Could Tierless Languages Reduce IoT Development Grief?

MART LUBBERS, Radboud University, Netherlands
PIETER KOOPMAN, Radboud University, Netherlands
ADRIAN RAMSINGH, University of Glasgow, United Kingdom
JEREMY SINGER, University of Glasgow, United Kingdom
PHIL TRINDER, University of Glasgow, United Kingdom

Internet of Things (IoT) software is notoriously complex, conventionally comprising multiple tiers. Traditionally an IoT developer must use multiple programming languages and ensure that the components interoperate correctly. A novel alternative is to use a single *tierless* language with a compiler that generates the code for each component and ensures their correct interoperation.

We report a systematic comparative evaluation of two tierless language technologies for IoT stacks: one for resource-rich sensor nodes (Clean with iTask), and one for resource-constrained sensor nodes (Clean with iTask and mTask). The evaluation is based on four implementations of a typical smart campus application: two tierless and two Python-based tiered.

(1) We show that tierless languages have the potential to significantly reduce the development effort for IoT systems, requiring 70% less code than the tiered implementations. Careful analysis attributes this code reduction to reduced interoperation (e.g. two embedded domain-specific languages (DSLs) and one paradigm versus seven languages and two paradigms), automatically generated distributed communication, and powerful IoT programming abstractions. (2) We show that tierless languages have the potential to significantly improve the reliability of IoT systems, describing how Clean iTask/mTask maintains type safety, provides higher order failure management, and simplifies maintainability. (3) We report the first comparison of a tierless IoT codebase for resource-rich sensor nodes with one for resource-constrained sensor nodes. The comparison shows that they have similar code size (within 7%), and functional structure. (4) We present the first comparison of two tierless IoT languages, one for resource-rich sensor nodes, and the other for resource-constrained sensor nodes.

CCS Concepts: • **Computer systems organization** → *Sensor networks*; **Embedded software**; • **Software and its engineering** → *Domain specific languages.*

Additional Key Words and Phrases: Tierless languages, IoT stacks

1 INTRODUCTION

Conventional Internet of Things (IoT) software stacks are notoriously complex and pose very significant software development, reliability, and maintenance challenges. IoT software architectures typically comprise multiple components organised in four or more tiers or layers [5, 60, 67]. This is due to the highly distributed nature of typical IoT applications that must read sensor data from end points (the *perception* layer), aggregate and select

Authors' addresses: Mart Lubbers, Radboud University, P.O. Box 9010, Nijmegen, Netherlands, firstname@cs.ru.nl; Pieter Koopman, Radboud University, P.O. Box 9010, Nijmegen, Netherlands, firstname@cs.ru.nl; Adrian Ramsingh, University of Glasgow, 17 Lilybank Gardens, Glasgow, United Kingdom, firstname.lastname@glasgow.ac.uk; Jeremy Singer, University of Glasgow, 17 Lilybank Gardens, Glasgow, United Kingdom, firstname.lastname@glasgow.ac.uk; Phil Trinder, University of Glasgow, 17 Lilybank Gardens, Glasgow, United Kingdom, firstname.lastname@glasgow.ac.uk.

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
2577-6207/2022/8-ART111 $15.00
https://doi.org/10.1145/3572901

Just Accepted

the data and communicate over a network (the *network* layer), store the data in a database and analyse it (the *application* layer) and display views of the data, commonly on web pages (the *presentation* layer).

Conventional IoT software architectures require the development of separate programs in various programming languages for each of the components/tiers in the stack. This is modular, but a significant burden for developers, and some key challenges are as follows. (1) Interoperating components in multiple languages and paradigms increases the developer's cognitive load who must simultaneously think in multiple languages and paradigms, i.e. manage significant semantic friction. (2) The developer must correctly interoperate the components, e.g. adhere to the API or communication protocols between components. (3) To ensure correctness the developer must maintain type safety across a range of very different languages and diverse type systems. (4) The developer must deal with the potentially diverse failure modes of each component, and of component interoperations.

A radical alternative development paradigm uses a single *tierless* language that synthesizes all components/tiers in the software stack. There are established *tierless* languages for web stacks, e.g. Links [15] or Hop [66].

In a tierless language the developer writes the application as a single program. The code for different tiers is simultaneously checked by the compiler, and compiled to the required component languages. For example, Links compiles to HTML and JavaScript for the web client and to SQL on the server to interact with the database system. Tierless languages for IoT stacks are more recent and less common, examples include Potato [79] and Clean with iTask/mTask [43].

IoT sensor nodes may be microcontrollers with very limited compute resources, or supersensors: resource-rich single board computers like a Raspberry Pi. A tierless language may target either class of sensor node, and microcontrollers are the more demanding target due to the limited resources, e.g. small memory, executing on bare metal etc.

Potentially a tierless language both reduces the development effort and improves correctness as correct interoperation and communication is automatically generated by the compiler. A tierless language may, however, introduce other problems. How expressive is the language? That is, can it readily express the required functionality? How maintainable is the software? Is the generated code efficient in terms of time, space, and power?

This paper reports a systematic comparative evaluation of two tierless language technologies for IoT stacks: one targeting resource-constrained microcontrollers, and the other resource-rich supersensors. The basis of the comparison is four implementations of a typical smart campus IoT stack [29]. Two implementations are conventional tiered Python-based stacks: PRS (Python Raspberry Pi System) and PWS (Python Wemos System). The other two implementations are tierless: CRS (Clean Raspberry Pi System) and CWS (Clean Wemos System). Our work makes the following research contributions, and the key results are summarised, discussed, and quantified in Section 9.

C1 We show that *tierless languages have the potential to significantly reduce the development effort for IoT systems*. We systematically compare code size (source lines of code (SLOC)) of the four smart campus implementations as a measure of development effort and maintainability [4, 63]. The tierless implementations require 70% less code than the tiered implementations. We analyse the codebases to attribute the code reduction to three factors. (1) Tierless languages benefit from reduced interoperation, requiring far fewer languages, paradigms and source code files e.g. CWS uses two languages, one paradigm and three source code files where PWS uses seven languages, two paradigms and 35 source code files (Tables 2 and 3). (2) Tierless languages benefit from automatically synthesized, and hence correct, communication between components that may be distributed. (3) Tierless languages benefit from high-level programming abstractions like compositional and higher-order task combinators (Section 6).

C2 We show that *tierless languages have the potential to significantly improve the reliability of IoT systems*. We demonstrate how tierless languages preserve type safety, improve maintainability and provide high-level

Just Accepted

failure management. For example, we illustrate a loss of type safety in PRS. We also critique current tool and community support (Section 7).

C3 We report *the first comparison of a tierless IoT codebase for resource-rich sensor nodes with one for resource-constrained sensor nodes*. The tierless smart campus implementations have a very similar code size (within 7%), as do the tiered implementations. This suggests that the development and maintenance effort of simple tierless IoT systems for resource-constrained and for resource-rich sensor nodes is similar, as it is for tiered technologies. The percentages of code required to implement each IoT functionality in the tierless Clean implementations is very similar, as it is in the tiered Python implementations. This suggests that the code for resource-constrained and resource-rich sensor nodes is broadly similar in tierless technologies, as in tiered technologies (Section 6.2)

C4 *We present the first comparison of two tierless IoT languages, one designed for resource-constrained sensor nodes (Clean with iTask and mTask), and the other for resource-rich sensor nodes (Clean with iTask).* We show that the bare metal execution environment enforces some restrictions on mTask although they remain high level. Moreover, the environment conveys some advantages, e.g. better control over timing (Section 8).

The current work extends [46] as follows. Contributions C3 and C4 are entirely new, and C1 is enhanced by being based on the analysis of four rather than two languages and implementations.

2 BACKGROUND AND RELATED WORK

2.1 University of Glasgow smart campus

The University of Glasgow (UoG) is partway through a ten-year campus upgrade programme, and a key goal is to embed pervasive sensing infrastructure into the new physical fabric to form a *smart campus* environment. As a prototyping exercise, we use modest commodity sensor nodes (i.e. Raspberry Pis) and low-cost, low-precision sensors for indoor environmental monitoring.

We have deployed sensor nodes into 12 rooms in two buildings. The IoT system has an online data store, providing live access to sensor data through a RESTful API. This allows campus stakeholders to add functionality at a business layer above the layers that we consider here. To date, simple apps have been developed including room temperature monitors and campus utilization maps [29]. A longitudinal study of sensor accuracy has also been conducted [27].

2.2 IoT applications

Web applications are necessarily complex distributed systems, with client browsers interacting with a remote webserver and data store. Typical IoT applications

are even more complex as they combine a web application with a second distributed system of sensor and actuator nodes that collect and aggregate data, operate on it, and communicate with the server.

Both web and IoT applications are commonly structured into tiers, e.g. the classical four-tier Linux, Apache, MySQL and PHP (LAMP) stack.

IoT stacks typically have more tiers than webapps, with the number depending on the complexity of the application [67]. While other tiers, like the business layer [52] may be added above them, the focus of our study is on programming the lower four tiers of the PRS, CRS, PWS and CWS stacks, as illustrated in Figure 1.

1. Perception Layer – collects the data, interacts with the environment, and consists of devices using light, sound, motion, air quality and temperature sensors.
2. Network Layer – replays the communication messages between the sensor nodes and the server through protocols such as MQTT.
3. Application Layer – acts as the interface between the presentation layer and the perception layer, storing and processing the data.

Just Accepted

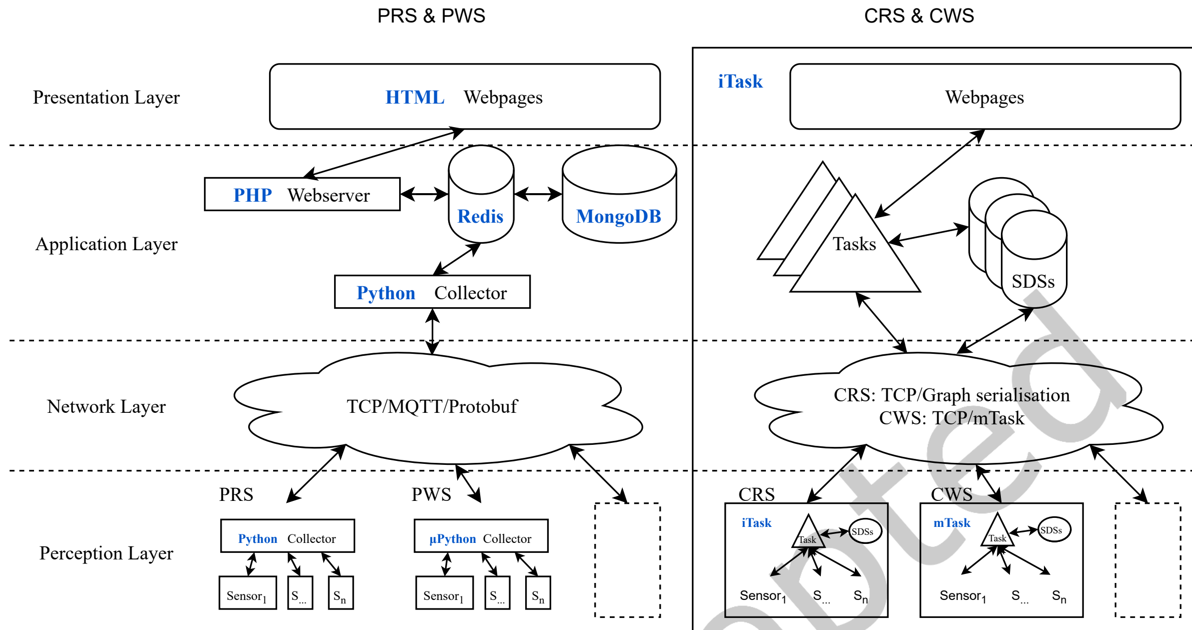

Fig. 1. PRS and PWS (left) together with CRS and PRS (right) mapped to the four-tier IoT architecture. Every box is the diagram denotes a source file or base. Bold blue text describes the language or technology used in that source. The network and perception layer are unique to the specific implementation, where the application and presentation layers are shared between implementations.

(4) Presentation Layer – utilises web components as the interface between the human and devices where application services are provided.

2.3 The benefits and challenges of developing tiered IoT stacks

Using multiple tiers to structure complex software is a common software engineering practice that provides significant architectural benefits for IoT and other software. The tiered Python PRS and PWS stacks exhibit these benefits.

(1) Modularity – tiers allow a system to be structured as a set of components with clearly defined functionality. They can be implemented independently, and may be interchanged with other components that have similar functionality [47]. In PRS and PWS, for example, a different NoSQL DBMS could relatively easily be substituted for MongoDB.
(2) Abstraction – the hierarchical composition of components in the stack abstracts the view of the system as a whole. Enough detail is provided to understand the roles of each layer and how the components relate to one another [10]. Figure 1 illustrates the abstraction of PRS and PWS into four tiers.
(3) Cohesion – well-defined boundaries ensure each tier contains functionality directly related to the task of the component [40]. The tiers in PRS and PWS contain all the functionality associated with perception, networking, application and presentation respectively.

However, a tiered architecture poses significant challenges for developers of IoT and other software. The tiered Python PRS and PWS stacks exhibit these challenges, and we analyse these in detail later in the paper.

Just Accepted

(1) Polyglot Development – the developer must be fluent in all the languages and components in the stack, known as being a full stack developer for webapps [50]. That is, the developer must correctly use multiple languages that have different paradigms, i.e. manage significant *semantic friction* [34]. For example the PWS developer must integrate components written in seven languages with two paradigms (Section 6.3).
(2) Correct Interoperation – the developer must adhere to the API or communication protocols between components. Sections 6.1 and 6.2 show that communication requires some 17% of PRS and PWS code, so around 100 lines of code. Section 6.4 discusses the complexity of writing this distributed communication code.
(3) Maintaining Type Safety – is a key element of the semantic friction encountered in multi-language stacks, and crucial for correctness. The developer must maintain type safety across a range of very different languages and diverse type systems, with minimal tool support. We show an example where PRS loses type safety over the network layer (Section 7.1).
(4) Managing multiple failure modes – different components may have different failure modes, and these must be coordinated. Section 7.2 outlines how PRS and PWS use heartbeats to manage failures.

Beyond PRS and PWS the challenges of tiered polyglot software development are evidenced in real world studies. As recent examples, a study of GitHub open source projects found an average of five different languages in each project, with many using tiered architectures [49]. An earlier empirical study of GitHub shows that using more languages to implement a project has a significant effect on project quality, since it increases defects [36]. A study of IoT stack developers found that interoperation poses a real challenge, that microservices blur the abstraction between tiers, and that both testing and scaling IoT applications to more devices are hard [51].

One way of minimising the challenges of developing tiered polyglot IoT software is to standardise and reuse components. This approach has been hugely successful for web stacks, e.g. browser standards. The W3C Web of Things aims to facilitate re-use by providing standardised metadata and other re-usable technological IoT building blocks [24]. However, the Web of Things has yet to gain widespread adoption. Moreover, as it is based on web technology, it requires the *thing* to run a web server, significantly increasing the hardware requirements.

Just Accepted

3 TIERLESS LANGUAGES

A radical approach to overcoming the challenges raised by tiered distributed software is to use a tierless programming language that eliminates the semantic friction between tiers by generating code for all tiers, and all communication between tiers, from a single program.

Typically a tierless program uses a single language, paradigm and type system, and the entire distributed system is simultaneously checked by the compiler.

There is intense interest in developing tierless, or multitier, language technologies with a number of research languages developed over the last fifteen years, e.g. [15, 19, 66, 79]. These languages demonstrate the advantages of the paradigm, including less development effort, better maintainability, and sound semantics of distributed execution. At the same time a number of industrial technologies incorporate tierless concepts, e.g. [8, 11, 73]. These languages demonstrate the benefits of the paradigm in practice. Some tierless languages use (embedded) domain-specific languages (DSLs) to specify parts of the multi-tier software.

Tierless languages have been developed for a range of distributed paradigms, including web applications, client-server applications, mobile applications, and generic distributed systems. A recent and substantial survey of these tierless technologies is available [82]. Here we provide a brief introduction to tierless languages with a focus on IoT software.

3.1 Tierless Web Languages

There are established tierless languages for web development, both standalone languages and DSLs embedded in a host language. Example standalone tierless web languages are Links [15] and Hop [66]. From a single declarative program the client, server and database code is simultaneously checked by the compiler, and compiled to the required component languages. For example, Links compiles to HTML and JavaScript for the client side and to SQL on the server-side to interact with the database system.

An example tierless web framework that uses a DSL is Haste [19], that embeds the DSL in Haskell. Haste programs are compiled multiple times: the server code is generated by the standard GHC Haskell compiler [26]; Javascript for the client is generated by a custom GHC compiler backend. The design leverages Haskell's high-level programming abstractions and strong typing, and benefits from GHC: a mature and sophisticated compiler.

3.2 Tierless IoT Languages

The use of tierless languages in IoT applications is both more recent and less common than for web applications. Tierless IoT programming may extend tierless web programming by adding network and perception layers. The presentation layer of a tierless IoT language, like tierless web languages, benefits from almost invariably executing in a standard browser. The perception layer faces greater challenges, often executing on one of a set of slow and resource-constrained microcontrollers. Hence, tierless IoT languages typically compile the perception layer to either C/C++ (the lingua franca of microcontrollers), or to some intermediate representation to be interpreted.

3.2.1 DSLs for microcontrollers. Many DSLs provide high-level programming for microcontrollers, for example providing strong typing and memory safety.

For example Copilot [31]

and Ivory [20] are imperative DSLs embedded in a functional language that compile to C/C++. In contrast to Clean iTask/mTask such DSLs are not tierless IoT languages as they have no automatic integration with the server, i.e. with the application and presentation layers.

3.2.2 Functional Reactive Programming. Functional reactive programming (FRP) is a declarative paradigm often used for implementing the perception layer of an IoT stack. Examples include mfrp [65],

Hailstorm [64], and Haski [80]. None of these languages are tierless IoT languages as they have no automatic integration with the server.

Potato goes beyond other FRP languages to provide a tierless FRP IoT language for resource rich sensor nodes [79]. It does so using the Erlang programming language and sophisticated virtual machine.

TOP allows for more complex collaboration patterns than FRP [74], and in consequence is unable to provide the strong guarantees on memory usage available in a restricted variant of FRP such as arrowized FRP [53].

3.2.3 Erlang/Elixir IoT systems. A number of production IoT systems are engineered in Erlang or Elixir, and many are mostly tierless. That is the perception, network and application layers are sets of distributed Erlang processes, although the presentation layer typically uses some conventional web technology. A resource-rich sensor node may support many Erlang processes on an Erlang VM, or low level code (typically C/C++) on a resource-constrained microcontroller can emulate an Erlang process. Only a small fraction of these systems are described in the academic literature, example exceptions are [69, 70], with many described only in grey literature or not at all.

3.3 Characteristics of Tierless IoT Languages

This study compares a pair of tierless IoT languages with conventional tiered Python IoT software. Clean with iTask and Clean iTask/mTask represent a specific set of tierless language design decisions, however many alternative designs are available. Crucially the limitations of the tierless Clean languages, e.g. that they currently

Just Accepted

provide limited security, should not be seen as limitations of tierless technologies in general. This section briefly outlines key design decisions for tierless IoT languages, discusses alternative designs, and describes the Clean designs. The Clean designs are illustrated in the examples in the following section.

3.3.1 Tier Splitting and Placement. A key challenge for a tierless language is to determine which parts of the program correspond to a particular tier and hence should be executed by a specific component on a specific host. For example a tierless web language must identify client code to ship to browsers, database code to execute in the DBMS, and application code to run on the server. Tierless web languages can make this determination statically, so-called *tier splitting* using types or syntactic markers like `server` or `client` pragmas [15, 19]. It is even possible to infer the splitting, relieving the developers from the need to specify it, as illustrated for Javascript as a tierless web language [56].

In Clean iTask/mTask and Clean with iTask tier splitting is specified by functions, e.g. the Clean with iTask `asyncTask` function identifies a task for execution on a remote device and `liftmTask` executes the given task on an IoT device. The tier splitting functions are illustrated in examples in the next section, e.g. on line 17 in Listing 3 and line 30 in Listing 4. Specifying splitting as functions means that new splitting functions can be composed, and that splitting is under program control, e.g. during execution a program can decide to run a task locally or remotely.

As IoT stacks are more complex than web stacks, the *placement* of data and computations onto the devices/hosts in the system is more challenging. In many IoT systems placement is manual: the sensor nodes are microcontrollers that are programmed by writing the program to flash memory. So physical access to the microcontroller is normally required to change the program, making updates challenging.

Techniques like over-the-air programming and interpreters allow microcontrollers to be dynamically provisioned, increasing their maintainability and resilience. For example Baccelli et al. provide a single language IoT system based on the RIOT operating system (OS) that allows runtime deployment of code snippets called containers [7]. Both client and server are written in JavaScript. However, there is no integration between the client and the server other than that they are programmed from a single source. Matè is an example of an early tierless sensor network framework where devices are provided with a virtual machine using TinyOS for dynamic provisioning [41].

In general different tierless languages specify placement in different ways, e.g. code annotations or configuration files, and at different granularities, e.g. per function or per class [82].

Clean iTask/mTask and Clean with iTask both use dynamic task placement. In Clean iTask/mTask sensor nodes are programmed once with the mTask RTS, and possibly some precompiled tasks. Thereafter a sensor node can dynamically receive mTask programs, compiled at runtime by the server. In Clean with iTask the sensor node runs an iTask server that recieves and executes code from the (IoT) server [54].

Placement happens automatically as part of the first-class splitting constructs, so line 30 in Listing 4 places `devTask` onto the `dev` sensor node.

3.3.2 Communication. Tierless languages may adopt a range of communication paradigms for communicating between components. Different tierless languages specify communication in different ways [82]. Remote procedures are the most common communication mechanism: a procedure/function executing on a remote host/machine is called as if it was local. The communication of the arguments to, and the results from, the remote procedure is automatically provided by the language implementation. Other mechanisms include explicit message passing between components; publish/subscribe where components subscribe to topics of interest from other components; reactive programming defines event streams between remote components; finally shared state makes changes in a shared and potentially remote data structure visible to components.

Clean iTask/mTask and Clean with iTask communicate using a combination of remote task invocation, similar to remote procedures, and shared state through Shared Data Stores (SDSs). Listing 3 illustrates: line 17 shows a

Just Accepted

server task launching a remote task, `devTask`, on to a sensor node; and line 19 shows the sharing of the remote `latestTemp` SDS.

3.3.3 Security. Security is a major issue and a considerable challenge for many IoT systems [2]. There are potentially security issues at each layer in an IoT application (Figure 1). The security issues and defence mechanisms at the application and presentation layers are relatively standard, e.g. defending against SQL injection attacks. The security issues at the network and perception layers are more challenging. Resource-rich sensor nodes can adopt some standard security measures like encrypting messages, and regularly applying software patches to the operating system. However microcontrollers often lack the computational resources for encryption, and it is hard to patch their system software because the program is often stored in flash memory. In consequence there are infamous examples of IoT systems being hijacked to create botnets [6, 30].

Securing the entire stack in a conventional tiered IoT application is particularly challenging as the stack is implemented in a collection of programming languages with low level programming and communication abstractions. In such polyglot distributed systems it is hard to determine, and hence secure, the flow of data between components. In consequence a small mistake may have severe security implications.

A number of characteristics of tierless languages help to improve security. Communication and placement vulnerabilities are minimised as communication and placement are automatically generated and checked by the compiler. So injection attacks and the exploitation of communication/placement protocol bugs are less likely. Vulnerabilities introduced by mismatched types are avoided as the entire system is type checked. Moreover, tierless languages can exploit language level security techniques. For example languages like Jif/split [84] and Swift [14] place components to protect the security of data. Another example are programming language technologies for controlling information flow, and these can be used to improve security. For example Haski uses them to improve the security of IoT systems [80].

However many tierless languages have yet to provide a comprehensive set of security technologies, despite its importance in domains like web and IoT applications. For example Erlang and many Erlang-based systems [69, 70], lack important security measures. Indeed security is not covered in a recent, otherwise comprehensive, survey of tierless technologies [82].

Clean with iTask and Clean iTask/mTask are typical in this respect: little effort has yet been expended on improving their security. Of course as tierless languages they benefit from static type safety and automatically generated communication and placement. Some preliminary work shows that, as the communication between layers is protocol agnostic, more secure alternatives can be used. One example is to run the iTask server behind a reverse proxy implementing TLS/SSL encryption [83]. A second is to add integrity checks or even encryption to the communication protocol for resource-rich sensor nodes [12].

4 TASK-ORIENTED AND IOT PROGRAMMING IN CLEAN

To make this paper self-contained we provide a concise overview of Clean, task-oriented programming (TOP), and IoT programming in iTask and mTask. The minor innovations reported here are the interface to the IoT sensors, and the Clean port for the Raspberry Pi.

Clean is a statically typed functional programming language similar to Haskell: both languages are pure and non-strict [1].

A key difference is how state is handled: Haskell typically embeds stateful actions in the IO Monad [28, 55]. In contrast, Clean has a uniqueness type system to ensure the single-threaded use of stateful objects like files and windows [9, 61]. Both Clean and Haskell support fairly similar models of generic programming [62], enabling functions to work on many types. As we shall see generic programming is heavily used in task-oriented programming [3, 32], for example to construct web editors and communication protocols that work for any user-defined datatype.

Just Accepted

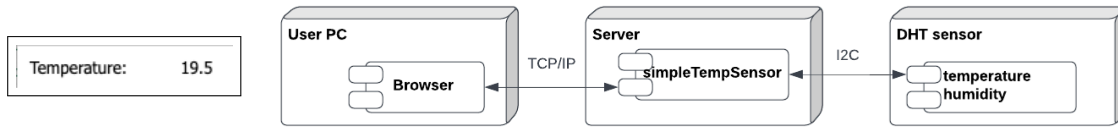

Fig. 2. iTask SimpleTempSensor: web page (left) and deployment diagram (right).

4.1 Task-Oriented Programming

TOP is a declarative programming paradigm for constructing interactive distributed systems [59]. Tasks are the basic blocks of TOP and represent work that needs to be done in the broadest sense. Examples of typical tasks range from allowing a user to complete a form, controlling peripherals, moderating other tasks, or monitoring a database. From a single declarative description of tasks all of the required software components are generated. This may include web servers, client code for browsers or IoT devices, and for their interoperation.

That is, from a single TOP program the language implementation automatically generates an *integrated distributed system*. Application areas range from simple web forms or blinking LEDs to multi-user distributed collaboration between people and machines [54].

TOP adds three concepts: tasks, task combinators, and Shared Data Stores (SDSs). Example basic tasks are web editors for user-defined datatypes, reading some IoT sensor, or controlling peripherals like a servo motor. Task combinators compose tasks into more advanced tasks, either in parallel or sequential and allow task values to be observed by other tasks. As tasks can be returned as the result of a function, recursion can be freely used, e.g. to express the repetition of tasks. There are also standard combinators for common patterns. Tasks can exchange information via SDSs [17]. All tasks involved can atomically observe and change the value of a typed SDS, allowing more flexible communication than with task combinators. SDSs offer a general abstraction of data shared by different tasks, analogous to variables, persistent values, files, databases and peripherals like sensors. Combinators compose SDSs into a larger SDS, and parametric lenses define a specific view on an SDS.

Just Accepted

4.2 The iTask embedded DSL

The iTask embedded DSL is designed for constructing multi-user distributed applications, including web [58] or IoT applications. Here we present iTask by example, and the first is a complete program to repeatedly read the room temperature from a digital humidity and temperature (DHT) sensor attached to the machine and display it on a web page (Listing 1). The first line is the module name, the third imports the `iTask` module, and the main function (lines 5 and 6) launches `readTempTask` and the iTask system to generate the web interface in Figure 2.

Interaction with a device like the DHT sensor using a protocol like 1-Wire or I^2C is abstracted into a library. So the `readTempTask` task starts by creating a `dht` sensor object (line 10) thereafter `repeatEvery` executes a task at the specified `interval`. This task reads the temperature from the `dht` sensor, and with a sequential composition combinator `>>~` passes the `temp` value to `viewInformation` that displays it on the web page (line 13). The tuning combinator `<<@` adds a label to the web editor displaying the temperature. Crucially, the iTask implementation transparently ships parts of the code for the web-interface to be executed in the browser, and Figure 2 shows the UML deployment diagram.

SimpleTempSensor only reports instantaneous temperature measurements. Extending it to store and manipulate timed temperature records produces a tiny tierless web application: TempHistory in Listing 2. A tierless IoT system can be controlled from a web interface in exactly the same way, e.g. to view and set the measurement frequencies of each of the room sensors. Line 5 defines a record to store `time` and `temp` measurements. Task

```
1  module simpleTempSensor
2
3  import iTasks
4
5  Start :: *World → *World
6  Start world = doTasks readTempTask world
7
8  readTempTask :: Task Real
9  readTempTask =
10     withDHT IIO_TempID λdht →
11     repeatEvery interval (
12         temperature dht >>~ λtemp →
13         viewInformation [] temp <<@
14         Label "Temperature"
15     )
```

Listing 1. SimpleTempSensor: a Clean with iTask program to read a local room temperature sensor and display it on a web page

manipulations are derived for `Measurement` (line 6) and these include displaying measurements in a web-editor and storing them in a file. Line 8 defines a persistent SDS to store a list of measurements, and for communication between tasks. The SDS is analogous to the SQL DBMS in many tiered web applications.

TempHistory defines two tasks that interact with the `measurementsSDS`: `measureTask` adds measurements at each detected change in the temperature. It starts by defining a `dht` object as before, and then defines a recursive `task` function parameterized by the `old` temperature. This function reads the temperature from the DHT sensor and uses the step combinator, ≫*, to compose it with a list of actions. The first of those actions that is applicable determines the continuation of this task. If no action is applicable, the task on the left-hand side is evaluated again. The first action checks whether the new temperature is different from the `old` temperature (line 16). If so, it records the current time and adds the new measurements to the `measurementsSDS`. The next action in line 19 is always applicable and waits (sleeps) for an interval before returning the old temperature.

On line 21 `task` is launched with an initial temperature.

The `controlSDS` task illustrates communication from the web page user and persistent data manipulation. That is, it generates a web page that allows users to control their view of the temperature measurements, as illustrated in Figure 3. The page contains (1) a web editor to enter the number `n` of elements to display, generated on line 25. (2) A display of the `n` most recent temperature and time measurements, lines 26 to 28. (3) Three buttons that are again combined with the step combinator ≫*, lines 28 to 33. The `Clear` button is `always` enabled and sets the SDS to an empty list before calling `controlSDS` recursively. The `Take` button is only enabled when the web editor produces a positive `n` and updates the `measurementsSDS` with the `n` most recent measurements before calling `controlSDS` recursively. The final action is `always` enabled and calls `controlSDS` recursively with the negation of the `byTemp` argument to change the sorting criteria.

Figure 3 shows two screenshots of web pages generated by the `TempHistory` program. Figure 4 is the deployment diagram showing the addition of the persistent `measurementsSDS` that stores the history of temperature measurements.

4.3 Engineering Tierless IoT Systems with iTask

A typical IoT system goes beyond a web application by incorporating a distributed set of sensor nodes each with a collection of sensors or actuators. That is, they add the perception and network layers in Figure 1. If the sensor

Just Accepted

```
1  module TempHistory
2
3  import iTasks, iTasks.Extensions.DateTime
4
5  :: Measurement = {time :: Time, temp :: Real}
6  derive class iTask Measurement
7
8  measurementsSDS :: SimpleSDSLens [Measurement]
9  measurementsSDS = sharedStore "measurements" []
10
11 measureTask :: Task ()
12 measureTask =
13     withDHT IIO_TempID λdht →
14     let task old =
15             temperature dht >>*
16             [ OnValue (ifValue ((≠) old) λtemp →
17                 get currentTime >>~ λtime →
18                 upd (λlist → [{time = time, temp = temp}: list]) measurementsSDS @! temp)
19             , OnValue (always (waitForTimer False interval @! old))
20             ] >>~ task
21     in task initialTemp
22
23 controlSDS :: Bool → Task [Measurement]
24 controlSDS byTemp =
25     ((Label "# to take"    @>> enterInformation []) -||
26      (Label "Measurements" @>>
27       viewSharedInformation [ViewAs (if byTemp (sortBy (λx y → x.temp < y.temp)) id)]
28                             measurementsSDS)) >>*
29         [OnAction (Action "Clear") (always (set [] measurementsSDS >-|  controlSDS byTemp))
30         ,OnAction (Action "Take")  (ifValue ((<) 0) (λn → upd (take n) measurementsSDS >-|
31                                                                        controlSDS byTemp))
32         ,OnAction (Action (if byTemp "Sort time" "Sort temp")) (always (controlSDS (not byTemp)))
33         ]
34
35 mainTask :: Task [Measurement]
36 mainTask = controlSDS False -|| measureTask
```

Listing 2. TempHistory: a tierless Clean with iTask webapplication that records and manipulates timed temperatures

Just Accepted

nodes have the computational resources to support an iTask server, as a Raspberry Pi does, then iTask can also be used to implement these layers, and integrate them with the application and presentation layers tierlessly.

As an example of tierless IoT programming in Clean with iTask Listing 3 shows a complete temperature sensing system with a server and a single sensor node (CRTS (Clean Raspberry Pi Temperature Sensor)), omitting only the module name and imports. It is similar to the SimpleTempSensor and TempHistory programs above, for example `devTask` repeatedly sleeps and records temperatures and times, and `mainTask` displays the temperatures on the web page in Figure 5. There are some important differences, however. The `devTask` (lines 8–13) executes on the sensor node and records the temperatures in a standard timestamped (lens on) an SDS: `dateTimeStampedShare latestTemp`. The `mainTask` (line 16) executes on the server: it starts `devTask` as an asynchronous task on the specified sensor node (line 17) and then generates a web page to display the latest temperature and time (lines 18 to 20).

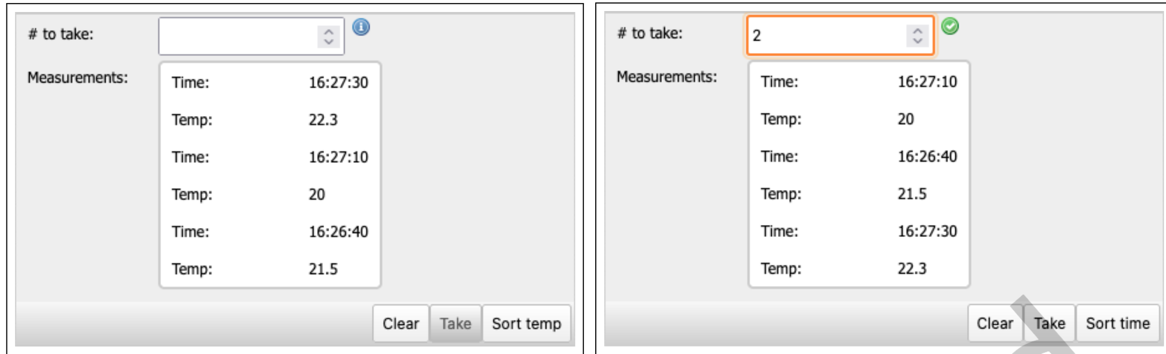

Fig. 3. Web pages generated by the `TempHistory` Clean with iTask tierless web application: on the left sorted by time; on the right sorted by temperature. The `Take` button is only enabled when the topmost editor contains a positive number.

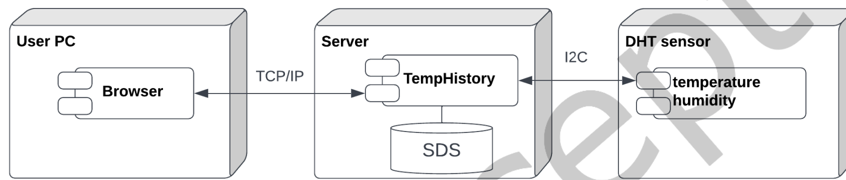

Fig. 4. Deployment diagram of the iTask TempHistory tierless web application from Listing 2.

The `tempSDS` is very similar to the `measurementsSDS` from the previous listings. The only difference is that we store measurements as tuples instead of tailor-made records. The `latestTemp` is a *lens* on the `tempSDS`. A lens is a new SDS that is automatically mapped to another SDS. Updating one of the SDSs that are coupled in this way automatically updates the other. The function `mapReadWrite` is parameterized by the read and write functions, the option to handle asynchronous update conflicts (here `?None`) and the SDS to be transformed (here `tempSDS`). The result of reading is the head of the list, if it exists. The type for writing `latestTemp` is a tuple with a new `DateTime` and temperature as `Real`.

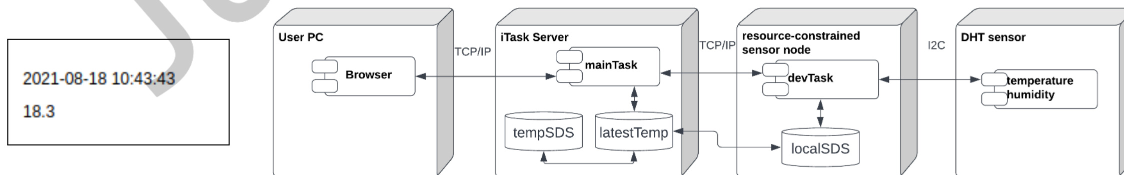

Fig. 5. Tierless iTask CRTS temperature sensing IoT system: web page (left) and deployment diagram (right).

Just Accepted

```
1  tempSDS :: SimpleSDSLens [(DateTime, Real)]
2  tempSDS = sharedStore "temperatures" []
3
4  latestTemp :: SDSLens () (? (DateTime, Real)) (DateTime, Real)
5  latestTemp = mapReadWrite (listToMaybe, λx xs → ?Just [x:xs]) ?None tempSDS
6
7  devTask :: Task DateTime
8  devTask =
9      withDHT IIO_TempID λdht →
10     forever (
11         temperature dht >>~ λtemp →
12         set temp (dateTimeStampedShare latestTemp) >-|
13         waitForTimer False interval)
14
15 mainTask :: Task ()
16 mainTask
17     =   asyncTask deviceInfo.domain deviceInfo.port devTask
18     -||  viewSharedInformation []
19         (remoteShare latestTemp deviceInfo)
20         <<@ Title "Latest temperature"
```

Listing 3. CRTS: a tierless temperature sensing IoT system. Written in Clean with iTask, it targets a resource-rich sensor node.

Just Accepted

4.4 The mTask embedded DSL

In many IoT systems the sensor nodes are resource constrained, e.g. inexpensive microcontrollers. These are far cheaper, and consume far less power, than a single-board computer like a Raspberry Pi. Microcontrollers also allow the programmer to easily control peripherals like sensors and actuators via the IO pins of the processor.

Microcontrollers have limited memory capacity, compute power and communication bandwidth, and hence typically no OS. These limitations make it impossible to run an iTask server: there is no OS to start the remote task, the code of the task is too big to fit in the available memory and the microcontroller processor is too slow to run it. The mTask embedded DSL is designed to bridge this gap: mTasks can be communicated from the server to the sensor node, to execute within the limitations of a typical microcontroller, while providing programming abstractions that are consistent with iTask.

Like iTask, mTask is task oriented, e.g. there are primitive tasks that produce intermediate values, a restricted set of task combinators to compose the tasks, and (recursive) functions to construct tasks. mTasks communicate using task values or SDSs that may be local or remote, and may be shared by some iTask tasks.

Apart from the embedded DSL, the mTask system contains a featherlight domain-specific *operating system* running on the microcontroller. This OS task scheduler receives the bytecode generated from one or more mTask programs and interleaves the execution of those tasks. The OS also manages SDS updates and the passing of task results. The mTask OS is stored in flash memory while the tasks are stored in RAM to minimise wear on the flash memory. While sending bytecode to a sensor node at runtime greatly increases the amount of communication, this can be mitigated as any tasks known at compile time can be preloaded on the microcontroller. In contrast, compiled programs, like C/C++, are stored in flash memory and there can only ever be a few thousand programs uploaded during the lifetime of the microcontroller before exhausting the flash memory.

4.5 Engineering Tierless IoT Systems with mTask

A tierless Clean IoT system with microcontroller sensor nodes integrates a set of iTask tasks that specify the application and presentation layers with a set of mTasks that specify the perception and network layers. We illustrate with CWTS (Clean Wemos Temperature Sensor): a simple room temperature sensor with a web display. CWTS is equivalent to the iTask CRTS (Listing 3), except that the sensor node is a Wemos microcontroller.

Listing 4 shows the complete program, and the key function is `devTask` with a top-level `Main` type (line 19). In mTask functions, shares, and devices can only be defined at this top level. The program uses the same shares `tempSDS` and `latestTemp` as CRTS, and for completeness we repeat those definitions.

The body of `devTask` is the mTask slice of the program (lines 21–26). With `DHT` we again create a temperature sensor object `dht`. The iTask SDS `latestTemp` is first transformed to a SDS that accepts only temperature values, the `dateTimeStampedShare` adds the data via a lens. The `mapRead` adjusts the read type. This new SDS of type `Real` is lifted to the mTask program with `liftsds`.

The `mainTask` is a simple iTask task that starts the `devTask` mTask task on the device identified by `deviceInfo` (line 30). At runtime the mTask slice is compiled to bytecode, shipped to the indicated device, and launched. Thereafter, `mainTask` reads temperature values from the `latestTemp` SDS that is shared with the mTask device, and displays them on a web page (Figure 5). The SDS—shared with the device using `liftsds`—automatically communicates new temperature values from the microcontroller to the server.

While this simple application makes limited use of the mTask embedded DSL, it illustrates some powerful mTask program abstractions like basic tasks, task combinators, named recursive and parameterized tasks, and SDSs. Function composition (line 23) and currying (line 26) are inherited from the Clean host language. As mTask tasks are dynamically compiled, it is also possible to select and customise tasks as required at runtime. For example, the interval used in the `rpeatevery` task (line 24) could be a parameter to the `devTask` function.

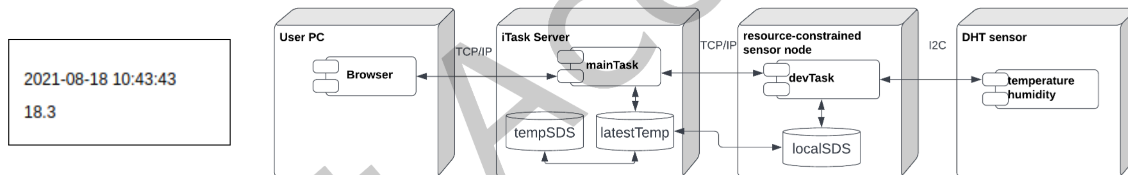

Fig. 6. Tierless iTask/mTask CWTS temperature sensing IoT system: web page (left) and deployment diagram (right).

5 UOG SMART CAMPUS CASE STUDY

The basis for our comparison between tiered and tierless technologies are four IoT systems that all conform to the UoG Smart Campus specifications (Section 5.3). There is a small (12 room) deployment of the conventional Python-based PRSS stack that uses Raspberry Pi supersensors, and its direct comparator is the tierless CRSS implementation: also deployed on Raspberry Pis.

To represent the more common microcontroller sensor nodes we select ESP8266X powered Wemos D1 Mini microcontrollers. To evaluate tierless technologies on microcontrollers we compare the conventional Python/MicroPython PWS stack with the tierless CWS implementation.

A similar range of commodity sensors is connected to both the Raspberry Pi and Wemos sensor nodes using various low-level communication protocols such as general purpose input/output pins (GPIO), I^2C, SPI and one-wire. The sensors are as follows: Temperature & Humidity: LOLIN SHT30; Light: LOLIN BH1750; Motion: LOLIN PIR; Sound: SparkFun SEN-12642; CO2: SparkFun CCS811.

Just Accepted

```
1  module cwts
2
3  import mTask.Language, mTask.Interpret, mTask.Interpret.Device.TCP
4
5  import iTasks, iTasks.Extensions.DateTime
6
7  deviceInfo = {TCPSettings | host = "...", port = 8123, pingTimeout = ?None}        // CO
8  interval = lit 10                                                                   // SN
9  DHT_pin = DigitalPin D4                                                             // SI
10
11 Start world = doTasks mainTask world                                                // WI
12
13 tempSDS :: SimpleSDSLens [(DateTime, Real)]
14 tempSDS = sharedStore "temperatures" []                                             // DI
15
16 latestTemp :: SDSLens () (? (DateTime, Real)) (DateTime, Real)
17 latestTemp = mapReadWrite (listToMaybe, λx xs → ?Just [x:xs]) ?None tempSDS          // DI
18
19 devTask :: Main (MTask v Real) | mtask, dht, liftsds v
20 devTask =
21   DHT (DHT_DHT DHT_pin DHT11) λdht →                                                 // SI
22   liftsds λlocalSds =                                                               // CO
23     mapRead (snd o fromJust) (dateTimeStampedShare latestTemp)                      // SN
24   In {main = rpeatEvery (ExactSec interval)                                         // SN
25     (temperature dht >>~.                                                           // SI
26      setSds localSds)}                                                              // SN
27
28 mainTask :: Task Real
29 mainTask
30   =   withDevice deviceInfo λdev → liftmTask devTask dev                            // CO
31    -||  viewSharedInformation [] latestTemp                                          // WI
32   <<@ Title "Latest temperature"                                                    // WI
```

Listing 4. CWTS: a tierless temperature sensing IoT system. Written in Clean iTask/mTask, it targets a resource-constrained sensor node. Each line is annotated with the functionality as analysed in Section 6.1.

Just Accepted

Figure 7 shows both a prototype Wemos-based sensor node and sensors and a Raspberry Pi supersensor. Three different development teams developed the four implementations: CWS and CRS were engineered by a single developer.

5.1 Tiered Implementations

The tiered PRS and PWS share the same server code executing on a commodity PC (Figure 1). The Python server stores incoming sensor data in two database systems, i.e. Redis (in-memory data) and MongoDB (persistent data). The real-time sensor data is made available via a streaming websockets server, which connects with Redis. There is also an HTTP REST API for polling current and historical sensor data, which hooks into MongoDB. Communication between a sensor node and the server is always initiated by the node.

PRS's sensor nodes are relatively powerful Raspberry Pi 3 Model Bs. There is a simple object-oriented Python collector for configuring the sensors and reading their values. The collector daemon service marshals the sensor

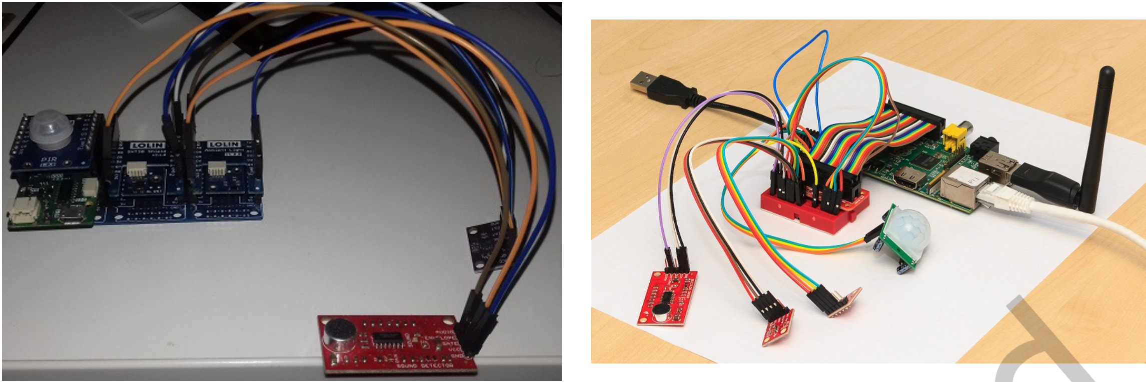

Fig. 7. Exposed views of sensor nodes: the Wemos on the left is used in PWS and CWS; the Raspberry Pi on the right is used in PRS and CRS.

Just Accepted

data and transmits using MQTT to the central monitoring server at a preset frequency. The collector caches sensor data locally when the server is unreachable.

In contrast to PRS, PWS's sensor nodes are microcontrollers running MicroPython, a dialect of Python specifically designed to run on small, low powered embedded devices [37]. To enable a fair comparison between the software stacks we are careful to use the same object-oriented software architecture, e.g. using the same classes in PWS and PRS.

Python and MicroPython are appropriate tiered comparison languages. Tiered IoT systems are implemented in a whole range of programming languages, with Python, MicroPython, C and C++ being popular for some tiers in many implementations. C/C++ implementations would probably result in more verbose programs and even less type safety. The other reasons for selecting Python and MicroPython are pragmatic. PRS had been constructed in Python, deployed, and was being used as an IoT experimental platform. Selecting MicroPython for the resource-constrained PWS sensor nodes facilitates comparison by minimising changes to the resource-rich and resource-constrained codebases.

We anticipate that the codebase for a tiered smart campus implementation in another imperative/object-oriented language, like C++, would be broadly similar to the PRS and PWS codebases.

5.2 Tierless Implementations

The tierless CRS and CWS servers share the same iTask server code (Figure 1), and can be compiled for many standard platforms. They use

SQLite as a database backend. Communication between a sensor node and the server is initiated by the server.

CRS's sensor nodes are Raspberry Pi 4s, and execute Clean iTask programs.

Communication from the sensor node to the server is implicit and happens via SDSs over TCP using platform independent execution graph serialisation [54].

CWS's sensor nodes are Wemos microcontrollers running mTasks. Communication and serialisation is, by design, very similar to iTask, i.e. via SDSs over either a serial port connection, raw TCP, or MQTT over TCP.

Room m.01.07: 2020-04-30 09:15:03
Temperature: 20.4
Humidity: 75
Noise: 15
Light: 70
Motion:
Co2: 460

Temperature: 24.69
Humidity:76.42
Sound Level: 51
Light: 223.9
CO2: 319
Motion: Active
Timestamp: 25 Aug 2021 06:53:27

Fig. 8. Web interfaces for CWS and CRS (left); PWS and PRS (right).

5.3 Operational Equivalence

To ensure that the comparison reported in the following sections is based on IoT stacks with equivalent functionality, we demonstrate that PWS, CWS and CRS, like PRS, meet the functional requirements for the UoG smart campus sensor system. We also compare the sensor node power consumption and memory footprint.

5.3.1 Functional Requirements. The main goal of the University of Glasgow (UoG) smart campus project is to provide a testbed for sensor nodes and potentially other devices to act as a data collection and computation platform for the UoG smart campus. The high-level functional requirements, as specified by the UoG smart campus project board, are as follows. The system should:

(1) be able to measure temperature and humidity as well as light intensity,
(2) scale to no more than 10 sensors per sensor node and investigate further sensor options like measuring sound levels,
(3) have access to communication channels like WiFi, Bluetooth and even wired networks.
(4) have a centralised database server,
(5) have a client interface to access information stored in the database,
(6) provide some means of security and authentication,
(7) have some means of managing and monitoring sensor nodes like updating software or detecting new sensor nodes.

All four smart campus implementations meet these high-level requirements.

5.3.2 Functional Equivalence. Observation of the four implementations shows that they operate as expected, e.g. detecting light or motion. To illustrate Figure 8 shows the web interface for the implementations where CWS and CRS are deployed in a different room from PWS and PRS.

All four implementations use an identical set of inexpensive sensors, so we expect the accuracy of the data collected is within tolerance levels. This is validated by comparing PRS and PWS sensor nodes deployed in the same room for some minutes. The measurements show only small variances, e.g. temperatures recorded differ by less than $0.4°C$, and light by less than 1 lux. For this room monitoring application precise timings are not critical, and we don't compare the timing behaviours of the implementations.

5.3.3 Memory and Power Consumption.

Memory. By design sensor nodes are devices with limited computational capacity, and memory is a key restriction. Even supersensors often have less than a GiB of memory, and microcontrollers often have just tens of KiBs.

Just Accepted

As the tierless languages synthesize the code to be executed on the sensor nodes, we need to confirm that the generated code is sufficiently memory efficient.

PWS	PRS	CWS	CRS
20,270	3,557,806	880	2,726,680

Table 1. UoG smart campus sensor nodes: maximum memory residency (bytes).

Just Accepted

Table 1 shows the maximum memory residency after garbage collection of the sensor node for all four smart campus implementations. The smart campus sensor node programs executing on the Wemos microcontrollers have low maximum residencies: 20270 bytes for PWS and 880 bytes for CWS. In CWS the mTask system generates very high level TOP bytecode that is interpreted by the mTask virtual machine and uses a small and predictable amount of heap memory. In PWS, the hand-written MicroPython is compiled to bytecode for execution on the virtual machine. Low residency is achieved with a fixed size heap and efficient memory management. For example both MicroPython and mTask use fixed size allocation units and mark&sweep garbage collection to minimise memory usage at the cost of some execution time [57].

The smart campus sensor node programs executing on the Raspberry Pis have far higher maximum residencies than those executing on the microcontrollers: 3.5MiB for PRS and 2.7MiB for CRS. In CRS the sensor node code is a set of iTask executing on a full-fledged iTask server running in distributed child mode and this consumes far more memory.

In PRS the sensor node program is written in Python, a language far less focused on minimising memory usage than MicroPython. For example an object like a string is larger in Python than in MicroPython and consequently does not support all features such as *f-strings*. Furthermore, not all advanced Python feature regarding classes are available in MicroPython, i.e. only a subset of the Python specification is supported [81].

In summary the sensor node code generated by both tierless languages, iTask and mTask, is sufficiently memory efficient for the target sensor node hardware. Indeed, the maximum residencies of the Clean sensor node code is less than the corresponding hand-written (Micro)Python code. Of course in a tiered stack the hand-written code can be more easily optimised to minimise residency, and this could even entail using a memory efficienthat thet language like C/C++. However, such optimisation requires additional developer effort, and a new language would introduce additional semantic friction.

Power. Sensor nodes and sensors are designed to have low power demands, and this is particularly important if they are operating on batteries. The grey literature consensus is that with all sensors enabled a sensor node should typically have sub-1W peak power draw.

The Wemos sensor nodes used in CWS and PWS have the low power consumption of a typical embedded device: with all sensors enabled, they consume around 0.2W. The Raspberry Pi supersensor node used in CRS and PRS use more power as they have a general purpose ARM processor and run mainstream Linux. With all sensors enabled, they consume 1–2W, depending on ambient load. So a microcontroller sensor node consumes an order of magnitude less power than a supersensor node.

6 IS TIERLESS IOT PROGRAMMING EASIER THAN TIERED?

This section investigates whether tierless languages make IoT programming *easier* by comparing the UoG smart campus implementations. The CRS and CWS implementations allow us to evaluate tierless languages for resource-rich and for resource-constrained sensor nodes respectively. The PRS and PWS allow a like-for-like comparison with tiered Python implementations.

6.1 Comparing Tiered and Tierless Codebases

Code Size. is widely recognised as an approximate measure of the development and maintenance effort required for a software system [63]. SLOC is a common code size metric, and is especially useful for multi-paradigm systems like IoT systems. It is based on the simple principle that the more lines of code, the more developer effort and the increased likelihood of bugs [63]. It is a simple measure, not dependent on some formula, and can be automatically computed [68].

Of course SLOC must be used carefully as it is easily influenced by programming style, language paradigm, and counting method [4]. Here we are counting code to compare development effort, use the same idiomatic programming style in each component, and only count lines of code, omitting comments and blank lines.

Table 2 enumerates the SLOC required to implement the UoG smart campus functionalities in PWS, PRS, CWS and CRS. Both Python and Clean implementations use the same server and communication code for Raspberry Pi and for Wemos sensor nodes (rows 5–7 of the table). The Sensor Interface (SI) refers to code facilitating the communication between the peripherals and the sensor node software. Sensor Node (SN) code contains all other code on the sensor node that does not belong to any another category, such as control flow. Manage Nodes (MN) is code that coordinates sensor nodes, e.g. to add a new sensor node to the system. Web Interface (WI) code provides the web interface from the server, i.e. the presentation layer. Database Interface (DI) code communicates between the server and the database(s). Communication (CO) code provides communication between the server and the sensor nodes, and executes on both sensor node and server, i.e. the network layer.

The most striking information in Table 2 is that *the tierless implementations require far less code than the tiered implementations*. For example 166/562 SLOC for CWS/PWS, or 70% fewer source lines. We attribute the code reduction to three factors: reduced interoperation, automatic communication, and high level programming abstractions. We analyse each of these aspects in the following subsections.

		Tiered Python		Tierless Clean	
Code Location	Functionality	PWS	PRS	CWS	CRS
Sensor Node	Sensor Interface	52	57	11	11
	Sensor Node	178	183	9	4
Server	Manage Nodes	76		35	30
	Web Interface	56		28	
	Database Interface	106		78	
Communication	Communication	94	98	5	4
Total SLOC		562	576	166	155
No. Files		35	38	3	3

Table 2. Comparing tiered and tierless smart campus code sizes: SLOC and number of source files. PWS and CWS execute on resource-constrained sensor nodes, while PRS and CRS execute on resource-rich sensor nodes.

Code Proportions. Comparing the percentages of code required to implement the smart campus functionalities normalises the data and avoids some issues when comparing SLOC for different programming languages, and especially for languages with different paradigms like object-oriented Python and functional Clean. Figure 9 shows the percentage of the total SLOC required to implement the smart campus functionalities in each of the four implementations, and is computed from the data in Table 2. It shows that there are significant differences between the percentage of code for each functionality between the tiered and tierless implementations. For

Just Accepted

example 17% of the tiered implementations specifies communication, whereas this requires only 3% of the tierless implementations, i.e. 6× less. We explore the reasons for this in Section 6.4. The other major difference is the massive percentage of Database Interface code in the tierless implementations: at least 47%. The Smart Campus specification required a standard DBMS, and the Clean/iTask SQL interface occupies some 78 SLOC. While this is a little less than the 106 SLOC used in Python (Table 2), it is a far higher percentage of systems with total codebases of only around 160 SLOC. Idiomatic Clean/iTask would use high level abstractions to store persistent data in an SDS, requiring just a few SLOC. The total size of CWS and CRS would be reduced by a factor of two and the percentage of Database Interface code would be even less than in the tiered Python implementations.

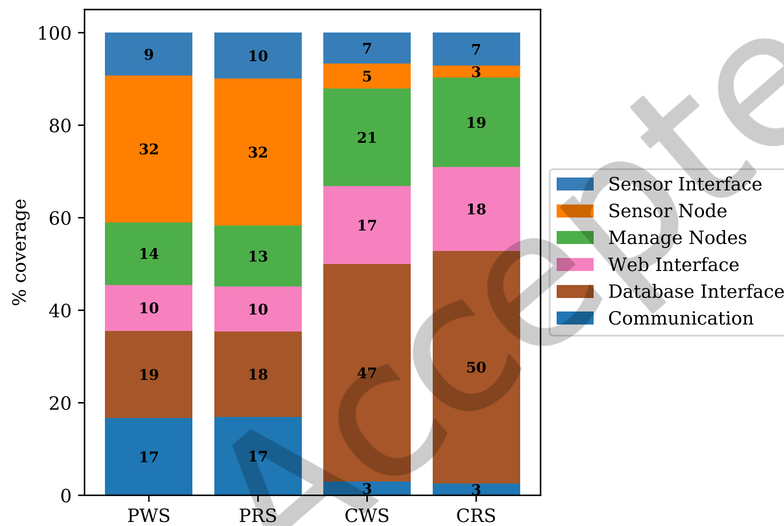

Fig. 9. Comparing the percentage of code required to implement each functionality in tiered/tierless and resource-rich/constrained smart campus implementations.

6.2 Comparing Codebases for Resource-Rich/Constrained Sensor Nodes

Before exploring the reasons for the smaller tierless codebase we compare the implementations for resource-rich and resource-constrained sensor nodes, again using SLOC and code proportions. Table 2 shows that the two tiered implementations are very similar in size: with PWS for microcontrollers requiring 562 SLOC and PRS for supersensors requiring 576 SLOC. The two tierless implementations are also similar in size: CWS requiring 166 and CRS 155 SLOC.

There are several main reasons for the similarity. One is that the server-side code, i.e. for the presentation and application layers, is identical for both resource rich/constrained implementations. The identical server code accounts for approximately 40% of the PWS and PRS codebases, and approximately 85% of the CWS and CRS codebases (Figure 9). For the perception and network layers on the sensor nodes, the Python and MicroPython implementations have the same structure, e.g. a class for each type of sensor, and use analogous libraries. Indeed,

Just Accepted

approaches like CircuitPython [78] allow the same code to execute on both resource-rich and resource-constrained sensor nodes.

Like Python and MicroPython, iTask and mTask are designed to be similar, as elaborated in Section 8. The similarity is apparent when comparing the iTask CRTS and iTask/mTask CWTS room temperature systems in Listings 3 and 4. That is, both implementations use similar SDS data stores and lenses; they have similar `devTasks` that execute on the sensor node, and the server-side `mainTasks` are almost identical: they deploy the remote `devTask` before generating the web page to report the readings.

In both Python and Clean the resource-constrained implementations are less than 7% larger than the resource-rich implementations. This suggests that *the development and maintenance effort of simple IoT systems for resource-constrained and for resource-rich sensor nodes is similar in tierless technologies,* just as it is in tiered technologies. A caveat is that the smart campus system is relatively simple, and developing more complex perception and network code on bare metal may prove more challenging. That is, the lack of OS support, and the restricted languages and libraries, may have greater impact. We return to this issue in Section 8.

6.3 Reduced Interoperation

		Languages				Paradigms	
Code Location	Functionality	PWS	PRS	CWS	CRS	Python	Clean
Sensor Node	Sensor Int.	μPython	Python	mTask	iTask	imp.	decl.
	Sensor Node	μPython	Python	mTask	iTask	imp.	decl.
Server	Manage Nodes	Python, JSON		iTask		imp.	decl.
	Web Int.	HTML, PHP		iTask		both	decl.
	Database Int.	Python,JSON,Redis		iTask		both	decl.
Communication	Communication	μPython	Python	iTask,mTask	iTask	imp.	decl.
	Total	7	6	2	1	2	1

Table 3. Smart Campus implementation languages and paradigm comparison.

The vast majority of IoT systems are implemented using a number of different programming languages and paradigms, and these must be effectively used and interoperated. A major reason that the tierless IoT implementations are simpler and shorter than the tiered implementations is that they use far fewer programming languages and paradigms. Here we use language to distinguish embedded DSLs from their host language: so iTask and mTask are considered distinct from Clean; and to distinguish dialects: so MicroPython is considered distinct from Python.

The tierless implementations use just two conceptually-similar DSLs embedded in the same host language, and a single paradigm (Table 3). In contrast, the tiers in PRS and PWS use six or more very different languages, and both imperative and declarative paradigms. Multiple languages are commonly used in other typical software systems like web stacks, e.g. a recent survey of open source projects reveals that on average at least five different languages are used [48]. Interoperating components in multiple languages and paradigms raises a plethora of issues.

Interoperation *increases the cognitive load on the developer* who must simultaneously think in multiple languages and paradigms. This is commonly known as semantic friction or impedance mismatch [34]. A simple illustration of this is that the tiered PRS source code comprises some 38 source and configuration files, whereas the tierless

Just Accepted

CRS requires just 3 files (Table 2). The source could be structured as a single file, but to separate concerns is structured into three modules, one each for SDSs, types, and control logic [75].

The developer must *correctly interoperate the components*, e.g. adhere to the API or communication protocols between components. The interoperation often entails additional programming tasks like marshalling or demarshalling data between components. For example, in the tiered PRS and PWS architectures, JSON is used to serialise and deserialise data strings from the Python collector component before storing the data in the Redis database (Listing 5).

```
channel = 'sensor_status.   sensor_types.sensor_type_name(s.sensor_type))
       self.r.publish(channel, s.SerializeToString())
```

```
for message in p.listen():
  if message['type'] not in ['message', 'pmessage']:
  continue

try:
 status = collector_pb2.SensorStatus.FromString(message['data'])
```

Listing 5. JSON Data marshalling in PRS and PWS: sensor node above, server below.

To ensure correctness the developer *must maintain type safety* across a range of very different languages and diverse type systems, and we explore this further in Section 7.1. The developer must also deal with the *potentially diverse failure modes*, not only of each component, but also of their interoperation, e.g. if a value of an unexpected type is passed through an API. We explore this further in Section 7.2.

6.4 Automatic Communication

In conventional tiered IoT implementations the developer must write and maintain code to communicate between tiers. For example PRS and PWS create, send and read MQTT [42] messages between the perception and application layers. Table 2 shows that communication between these layers require some 94 SLOC in PWS and 98 in PRS, accounting for 17% of the codebase (bottom bars in Figure 9). To illustrate, Listing 6 shows part of the code to communicate sensor readings from the PWS sensor node to the Redis store on the server.

Not only must the tiered developer write additional code, but IoT communication code is often intricate. In such a distributed system the sender and receiver must be correctly configured, correctly follow the communication protocol through all execution states, and deal with potential failures. For example line 3 of Listing 6: `redis host = config.get('Redis', 'Host')` will fail if either the host or IP are incorrect.

```
def main():
    config.init('mqtt')
    redis_host = config.get('Redis', 'Host')
    redis_port = config.getint('Redis', 'Port')
    r = redis.StrictRedis(host=redis_host, port=redis_port)
    p = r.pubsub()
    p.psubscribe("sensor_status.*")
    for message in p.listen():
        if message['type'] not in ['message', 'pmessage']:
            print "Ignoring message             . . .
```

Listing 6. Tiered Communication Example: MQTT transmission of sensor values in PWS.

Just Accepted

In contrast, the tierless CWS and CRS communication is not only highly automated, but also automatically correct because matching sender and receiver code is generated by the compiler. Table 2 shows that communication is specified in just 5 SLOC in CWS and 4 in CRS, or just 3% of the codebase (bottom bars in Figure 9).

Listing 4 illustrates communication in a tierless IoT language. That is, the CWTS temperature sensor requires just three lines of communication code, and uses just three communication functions. The `withDevice` function on line 30 integrates a sensor node with the server, allowing tasks to be sent to it. The `liftmTask` on line 30 integrates an mTask in the iTask runtime by compiling it and sending it for interpretation to the sensor node. The `liftsds` on line 22 integrates SDSs from iTask into mTask, allowing mTasks to interact with data from the iTask server. The exchange of data, user interface, and communication are all automatically generated.

6.5 High Level Abstractions

Another reason that the tierless Clean implementations are concise is because they use powerful higher order IoT programming abstractions. For comprehensibility the simple temperature sensor from Section 4.4 (Listing 4) is used to compare the expressive power of Clean and Python-based IoT programming abstractions. There are implementations for all four configurations: PRTS (Python Raspberry Pi Temperature Sensor)[1], PWTS (Python Wemos Temperature Sensor)[1] , CRTS[2] and CWTS[2] but as the programming abstractions are broadly similar, we compare only the PWTS and CWTS implementations.

Although the temperature sensor applications are small compared to the smart campus application, they share some typical IoT stack traits. The architecture consists of a server and a single sensor node (Figure 6). The sensor node measures and reports the temperature every ten seconds to the server while the server displays the latest temperature via a web interface to the user.

Table 4 compares the SLOC required for the MicroPython and Clean iTask/mTask Wemos temperature sensors: PWTS and CWTS respectively. The code sizes here should not be used to compare the programming models as implementing such a small application as a conventional IoT stack requires a significant amount of configuration and other machinery that would be reused in a larger application. Hence, the ratio between total PWTS and CWTS code sizes (298:15) is far greater than for realistic applications like PWS and CWS (471:166).

Location	Functionality	PWTS	CWTS	Lines (Listing 4)
Sensor Node	Sensor Interface	14	3	9, 21, 25
	Sensor Node	67	4	8, 23, 24, 26
Server	Web Interface	17	3	11, 31, 32
	Database Interface	106	2	14, 17
Communication	Communication	94	3	7, 22, 30
Total SLOC		298	15	
No. Files		27	1	

Table 4. Comparing Clean and Python programming abstractions using the PWTS and CWTS temperature sensors (SLOC and total number of files.)

The multiple tiers in PRS and PWS provide different levels of abstraction and separation of concerns.

[1]Lubbers, M.; Koopman, P.; Ramsingh, A.; Singer, J.; Trinder, P. (2021): Source code, line counts and memory stats for PRS, PWS, PRT and PWT. Zenodo. 10.5281/zenodo.5081386.

[2]Lubbers, M.; Koopman, P.; Ramsingh, A.; Singer, J.; Trinder, P. (2021): Source code, line counts and memory stats for CRS, CWS, CRTS and CWTS. Zenodo. 10.5281/zenodo.5040754.

Just Accepted

However, there are various ways that high-level abstractions make the CWS much shorter than PRS and PWS implementations.

Firstly, functional programming languages are generally more concise than most other programming languages because their powerful abstractions like higher-order and/or polymorphic functions require less code to describe a computation. Secondly, the TOP paradigm used in iTask and mTask reduces the code size further by making it easy to specify IoT functionality concisely. As examples, the step combinator `>>*.` allows the task value on the left-hand side to be observed until one of the steps is enabled;

and the `viewSharedInformation` (line 31 of Listing 4) part of the UI will be automatically updated when the value of the SDS changes. Moreover, each SDS provides automatic updates to all coupled SDSs and associated tasks. Thirdly, the amount of explicit type information is minimised in comparison to other languages, as much is automatically inferred [33].

7 COULD TIERLESS IOT PROGRAMMING BE MORE RELIABLE THAN TIERED?

This section investigates whether tierless languages make IoT programming more reliable. Arguably the much smaller and simpler code base is inherently more understandable, and more likely to be correct. Here we explore specific language issues, namely those of preserving type safety, maintainability, failure management, and community support.

7.1 Type Safety

Strong typing identifies errors early in the development cycle, and hence plays a crucial role in improving software quality. In consequence almost all modern languages provide strong typing, and encourage static typing to minimise runtime errors.

That said, many distributed system components written in languages that primarily use static typing, like Haskell and Scala, use some dynamic typing, e.g. to ensure that the data arriving in a message has the anticipated type [21, 25].

In a typical tiered multi-language IoT system the developer must integrate software in different languages with very different type systems, and potentially executing on different hardware. The challenges of maintaining type safety have long been recognised as a major component of the semantic friction in multi-language systems, e.g. [34].

Even if the different languages used in two components are both strongly typed, they may attribute, often quite subtly, different types to a value. Such type errors can lead to runtime errors, or the application silently reporting erroneous data. Such errors can be hard to find. Automatic detection of such errors is sometimes possible, but requires an addition tool like Jinn [23, 39].

```
message SensorData {
    enum SensorType { TEMPERATURE = 1; . . .  }
    SensorType sensor_type   = 1;
    uint64     timestamp     = 2;
    double     float_value   = 3;
}
```

```
channel = 'sensor_status.   sensor_types.sensor_type_name(s.sensor_type))
        self.r.publish(channel, s.SerializeToString())
```

Listing 7. PRS loses type safety as a sensor node sends a `double`, and the server stores a `string`.

Analysis of the PRS codebase reveals an instance where it, fairly innocuously, loses type safety. The fragment in Listing 7 first shows a `double` sensor value being sent from the sensor node, and then shows the value being

Just Accepted

```
failover :: [TCPSettings] (Main (MTask BCInterpret a)) → Task a
failover []     _     = throw "Exhausted device pool"
failover [d:ds] mtask = try (withDevice d (liftmTask mtask)) except
where except MTEUnexpectedDisconnect = failover ds mtask
      except _                       = throw e
```

Listing 8. An mTask failover combinator.

stored in Redis as a `string` on the server. As PWS preserves the same server components it also suffers from the same loss of type safety.

A tierless language makes it possible to guarantee type safety across an entire IoT stack. For example the Clean compiler guarantees static type safety as the entire CWS software stack is type checked, and generated, from a single source. Tierless web stack languages like Links [15] and Hop [66] provide the same guarantee for web stacks.

7.2 Failure Management

Some IoT applications, including smart campus and other building monitoring applications, require high sensor uptimes. Hence, if a sensor or sensor node fails the application layer must be notified, so that it can report the failure. In the UoG smart campus system a building manager is alerted to replace the failed device.

In many IoT architectures, including PRS and PWS, detecting failure is challenging because the application layer listens to the devices. When a device comes online, it registered with the application and starts sending data. When a device goes offline again, it could be because the power was out, the device was broken or the device just paused the connection.

If a sensor node fails in CWS, the iTask/mTask combinator interacting with a sensor node will throw an iTask exception. The exception is propagated and a handler can respond, e.g. rescheduling the task on a different device in the room, or requesting that a manager replaces the device. That is, iTask, uses standard succinct declarative exception handling.

In the UoG smart campus application, this can be done by creating a pool of sensor nodes for each room and when a sensor node fails, assign another one to the task.

Listing 8 shows a failover combinator that executes an mTask on one of a pool of sensor nodes. If a sensor node unexpectedly disconnects, the next sensor node is tried until there are no sensor nodes left. If other errors occur they are propagated as usual.

Currently, PRS and PWS both use heartbeats to confirm that the sensor nodes are operational, and will report failures. At the cost of extending the codebase, failover to an alternate sensor node could be provided.

7.3 Maintainability

Far more engineering effort is expended on maintaining a system, than on the initial development. Tiered and tierless IoT systems have very different maintainability properties.

The modularity of the tiered stack makes replacing tiers/components easy. For example in PWS or PRS the MongoDB NoSQL DBMS could be readily be replaced by an alternative like CouchDB. Because a tierless compiler must generate code for components, replacing them may not be so easy. If there are iTask abstractions for the component then replacement is straightforward. For example replacing SQLite with some other SQL DBMS simply entails recompilation of the application.

However incorporating a component that does not yet have a task abstraction, like a NoSQL DBMS, is more involved. That is, a foreign function interface to the new component must be implemented, along with a suitable iTask abstraction for operations on the component.

Just Accepted

Many maintenance tasks are smaller in scale and occur within the components or tiers. Consider a simple change, for example if the temperature value recorded by a sensor changes from integer to real.

All tiers of a tiered stack must be correctly and consistently refactored to reflect the change of temperature data type: so changes at the perception, network, application and presentation layers. A PWS developer works in seven languages and two paradigms to effect the change (Table 2), and must edit many source files. Many programming errors are either detected at runtime when testing the stack, or worse not automatically detected and produce erroneous results.

In a tierless language the source code is much smaller and so it is easier to comprehend, i.e. to understand what refactoring is required. A CWS developer works in only two languages and a single paradigm to effect the change, and will edit no more than three source files (Table 2). Moreover, the compiler will statically detect many programming errors.

More substantial in-component maintenance raises similar issues as for tiered implementations. If the maintenance activity requires a new task combinator, this is readily constructed in iTask, but may require changing the DSL implementation in mTask, i.e. to change the compiler and the bytecode interpreter. That is, mTask is more *brittle* than iTask.

In summary, while a tiered approach makes replacing components easy, refactoring within the components is far harder in a multi-tier multi-language IoT implementation than in a tierless IoT implementation.

7.4 Support

Community and tool support are essential for engineering reliable production software. PRS and PWS are both Python based, and Python/MicroPython are among the most popular programming languages [13]. Python is also a common choice for some tiers of IoT applications [77]. Hence, there are a wide range of development tools like IDEs and debuggers, a thriving community and a wealth of training material. There are even specialised IoT Boards like PyBoard & WiPy that are specifically programmed using Python variations like MicroPython.

In contrast, tierless languages are far less mature than the languages used in tiered stacks, and far less widely adopted. This means that for CWS and CRS there are fewer tools, a far smaller developer community, and less training material available.

CWS and CRS are both written in DSLs embedded in Clean, a fairly stable industrial-grade but niche functional programming language. The DSLs are implemented in Clean but require experimental compiler extensions that are often undocumented. There are few maintainers of the DSLs and documentation is often sparse. Acquiring information about the systems requires distilling academic papers and referring to the source code. There is a Clean IDE, but it does not contain support for the iTask or mTask DSLs.

8 COMPARING TIERLESS LANGUAGES FOR RESOURCE-RICH/CONSTRAINED SENSOR NODES

This section compares two tierless IoT languages: one for resource-rich, and the other for resource-constrained, sensor nodes. Key issues are the extent to which the very significant resource constraints of a microcontroller limit the language, and the benefits of executing on bare metal, i.e. without an OS.

With the tierless Clean technologies described here, iTask are always used to program the application and presentation layers of the IoT stack. So any differences occur in the perception and network layer programming. If sensor nodes have the capacity to support iTask, a tierless IoT system can be constructed in Clean using only iTask, as in CRS. Alternatively for sensor nodes with low computational power, like typical microcontrollers, mTask is used for the perception and network layers, as in CWS.

This section compares the iTask and mTask embedded DSLs, with reference to CRS and CWS as exemplars. Table 5 summarises the differences between the Clean embedded IoT DSLs and their host language.

Just Accepted

Property	Clean	iTask	mTask
Function for an IoT System	Host Language	Specify distributed workflows	Specify sensor node workflow
Referentially transparent	Yes	Yes	Yes
Evaluation strategy	Lazy	Lazy	Strict
Higher-order functions	Yes	Yes	No
User-defined datatypes	Yes	Yes	No
Task oriented	No	Yes	Yes
Higher-order tasks	–	Yes	No
Execution Target	Commodity PC	Commodity PC and Browser	Microcontroller
Language Implementation	Compiled or interpreted	Compiled and interpreted	Interpreted

Table 5. Comparing tierless IoT languages for resource-rich sensor nodes (iTask embedded DSL), for resource-constrained sensor nodes (mTask embedded DSL), and their Clean host language.

Just Accepted

8.1 Language Restrictions for Resource-Constrained Execution

Executing components on a resource-constrained sensor node imposes restrictions on programming abstractions available in a tierless IoT language or DSL. The small and fixed-size memory are key limitations. The limitations are shared by any high-level language that targets microcontrollers such as BIT, PICBIT, PICOBIT, Microscheme and uLisp [18, 22, 35, 71, 76]. Even in low level languages some language features are disabled by default when targeting microcontrollers, such as runtime type information (RTTI) in C++.

Here we investigate the restrictions imposed by resource-constrained sensor nodes on mTask, in comparison with iTask. While iTask and mTask are by design superficially similar languages, to execute on resource-constrained sensor nodes mTasks are more restricted, and have a different semantics.

mTask programs do not support user defined higher order functions, the only higher order functions available are the predefined mTask combinators. Programmers can, however, use any construct of the Clean host language to construct an mTask program, including higher order functions and arbitrary data types. For example folding an mTask combinator over a list of tasks. The only restriction is that any higher order function must be macro expanded to a first order mTask program before being compiled to bytecode.

As an example in Listing 4 we use `temperature dht >>~. setSds localSds` instead of `temperature dht >>~. λ temp → setSds localSds temp`

In contrast to iTask, mTask programs have no user defined or recursive data types. It is possible to add user defined types—as long as they are not sum types—to mTask, but this requires significant programming effort. Due to the language being shallowly embedded, pattern matching and field selection on user defined types is not readily available and thus needs to be built into the language by hand. Alleviating this limitation remains future work.

mTask programs mainly use strict rather than lazy evaluation to minimise the requirement for a variable size heap. This has no significant impact for the mTask programs we have developed here, nor in other IoT applications we have engineered.

mTask abstractions are less easily extended than iTask. For example iTask can be extended with a new combinator that composes a specific set of tasks for some application. Without higher order functions the equivalent combinator can often not be expressed in mTask, and adding it to mTask requires extending the DSL

rather than writing a new definition in it. On the other hand, it is possible to outsource this logic to the iTask program as mTask and iTask tasks are so tightly integrated.

8.2 The Benefits of a Bare Metal Execution Environment

Despite the language restrictions, components of a tierless language executing on a microcontroller can exploit the bare metal environment. Many of these benefits are shared by other bare metal languages like MicroPython or C/C++. So as mTask executes on bare metal it has some advantages over iTask. Most notably mTask has better control of timing as on bare metal there are no other processes or threads that compete for CPU cycles. This makes the mTask `repeatEvery` (Listing 4, line 24) much more accurate than the iTask `waitForTimer` (Listing 3, line 13). While exact timing is not important in this example, it is significant for many other IoT applications.

In contrast iTask cannot give real time guarantees. One reason is that an iTask server can ship an arbitrary number of iTask or mTask tasks to a device. Such competing tasks, or indeed other OS threads and processes, consume processor time and reduce the accuracy of timings. However, even when using multiple mTasks, it is easier to control the number of tasks on a device than controlling the number of processes and threads executing under an OS.

An mTask program has more control over energy consumption. The mTask embedded DSL and the mTask run-time system (RTS) are designed to minimise energy usage [16]. Intensional analysis of the declarative task description and current progress at run time allow the RTS to schedule tasks and maximise idle time. As the RTS is the only program running on the device, it can enforce deep sleep and wake up without having to worry about influencing other processes.

The mTask RTS has direct control of the peripherals attached to the microcontroller, e.g. over GPIO pins. There is no interaction with, or permission required from, the OS. Moreover, microcontrollers typically have better support for hardware interrupts, reducing the need to poll peripherals.

The downside of this direct control is that CWS has to handle some exceptions that would otherwise be handled by the OS in CRS and hence the device management code is longer: 28 versus 20 SLOC in Table 2.

8.3 Summary

Table 5 summarises the differences between the Clean IoT embedded DSLs and their host language. The restrictions imposed by a resource-constrained execution environment on the tierless IoT language are relatively minor. Moreover the mTask programming abstraction is broadly compatible with iTask. As a simple example compare the iTask and mTask temperature sensors in Listings 3 and 4. As a more realistic example, the mTask based CWS smart campus implementation is similar to the iTask based CRS, and requires less than 10% additional code: 166 SLOC compared with 155 SLOC (Table 2).

Even with these restrictions, mTask programming is at a far higher level of abstraction than almost all bare metal languages, e.g. BIT, PICBIT, PICOBIT and Microscheme. That is mTask provides a set of higher order task combinators, shared distributed data stores, etc. (Section 4.4). Moreover, it seems that common sensor node programs are readily expressed using mTask. In addition to the CWTS and CWS systems outlined here, other case studies include Arduino examples as well as some bigger tasks [38, 44, 45]. We conclude that the programming of sensor tasks is well-supported by both DSLs.

9 CONCLUSION

9.1 Summary

We have conducted a systematic comparative evaluation of two tierless language technologies for IoT stacks: one for resource-rich, and the other for resource-constrained sensor nodes. The basis is four implementations of a deployed smart campus IoT stack: two conventional tiered and Python-based stacks, and two tierless Clean

Just Accepted

stacks. An operational comparison of implementations demonstrates that they have equivalent functionality, and meet the UoG smart campus functional requirements (Section 5).

We show that *tierless languages have the potential to significantly reduce the development effort for IoT systems*. Specifically the tierless CWS and CRS stacks require far less code, i.e. 70% fewer source lines, than the tiered PWS and PRS stacks (Table 2). We analyse the code reduction and attribute it to the following three main factors. (1) Tierless developers need to manage less interoperation: CRS uses a single DSL and paradigm, and CWS uses two DSLs in a single paradigm and three source code files. In contrast, both PRS and PWS use at least six languages in two paradigms and spread over at least 35 source code files (Tables 2 and 3). Thus, a tierless stack minimises semantic friction. (2) Tierless developers benefit from automatically generated, and hence correct, communication (Listing 4), and write 6× less communication code (Figure 9).

(3) Tierless developers can exploit powerful high-level declarative and task-oriented IoT programming abstractions (Table 4), specifically the composable, higher-order task combinators outlined in Section 4.2. Our empirical results for IoT systems are consistent with the benefits claimed for tierless languages in other application domains. Namely that a tierless language provides a *Higher Abstraction Level*, *Improved Software Design*, and improved *Program Comprehension* [82].

We show that *tierless languages have the potential to significantly improve the reliability of IoT systems*. We illustrate how Clean maintains type safety, contrasting this with a loss of type safety in PRS. We illustrate higher order failure management in Clean iTask/mTask in contrast to the Python-based failure management in PRS. For maintainability a tiered approach makes replacing components easy, but refactoring within the components is far harder than in a tierless IoT language. Again our findings are consistent with the simplied *Code Maintenance* benefits claimed for tierless languages [82].

Finally, we contrast community support for the technologies (Section 7).

We report *the first comparison of a tierless IoT codebase for resource-rich sensor nodes with one for resource-constrained sensor nodes*. (1) The tierless implementations have very similar code sizes (SLOC), as do the tiered implementations: less than 7% difference in Table 2. This suggests that the development and maintenance effort of simple tierless IoT systems for resource-constrained and for resource-rich sensor nodes is similar, as it is for tiered technologies. (2) The percentages of code required to implement each IoT functionality in the tierless Clean implementations is very similar as it is in the tiered Python implementations (Figure 9). This suggests that the code for resource-constrained and resource-rich sensor nodes can be broadly similar in tierless technologies, as it is in many tiered technologies (Section 6.2).

We present *the first comparison of two tierless IoT languages: one designed for resource-constrained sensor nodes (Clean with iTask and mTask), and the other for resource-rich sensor nodes (Clean with iTask).* Clean with iTask can implement all layers of the IoT stack if the sensor nodes have the computational resources, as the Raspberry Pis do in CRS. On resource constrained sensor nodes mTask are required to implement the perception and network layers, as on the Wemos minis in CWS. We show that a bare metal execution environment allows mTask to have better control of peripherals, timing and energy consumption. The memory available on a microcontroller restricts the programming abstractions available in mTask to a fixed set of combinators, no user defined or recursive data types, strict evaluation, and makes it harder to add new abstractions. Even with these restrictions mTask provide a higher level of abstraction than most bare metal languages, and can readily express many IoT applications including the CWS UoG smart campus application (Section 8). Our empirical results are consistent with the benefits of tierless languages listed in Section 2.1 of [82].

9.2 Reflections

This study is based on a specific pair of tierless IoT languages, and the Clean language frameworks represent a specific set of tierless language design decisions. Many alternative tierless IoT language designs are possible, and

Just Accepted

some are outlined in Section 3.3. Crucially the limitations of the tierless Clean languages, e.g. that they currently provide limited security, should not be seen as limitations of tierless technologies in general.

This study has explored some, but not all, of the potential benefits of tierless languages for IoT systems. An IoT system specified as a single tierless program is amenable to a host of programming language technologies. For example, if the language has a formal semantics, as Links, Hop and Clean tasks do [15, 59, 66], it is possible to prove properties of the system, e.g. [72]. As another example program analyses can be applied, and Section 3.3 and [82] outline some of the analyses could be, and in some cases have been, used to improve IoT systems. Examples include automatic tier splitting [56], and controlling information flow to enhance security [80].

While offering real benefits for IoT systems development, tierless languages also raise some challenges. Programmers must master new tierless programming abstractions, and the semantics of these automatic multi-tier behaviours are necessarily relatively complex. In the Clean context this entails becoming proficient with the iTask and mTask DSLs. Moreover, specifying a behaviour that is not already provided by the tierless language requires either a workaround, or extending a DSL. However, implementing the relatively simple smart campus application required no such adaption. Finally, tierless IoT technology is very new, and both tool and community support have yet to mature.

9.3 Future Work

This paper is a technology comparison between tiered and tierless technologies. The metrics reported, such as code size, numbers of source code files, and of paradigms are only indirect, although widely accepted, measures of development effort. A more convincing evaluation of tierless technologies could be provided by conducting a carefully designed and substantial user study, e.g. using N-version programming.

A study that implemented common benchmarks or a case study in multiple tierless IoT languages would provide additional evidence for the generality of the tierless approach. Such a study would enable the demonstration and comparison of alternative design decisions within tierless languages, as outlined in Section 3.3.

In ongoing work we are extending the mTask system in various ways. One extension allows mTasks to communicate directly, rather than via the iTask server. Another provides better energy management, which is crucial for battery powered sensor nodes.

Just Accepted

ACKNOWLEDGMENTS

Thanks to Kristian Hentschel and Dejice Jacob who developed and maintain PRS and to funders: Royal Netherlands Navy, the Radboud-Glasgow Collaboration Fund, and UK EPSRC grants MaRIONet (EP/P006434) and STARDUST (EP/T014628). We also thank Lito Michala, Jose Cano, Greg Michaelson, Rinus Plasmeijer, and the anonymous TIOT reviewers for valuable feedback on the paper.

REFERENCES

[1] Peter Achten. 2007. Clean for Haskell98 Programmers – A Quick Reference Guide –. (July 13 2007). http://www.st.cs.ru.nl/papers/2007/CleanHaskellQuickGuide.pdf

[2] Nada Alhirabi, Omer Rana, and Charith Perera. 2021. Security and Privacy Requirements for the Internet of Things: A Survey. *ACM Trans. Internet Things* 2, 1, Article 6 (Feb 2021), 37 pages. https://doi.org/10.1145/3437537

[3] Artem Alimarine and Rinus Plasmeijer. 2002. A Generic Programming Extension for Clean. In *Implementation of Functional Languages*, Thomas Arts and Markus Mohnen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–185.

[4] Kalev Alpernas, Yotam M. Y. Feldman, and Hila Peleg. 2020. The Wonderful Wizard of LoC: Paying Attention to the Man behind the Curtain of Lines-of-Code Metrics. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 146–156. https://doi.org/10.1145/3426428.3426921

[5] Mandla Alphonsa. 2021. A Review on IOT Technology Stack, Architecture and Its Cloud Applications in Recent Trends. In *ICCCE 2020*, Amit Kumar and Stefan Mozar (Eds.). Springer Singapore, Singapore, 703–711. https://doi.org/10.1007/978-981-15-7961-5_67

[6] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[7] Emmanuel Baccelli, Joerg Doerr, Ons Jallouli, Shinji Kikuchi, Andreas Morgenstern, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. 2018. Reprogramming Low-end IoT Devices from the Cloud. In *2018 3rd Cloudification of the Internet of Things (CIoT)*. IEEE, Paris, France, 1–6. https://doi.org/10.1109/CIOT.2018.8627129

[8] Vincent Balat. 2006. Ocsigen: Typing Web Interaction with Objective Caml. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA). Association for Computing Machinery, New York, NY, USA, 84–94. https://doi.org/10.1145/1159876.1159889

[9] Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science* 6, 6 (1996), 579–612. https://doi.org/10.1017/S0960129500070109

[10] Alvine Boaye Belle, Ghizlane El-Boussaidi, Christian Desrosiers, and Hafedh Mili. 2013. The layered architecture revisited: Is it an optimization problem?. In *Proceedings of the Twenty-Fifth International Conference on Software Engineering & Knowledge E*, Vol. 1. KSI Research Inc, Boston, MA, USA, 344–349.

[11] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. 2011. Composing Reactive GUIs in F# Using WebSharper. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 203–216. https://doi.org/10.1007/978-3-642-24276-2_13

[12] Michel Boer, de. 2020. *Secure Communication Channels for the mTask System.* Bachelor's Thesis. Radboud University, Nijmegen.

[13] Stephen Cass. 2020. The top programming languages: Our latest rankings put Python on top-again-[Careers]. *IEEE Spectrum* 57, 8 (2020), 22–22. https://doi.org/10.1109/MSPEC.2020.9150550

[14] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 31–44. https://doi.org/10.1145/1323293.1294265

[15] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

[16] Sjoerd Crooijmans. 2021. *Reducing the Power Consumption of IoT Devices in Task-Oriented Programming.* Master's thesis. Radboud University, Nijmegen. https://doi.org/10.1109/ASE.1999.802242

[17] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proceedings of the 26th International Symposium on Implementation and Application of Functional Languages* (Boston, MA, USA) *(IFL '14)*. Association for Computing Machinery, New York, NY, USA, Article 9, 11 pages. https://doi.org/10.1145/2746325.2746333

[18] Danny Dubé. 2000. BIT: A very compact Scheme system for embedded applications. In *Proceedings of the Workshop on Scheme and Functional Programming*. Rice University, Houston, TX, 1–9. http://www.schemeworkshop.org/2000/dube.ps Available as Rice Technical Report 00-368.

[19] Anton Ekblad and Koen Claessen. 2014. A Seamless, Client-Centric Programming Model for Type Safe Web Applications. *SIGPLAN Not.* 49, 12 (sep 2014), 79–89. https://doi.org/10.1145/2775050.2633367

[20] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. 2015. Guilt Free Ivory. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2804302.2804318

[21] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell* (Tokyo, Japan) *(Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 118–129. https://doi.org/10.1145/2034675.2034690

[22] Marc Feeley and Danny Dubé. 2003. PICBIT: A Scheme system for the PIC microcontroller. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*. University of Utah, Salt Lake City, UT, 7–15. http://www.iro.umontreal.ca/~feeley/papers/FeeleyDubeSW03.pdf Proceedings available as Technical Report UUCS-03-023.

[23] Michael Furr and Jeffrey S. Foster. 2005. Checking Type Safety of Foreign Function Calls. *SIGPLAN Not.* 40, 6 (June 2005), 62–72. https://doi.org/10.1145/1064978.1065019

[24] Dominique Guinard and Vlad Trifa. 2016. *Building the Web of Things: With Examples in Node.Js and Raspberry Pi* (1st ed.). Manning Publications Co., USA.

[25] Munish Gupta. 2012. *Akka essentials.* Packt Publishing, Birmingham, England.

[26] Cordelia Hall, Kevin Hammond, Will Partain, Simon L Peyton Jones, and Philip Wadler. 1993. The Glasgow Haskell compiler: a retrospective. In *Functional Programming, Glasgow 1992 (Workshops in Computing)*. Springer, Berlin, 62–71. https://doi.org/10.1007/978-1-4471-3215-8_6

[27] Natascha Harth, Christos Anagnostopoulos, and Dimitrios Pezaros. 2018. Predictive intelligence to the edge: impact on edge analytics. *Evolving Systems* 9, 2 (2018), 95–118. https://doi.org/10.1007/s12530-017-9190-z

Just Accepted

[28] HaskellWiki. 2020. Introduction to IO — HaskellWiki,. https://wiki.haskell.org/index.php?title=Introduction_to_IO&oldid=63493 [Online; accessed 19-January-2021].

[29] Kristian Hentschel, Dejice Jacob, Jeremy Singer, and Matthew Chalmers. 2016. Supersensors: Raspberry Pi Devices for Smart Campus Infrastructure. In *4th International Conference on Future Internet of Things and Cloud, FiCloud 2016*, Muhammad Younas, Irfan Awan, and Winston Seah (Eds.). IEEE, Vienna, Austria, 58–62. https://doi.org/10.1109/FiCloud.2016.16

[30] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. 2019. Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet. In *Network and Distributed Systems Security (NDSS) Symposium 2019*. Internet Society, San Diego, CA, USA, 15. https://doi.org/10.14722/ndss.2019.23488

[31] Joey Hess. 2020. arduino-copilot: Arduino programming in haskell using the Copilot stream DSL. //hackage.haskell.org/package/arduino-copilot

[32] Ralf Hinze. 2000. A New Approach to Generic Functional Programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA, USA) *(POPL '00)*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/325694.325709

[33] John Hughes. 1989. Why functional programming matters. *Comput. J.* 32, 2 (1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[34] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. 2009. A classification of object-relational impedance mismatch. In *2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*. IEEE, IEEE, Gosier, France, 36–43. https://doi.org/10.1109/DBKDA.2009.11

[35] David Johnson-Davies. 2020. Lisp for microcontrollers. https://ulisp.com

[36] P. S. Kochhar, D. Wijedasa, and D. Lo. 2016. A Large Scale Study of Multiple Programming Languages and Code Quality. In *23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, Osaka, Japan, 563–573. https://doi.org/10.1109/SANER.2016.112

[37] Ravi Kishore Kodali and Kopulwar Shishir Mahesh. 2016. Low cost ambient monitoring using ESP8266. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*. IEEE, IEEE, Greater Noida, India, 779–782. https://doi.org/10.1109/IC3I.2016.7918788

[38] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. ACM Press, Vienna, Austria, 1–11. https://doi.org/10.1145/3183895.3183902

[39] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. 2010. Jinn: Synthesizing Dynamic Bug Detectors for Foreign Language Interfaces. *SIGPLAN Not.* 45, 6 (June 2010), 36–49. https://doi.org/10.1145/1809028.1806601

[40] Jong Kook Lee, Seung Jae Jung, Soo Dong Kim, Woo Hyun Jang, and Dong Han Ham. 2001. Component identification method with coupling and cohesion. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*. IEEE, IEEE, Macao, China, 79–86. https://doi.org/10.1109/APSEC.2001.991462

[41] Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. *ACM Sigplan Notices* 37, 10 (2002), 85–95. https://doi.org/10.1145/605432.605407

[42] Roger Light. 2017. Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software* 2, 13 (2017), 265.

[43] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2019. Interpreting Task Oriented Programs on Tiny Computers. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL '19)*, Jurriën Stutterheim and Wei Ngan Chin (Eds.). Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3412932.3412936 event-place: Singapore, Singapore.

[44] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2019. Multitasking on Microcontrollers using Task Oriented Programming. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, Opatija, Croatia, 1587–1592. https://doi.org/10.23919/MIPRO.2019.8756711

[45] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. in-press. Writing Internet of Things applications with Task Oriented Programming. In *Central European Functional Programming School: 8th Summer School, CEFP 2019, Budapest, Hungary, July 17–21, 2019, Revised Selected Papers*. Springer International Publishing, Budapest, Hungary, 51. https://doi.org/10.1145/3310232.3310239

[46] Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. 2020. Tiered versus Tierless IoT Stacks: Comparing Smart Campus Software Architectures. In *Proceedings of the 10th International Conference on the Internet of Things* (Malmö, Sweden) *(IoT '20)*. Association for Computing Machinery, New York, NY, USA, Article 21, 9 pages. https://doi.org/10.1145/3410992.3411002

[47] Alan MacCormack, John Rusnak, and Carliss Y Baldwin. 2007. *The impact of component modularity on design evolution: Evidence from the software industry.* Technical Report 08-038. Harvard Business School Technology and Operations Management Unit, Boston, Massachusetts. https://doi.org/10.2139/ssrn.1071720

[48] Philip Mayer and Alexander Bauer. 2015. An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering* (Nanjing, China) *(EASE '15)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2745802.2745805

[49] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1 (2017), 1. https://doi.org/10.1186/s40411-017-0035-z

Just Accepted

[50] Daniele Mazzei, Giacomo Baldi, Gabriele Montelisciani, and Gualtiero Fantoni. 2018. A full stack for quick prototyping of IoT solutions. *Annals of Telecommunications* 73, 7-8 (2018), 439–449. https://doi.org/10.1007/s12243-018-0644-5

[51] Rebeca C. Motta, Káthia M. de Oliveira, and Guilherme H. Travassos. 2018. On Challenges in Engineering IoT Software Systems. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering* (Sao Carlos, Brazil) *(SBES '18)*. Association for Computing Machinery, New York, NY, USA, 42–51. https://doi.org/10.1145/3266237.3266263

[52] Henry Muccini and Mahyar Tourchi Moghaddam. 2018. IoT Architectural Styles. In *Software Architecture*, Carlos E. Cuesta, David Garlan, and Jennifer Pérez (Eds.). Springer International Publishing, Cham, 68–85. https://doi.org/10.1007/978-3-030-00761-4_5

[53] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695 event-place: Pittsburgh, Pennsylvania.

[54] Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages*. ACM, Bristol, UK, 7.

[55] Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

[56] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 69–81. https://doi.org/10.1145/2661136.2661146

[57] Sebastian Plamauer and Martin Langer. 2017. Evaluation of micropython as application layer programming language on cubesats. In *ARCS 2017; 30th International Conference on Architecture of Computing Systems*. VDE, VDE, Vienna, Austria, 1–9.

[58] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*. ACM, Freiburg, Germany, 141–152.

[59] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming* (Leuven, Belgium) *(PPDP '12)*. Association for Computing Machinery, New York, NY, USA, 195–206. https://doi.org/10.1145/2370776.2370801

[60] Arvind Ravulavaru. 2018. *Enterprise internet of things handbook : build end-to-end IoT solutions using popular IoT platforms*. Packt Publishing, Birmingham, UK.

[61] Clean 3.0 Language Report. 2020. Uniqueness Typing. https://cloogle.org/doc/#_9 [Online; accessed 02-February-2021].

[62] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. 2008. Comparing Libraries for Generic Programming in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/1411286.1411301

[63] Jarrett Rosenberg. 1997. Some misconceptions about lines of code. In *Proceedings fourth international software metrics symposium*. IEEE, IEEE, Albuquerque, NM, USA, 137–142. https://doi.org/10.1109/METRIC.1997.637174

[64] Abhiroop Sarkar and Mary Sheeran. 2020. Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming* (Bologna, Italy) *(PPDP '20)*. Association for Computing Machinery, New York, NY, USA, Article 12, 16 pages. https://doi.org/10.1145/3414080.3414092

[65] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: a functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity*. Association for Computing Machinery, New York, NY, USA, 36–44. https://doi.org/10.1145/2892664.2892670

[66] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: a language for programming the web 2.0. In *OOPSLA Companion*. ACM, Portland, Oregon, USA, 975–985.

[67] Pallavi Sethi and Smruti R Sarangi. 2017. Internet of things: architectures, protocols, and applications. *Journal of Electrical and Computer Engineering* 2017 (2017), 25 pages. https://doi.org/10.1155/2017/9324035

[68] Steven D Sheetz, David Henderson, and Linda Wallace. 2009. Understanding developer and manager perceptions of function points and source lines of code. *Journal of Systems and Software* 82, 9 (2009), 1540–1549. https://doi.org/10.1016/j.jss.2009.04.038

[69] Kazuhiro Shibanai and Takuo Watanabe. 2018. Distributed Functional Reactive Programming on Actor-Based Runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/3281366.3281370 event-place: Boston, MA, USA.

[70] Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. 2012. Drop the Phone and Talk to the Physical World: Programming the Internet of Things with Erlang. In *Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications* (Zurich, Switzerland) *(SESENA '12)*. IEEE Press, Piscataway, NJ, 8–14. https://doi.org/10.1109/SESENA.2012.6225763

Just Accepted

[71] Vincent St-Amour and Marc Feeley. 2009. PICOBIT: a compact scheme system for microcontrollers. In *International Symposium on Implementation and Application of Functional Languages*. Springer, South Orange, NJ, USA, 1–17.

[72] Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019. TopHat: A Formal Foundation for Task-Oriented Programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming* (Porto, Portugal) *(PPDP '19)*. Association for Computing Machinery, New York, NY, USA, Article 17, 13 pages. https://doi.org/10.1145/3354166.3354182

[73] Isaac Strack. 2015. *Getting Started with Meteor.js JavaScript Framework* (2. ed.). Packt Publishing, Birmingham, England.

[74] Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2018. Maintaining Separation of Concerns Through Task Oriented Software Development. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Vol. 10788. Springer, Cham, 19–38. https://doi.org/10.1007/978-3-319-89719-6_2

[75] Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2018. Maintaining Separation of Concerns Through Task Oriented Software Development. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 19–38. https://doi.org/10.1007/978-3-319-89719-6_2

[76] Ryan Suchocki and Sara Kalvala. 2015. Microscheme: Functional programming for the Arduino. In *Proceedings of the 2014 Scheme and Functional Programming Workshop*. University of Indiana, Washington DC, USA, 9.

[77] Giacomo Tanganelli, Carlo Vallati, and Enzo Mingozzi. 2015. CoAPthon: Easy development of CoAP-based IoT applications with Python. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, Milan, Italy, 63–68. https://doi.org/10.1109/WF-IoT.2015.7389028

[78] CircuitPython Team. 2022. CircuitPython. https://circuitpython.org/ [Online; accessed 2-March-2022].

[79] Christophe Troyer, de, Jens Nicolay, and Wolfgang Meuter, de. 2018. Building IoT Systems Using Distributed First-Class Reactive Programming. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, Nicosia, Cyprus, 185–192. https://doi.org/10.1109/CloudCom2018.2018.00045

[80] Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. 2020. Towards Secure IoT Programming in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. Association for Computing Machinery, New York, NY, USA, 136–150. https://doi.org/10.1145/3406088.3409027

[81] Micropython Official Website. 2022. MicroPython Differences from CPython. https://docs.micropython.org/en/latest/genrst/index.html.

[82] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A survey of multitier programming. *ACM Computing Surveys (CSUR)* 53, 4 (2020), 1–35. https://doi.org/10.1145/3397495

[83] Mark Wijkhuizen. 2018. *Security analysis of the iTasks framework*. Bachelor's Thesis. Radboud University, Nijmegen. http://www.ru.nl/publish/pages/769526/arjan_oortgiese.pdf

[84] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers. 2002. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)* 20, 3 (2002), 283–328. https://doi.org/10.1145/566340.566343

Just Accepted