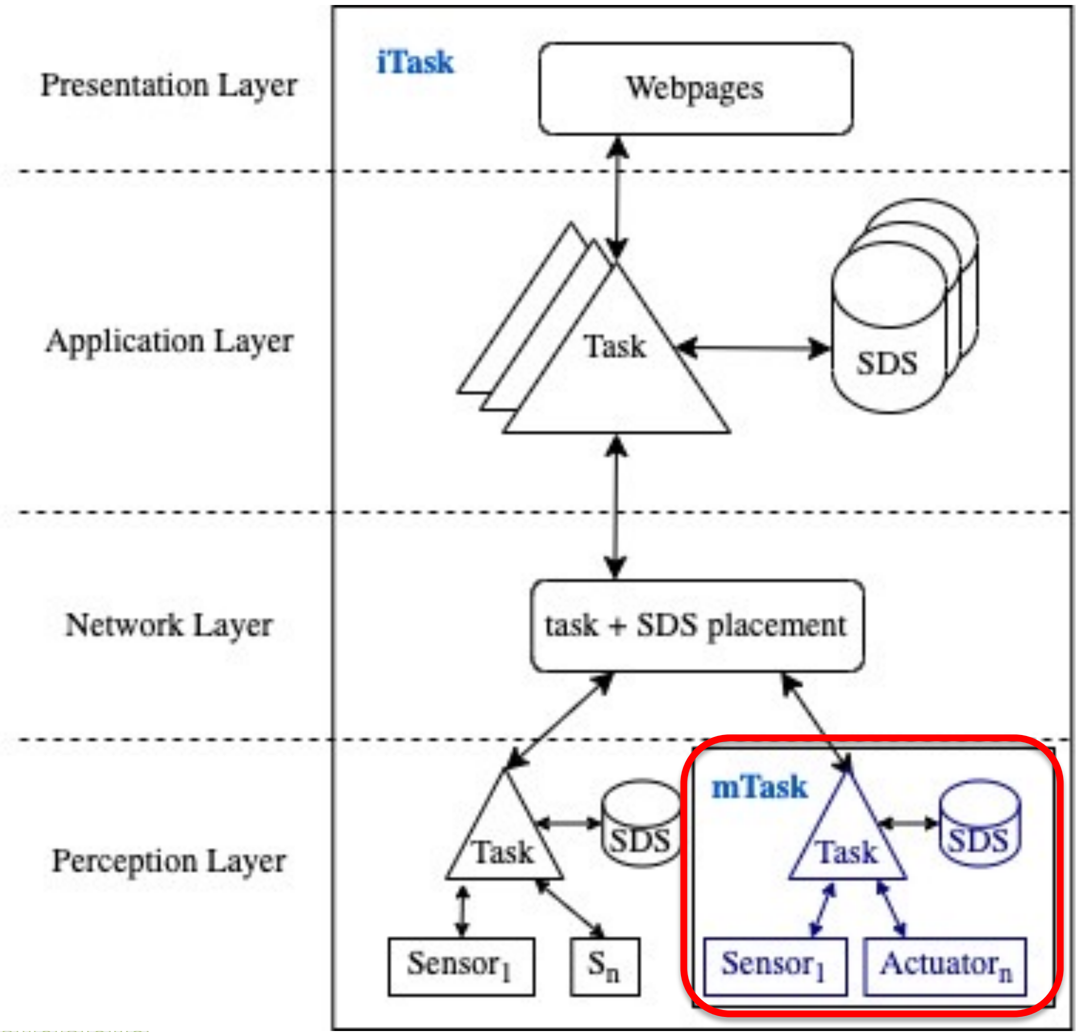
- remote task on a device like the Wemos D1
- challenge: limited resources
  - processor is too slow
  - memory is too small (4 MB flash and 50 KB RAM)
  - tasks are too dynamic to store in flash (wear)
- solution: mTask: restricted version of iTask





byte code interpreter + tiny OS

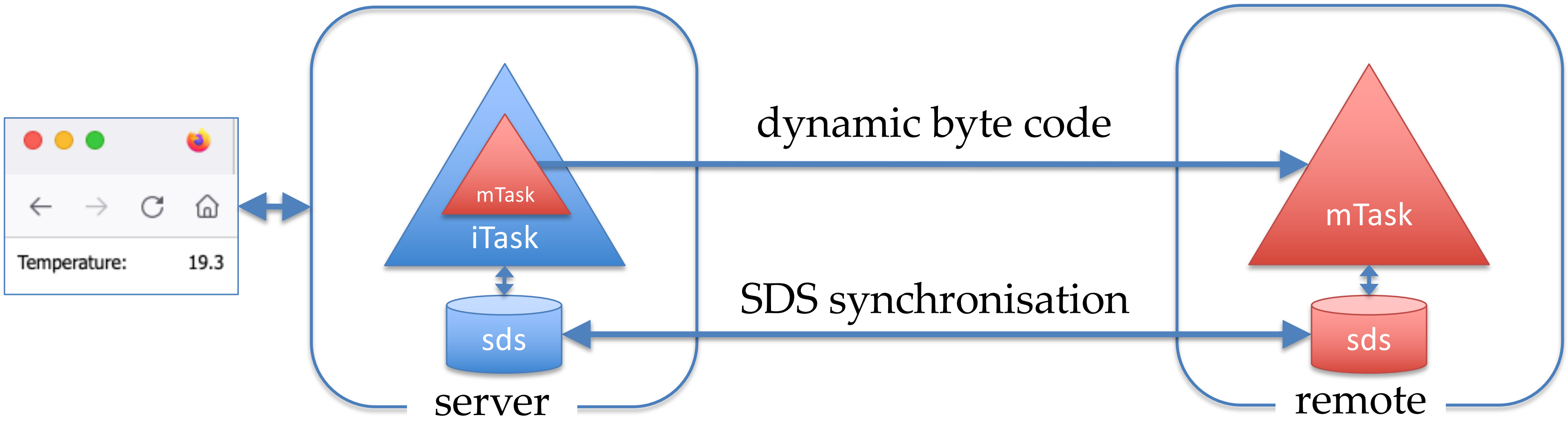
Task-Oriented Programming for the IoT on restricted devices



mTask architecture

- single source for all code
  - \+ typed: no runtime errors
  - \+ no version problems
  - \+ no semantic friction
- a separate part for edge node
  - runtime compiled to bytecode
  - runtime shipment to device
  - bytecode interpreter on device featherlight **domain-specific OS**
  - tasks are stored in RAM, prevents wear of flash memory

mTask architecture



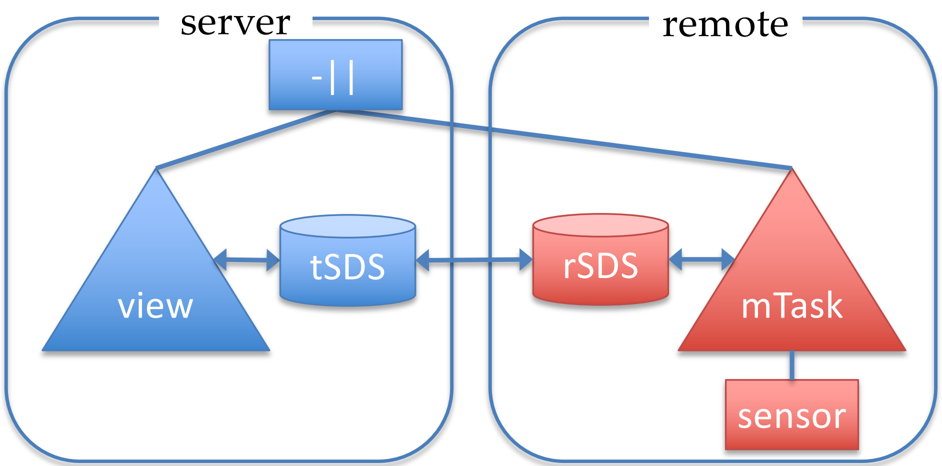
- mTask is integrated in iTask
- same high-level single source
- same static quality guarantees
- automatic SDS synchronisation
- storage and GUI generated

- program is shipped dynamically
- no maintenance problems, remote program is always up to date
- mTask OS optimizes task execution
- communication is generated

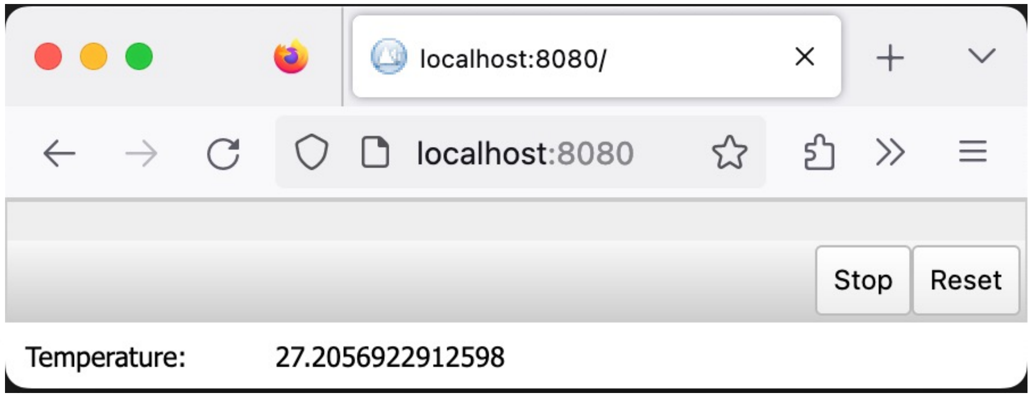
iTask – mTask example

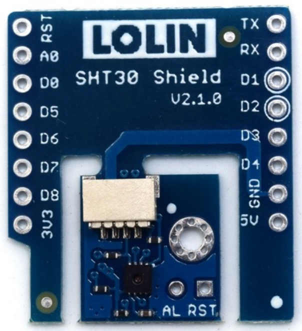
TOP for IoT by example

remote temperature sensor

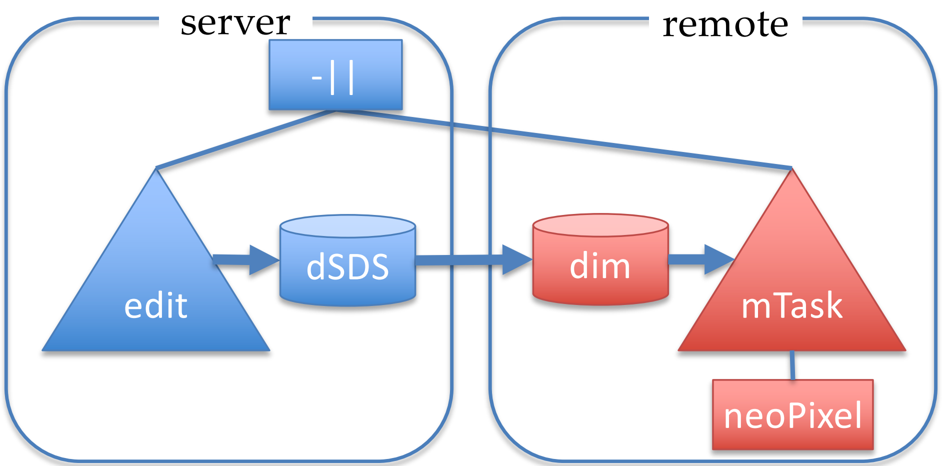



```
tSDS = sharedStore "tSDS" -273.15

Start w = doTasks combinedTask w

combinedTask =
  runMTask tempMTask -||
  (Label "Temperature" @>> viewSharedInformation [] tSDS)

tempMTask =
  dht dhtWemosSHT30Shield \sensor->
  lowerSds \rSDS = tSDS In
  {main = rpeat (
      temperature sensor >>~. \t.
      setSds rSDS t >>|.
      delay delta)
  }
delta = ms 200
```

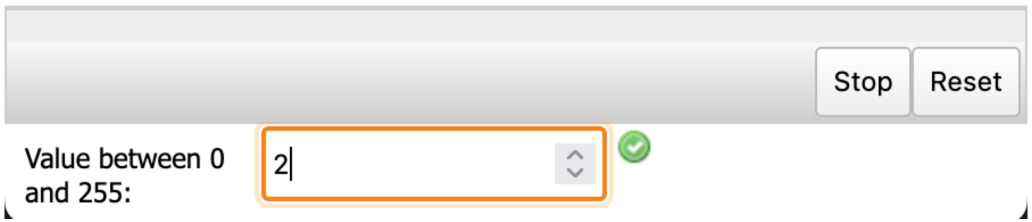




controlling a neoPixel LED




```
dimSDS = sharedStore "dim" 0

Start w = doTasks combinedTask w

combinedTask =
  runMTask color -||
  (updateSharedInformation [] dimSDS <<@ Label "Value between 0 and 255")

color =
   neopixel neopixelWemosRGBLEDShield \neo->
   lowerSds \dim = dimSDS In
   {main = rpeat (getSds dim
                 >>~. \d->setPixelColor neo ledNum d d d)
   }

ledNum = lit 0
```

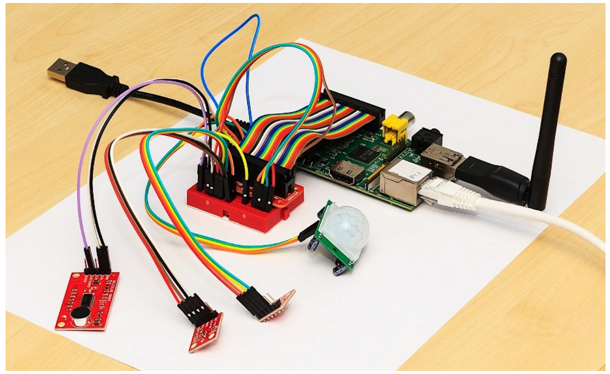


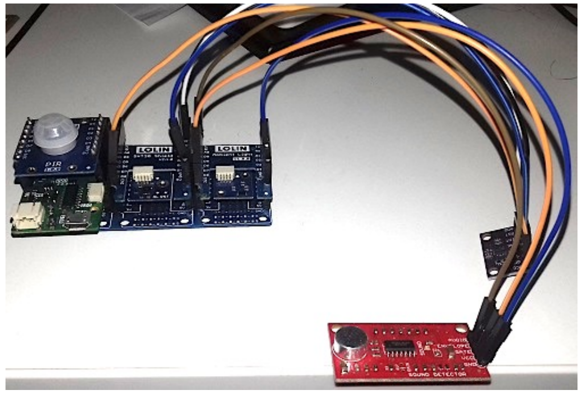
Comparing code size and paradigms

Development and maintenance

case study: University of Glasgow - smart campus sensor

- real-world example to compare tiered and TOP code
- sensor in each room to make campus smart
  - UoG ten-year campus upgrade programme
  - apps to monitor campus use, room temperature, …
  - existing prototype in Python on Raspberry Pi https://ieeexplore.ieee.org/document/7575844
- 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	
sensor node	MicroPython	Python	mTask	iTask
server + data storage + communication	Python, JSON, Redis, MongoDB, HTML, PHP	Python, JSON, Redis, MongoDB, HTML, PHP	iTask	iTask
languages used	7	6	2	1

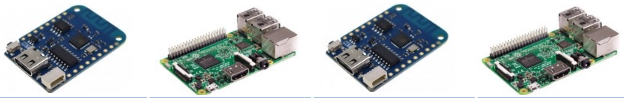



SUS Trainable

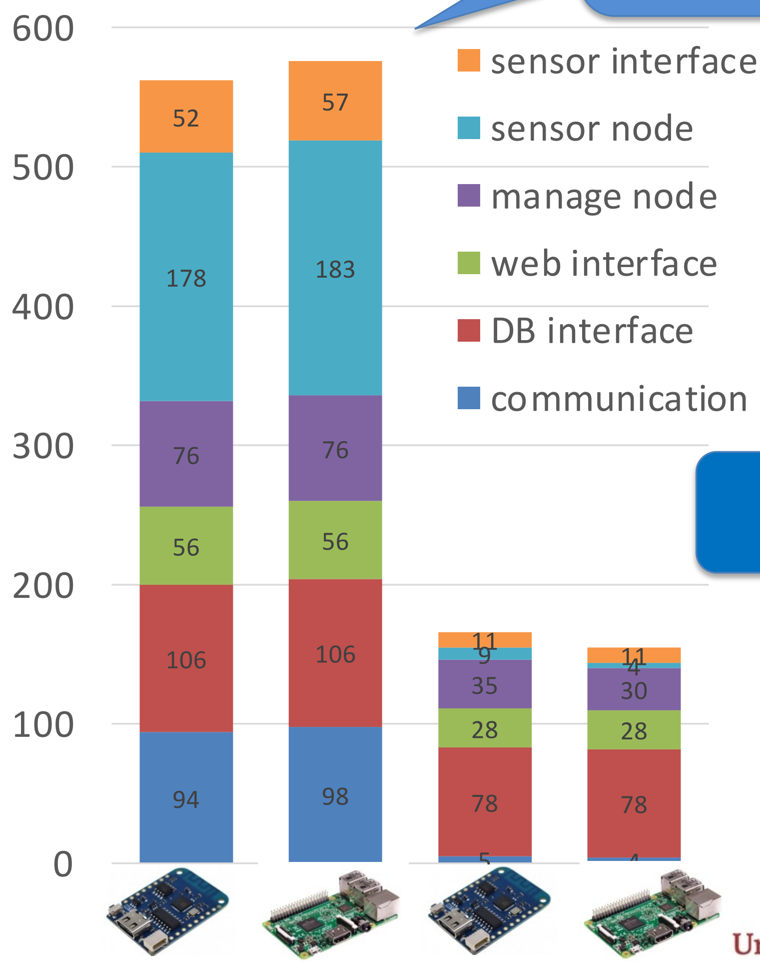
Radboud University

smart campus sensor - code size

tierless:
often 90% reduction
on average 73%



SLOC	PWS	PRS	CWS	CRS
	tiered		tierless	
sensor node	178	183	9	4
sensor interface	52	57	11	11
communication	94	98	5	4
web interface	56	56	28	28
DB interface	106	106	87	87
swap DB to SDS	*-*	*-*	*8*	*8*
manage node	76	76	5	4
total	**562**	**576**	**166**	**155**
files	35	38	3	5



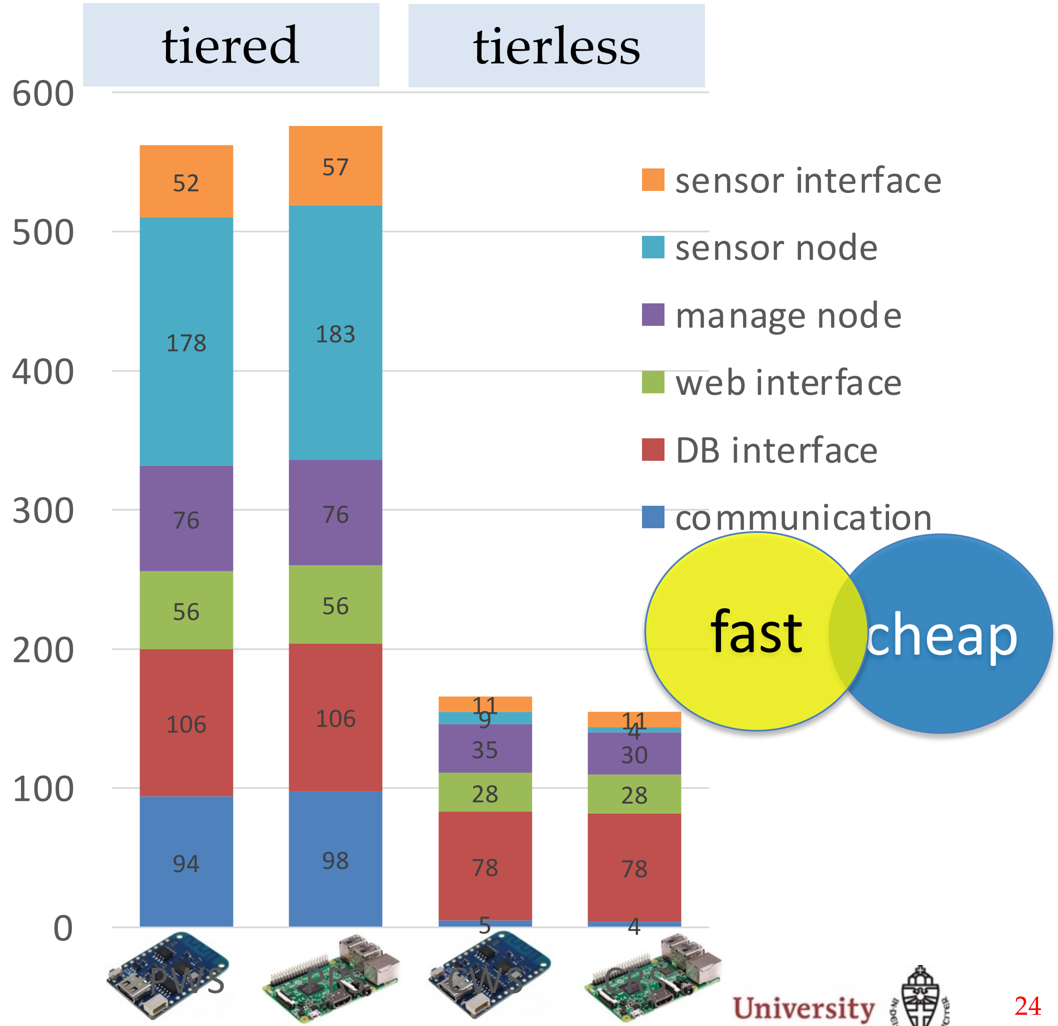
$E = mc^2$

SOS Trainable

University

smart campus sensor - comparison

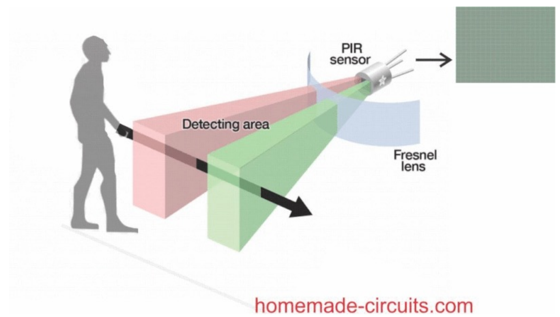
- restricted hardware:
  - additional language and decisions
  - limited additional code
- mTask ships tasks dynamically, static allocation in the tiered approach
- tierless is a single program
  - checked by the compiler
- tierless requires less languages
- tierless requires less paradigms
- tierless requires less code
- hence, tierless is better maintainable



green computing (previous summer school)

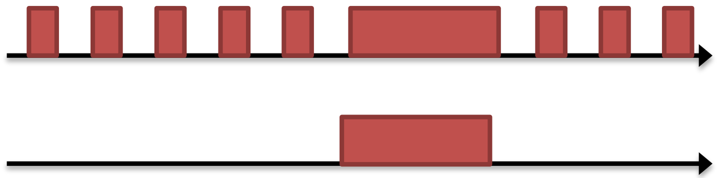
good

- **automatic sleeping**
  - microcontrollers have sleep modes to save energy,
    even light sleep saves 99,7% energy
  - the mTask OS on the device assigns an execution region to each task
  - regions based on expected change. e.g., temperature <0, 2000> ms
  - without urgent tasks the devices takes a nap,
    whenever awake it executes all tasks that cannot be delayed to next round
- **interrupt handling**
  - sensor wakes up or interrupts microcontroller
  - less energy needed than polling
  - fewer events missed



polling

interrupt



other aspects of tiered / tierless programming

tiered (Python and friends)

- reliable
  - finished system had some errors
- maintenance
  - updates are pretty tricky
- evolution
  - hard
  - fail-safe system was too much work
- efficient engineering
  - wide variety of tools available
  - many courses
  - wide community (e.g. stack overflow)

tierless (iTask + mTask)

- reliable
  - no errors found after the type check and tests
- maintenance
  - update the single source and recompile
- evolution
  - much easier
  - fail-safe system in a few lines of code
- efficient engineering
  - all benefits of pure functional programming
  - a single TOP implementation
  - support by a few friendly people and companies

the bad and the ugly

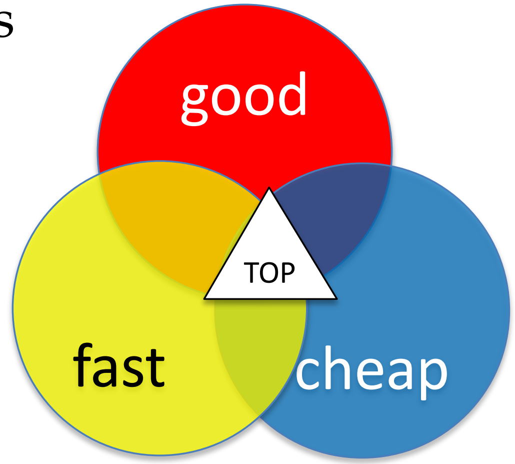
- we have to learn a new paradigm
  - embedded in functional programming language Clean
  - iTask for server and fast devices
  - mTask for restricted devices
  - *this is not yet another Python variant*

- TOP is not yet mainstream
  - limited courses available
  - experienced programmers are hard to find
  - no help on stackoverflow.com, nor on ChatGPT

  - but there is cloogle.org, clean-lang.org, top-software.nl, nitrile, Eastwood

sustainable IoT programming

- IoT programming is challenging: distributed & heterogenous
- a tierless approach simplifies development and maintenance
  - static types and code generation prevent runtime errors
  - single source language prevents semantic friction
- TOP provides concise logic, typically 90% less code
  - only 10% to 25% of the code
  - $E = mc^2$: less errors
- restricted devices make the IoT greener, but add challenges
  - limited processing, tiny amounts of memory
  - mTask integrates seamlessly with iTask and controls restricted devices



10^{-2} to 10^{-4} energy use

this gives us motivation to study

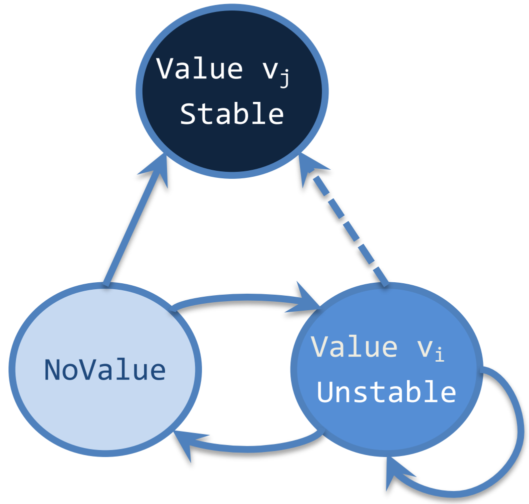
Task-Oriented Programming

iTask

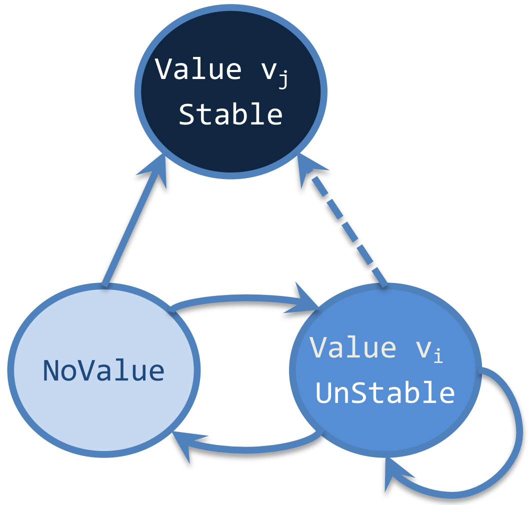
- embedded in the functional programming language Clean
  - https://clean-lang.org/
  - dedicated search engine: https://cloogle.org/

- tasks are repeated until they produce a stable value, or become obsolete
- all tasks produces a task value

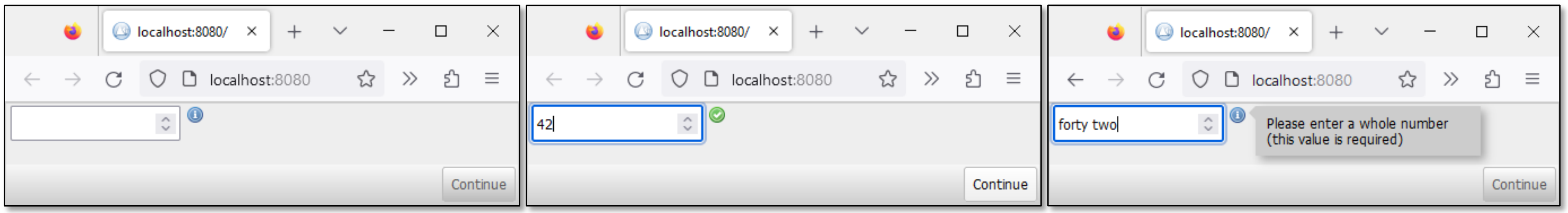
```
:: TaskValue a = NoValue | Value a Stability
:: Stability :== Bool
```



Interactive tasks




```
module example
import iTasks, StdEnv

Start :: *World -> *World
Start world = doTasks task world

task :: Task Int
task = enterInformation [] >>? \i -> return i
```



iTask versus mTask

- 'normal' embedded DSL in Clean
  - iTask library is just a set of functions
  - fully integrated in Clean
  - single view: execution
  - generates a web-server as GUI
  - error messages in terms of functions
- Full-fledged pure FP language
  - implements Task Oriented Programming
  - referential transparency
  - lazy evaluation
  - higher-order functions
  - fancy datatypes
  - high hardware requirements

- class-based DSL in Clean
  - mTask library is just a set of classes
  - fully integrated in Clean and iTask
  - multiple views possible
  - no GUI, but interaction with peripherals
  - error messages in terms of classes
- limited pure FP language
  - implements Task-Oriented Programming
  - referential transparency
  - strict evaluation
  - first-order functions
  - simple datatypes
  - runs on restricted hardware

functions in iTask

- our `runMTask` is made for the occasion
- type `BCInterpret` select evaluation of mTask

```
runMTask :: ((Main (BCInterpret (TaskValue u))))->Task () | type u
runMTask mTask = enterDeviceInfo
  >>? \spec->withDevice spec (\dev->liftmTask mTask dev)
  >>* [ OnAction (Action "Stop") (always (return ()))
      , OnAction (Action "Reset") (always (runMTask mTask))
      ]
```

combine this with other iTasks

- `enterDeviceInfo` ask user for device to run task
- `liftmTask` compiles and dynamically ships mTask to indicated device
- `withDevice` integrates the device in the iTask system

functions in mTask: only update on changed temperature

- idea **`fun \`***`name`* **`= (\`***`arg`* **`->`** *`body`* **`In`** *`main`***`)`**

```
tempMTask2 =
  dht dhtWemosSHT30Shield \sensor->
  lowerSds \rSDS = tSDS In
  fun \measure = (\old ->
    temperature sensor >>*.
         [IfValue (\new -> new  !=. old)
                 (\new.setSds rSDS new >>|. measure new)]) In
    {main = getSds rSDS >>~. measure}
```

- always exactly one function argument, `()`, `x`, `(x,y)`, ..
- do not forget the \'s and `In`
- define any number of functions you need

multiple functions with multiple arguments

```
blinkTask = neopixel neopixelWemosRGBLEDShield \neo ->
  fun \b2i = (\b -> If b level off) In
  fun \blink = (\(led, s, d) ->
                 setPixelColor neo led (b2i s) (b2i s) (b2i s)
           >>|. delay d
           >>|. blink (led, Not s, d)) In
    {main = blink (lit 0, false, delta) .||.
               blink (lit 1, false, delta *. lit 2)}
off   = lit 0
level = lit 10
delta = ms 500
```

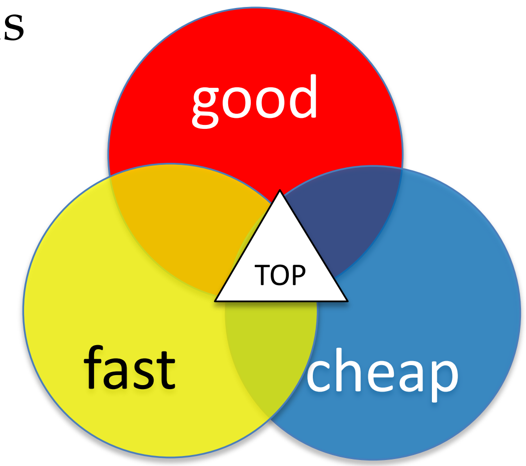
delay is not blocking

task composition cloogle.org

<table>
<tr><th></th><th>iTask</th><th>mTask</th></tr>
<tr><td>• sequential</td><td></td><td></td></tr>
<tr><td>▪ step, with list of continuations</td><td><code>>>*</code></td><td><code>>>*.</code></td></tr>
<tr><td>▪ on value (stable or unstable)</td><td><code>>>~</code></td><td><code>>>~.</code></td></tr>
<tr><td>▪ on stable value</td><td><code>>>-</code></td><td><code>>>=.</code></td></tr>
<tr><td>▪ on stable value, or value with user input</td><td><code>>>?</code></td><td></td></tr>
<tr><td>▪ on stable value, ignore result</td><td><code>>-|</code></td><td><code>>>|.</code></td></tr>
<tr><td>• parallel</td><td></td><td></td></tr>
<tr><td>▪ or: results are disjunctively combined</td><td><code>-||-</code></td><td><code>.||.</code></td></tr>
<tr><td>▪ and: results are conjunctively combined</td><td><code>-&&-</code></td><td><code>.&&.</code></td></tr>
<tr><td>▪ use left result</td><td><code>-||</code></td><td></td></tr>
<tr><td>▪ use right result</td><td><code>||-</code></td><td></td></tr>
<tr><td>• convert plain value to task</td><td><code>return</code></td><td><code>rtrn</code></td></tr>
</table>

conclusion: sustainable IoT programming

- IoT programming is challenging: distributed & heterogenous
- Task-Oriented Programming is not difficult nor weird
- **reliable**
  - strong typing, single source, single paradigm
- **maintainable & evolvable**
  - concise single source
  - storage, GUI and communication derived from types
- **efficient engineering**
  - single strongly typed and concise source
- **sustainable**
  - restricted hardware is energy friendly
  - TOP ensure efficient construction and evolution



? ? ?

10^{-2} to 10^{-4} energy use

user defined mTask constructs

- the host language is your powerful macro language

```
:: When v a b = When infix 2 ((v a)-> MTask v b) ((v a) -> v Bool)

(>>?.) infixr 1 :: (MTask v a) (When v a b) -> MTask v b
                       | mtask v & type a & type b
(>>?.) t (f When c) = t >>*. [IfValue c f]

tempMTask3 :: (Main (MTask v ())) | mtask, lowerSds, dht v
tempMTask3 =
  dht dhtWemosSHT30Shield \sensor->
  lowerSds \rSDS = tSDS In
  fun \measure = (\old ->
    temperature sensor >>?.
         (\new -> setSds rSDS new >>|. measure new)
    When ((!=.) old)) In
  {main = getSds rSDS >>~. measure}
```