Tiered versus Tierless IoT Stacks: Comparing Smart Campus Software Architectures

Mart Lubbers
Pieter Koopman
Radboud University
Nijmegen, The Netherlands
firstname@cs.ru.nl

Adrian Ramsingh
Jeremy Singer
Phil Trinder
University of Glasgow
Glasgow, United Kingdom
firstname.lastname@glasgow.ac.uk

ABSTRACT

Internet of Things (IoT) software stacks are notoriously complex, conventionally comprising multiple tiers/components and requiring that the developer not only uses multiple programming languages, but also correctly interoperate the components. A novel alternative is to use a single *tierless* language with a compiler that generates the code for each component, and for their correct interoperation.

We report the first ever systematic comparison of tiered and tierless IoT software architectures. The comparison is based on two implementations of a non-trivial smart campus application. PRSS has a conventional tiered Python-based architecture, and Clean Wemos Super Sensors (CWSS) has a novel tierless architecture based on Clean and the iTask and mTask embedded DSLs. An operational comparison of CWSS and PRSS demonstrates that they have equivalent functionality, and that both meet the University of Glasgow (UoG) smart campus requirements.

Crucially, the tierless CWSS stack requires 70% less code than the tiered PRSS stack. We analyse the impact of the following three main factors. (1) Tierless developers need to manage less interoperation: CWSS uses two DSLs in a single paradigm where PRSS uses five languages and three paradigms. (2) Tierless developers benefit from automatically generated, and hence correct, communication. (3) Tierless developers can exploit the powerful high-level abstractions such as Task Oriented Programming (TOP) in CWSS. A far smaller and single paradigm codebase improves software quality, dramatically reduces development time, and improves the maintainability of tierless stacks.

CCS CONCEPTS

• **Software and its engineering** → *Functional languages*; • **Computer systems organization** → *n-tier architectures*; • **Hardware** → **Sensors and actuators**.

KEYWORDS

Internet of Things, Network Reliability, Domain Specific Languages, Software Architectures

ACM Reference Format:
Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. 2020. Tiered versus Tierless IoT Stacks: Comparing Smart Campus Software Architectures. In *10th International Conference on the Internet of Things (IoT '20), October 6–9, 2020, Malmö, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3410992.3411002

1 INTRODUCTION

Conventional IoT software stacks comprise multiple tiers and components and pose very significant software development and maintenance challenges. This is due to the nature of typical IoT applications that must read sensor data, aggregate and select the data, communicate over a network, store the data in a database, and analyse and display views of the data, commonly on webpages.

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) The developer must be fluent in all of the languages, components and their interactions. That is, the developer must correctly use multiple languages that have different paradigms and type systems. (2) The developer must correctly adhere to communication protocols between sensor nodes and server. (3) The developer must deal with the failure modes of each component.

A radical alternative software architecture uses a single language that generates all components/tiers in the IoT stack. Such *tierless* languages are more common for web stacks, e.g. Links [4] or Hop [17]. In a tierless architecture 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.

Potentially a tierless software architecture both reduces the development effort and improves correctness as correct interoperation and communication is automatically generated by the compiler. It may, however, introduce other problems: is the generated code time, space and power efficient?

This paper reports the first ever systematic comparison of tiered and tierless IoT software architectures, to the best of our knowledge. The comparison is based on two implementations of a non-trivial smart campus application. Like many universities, the University of Glasgow seeks to better utilise their built environment by deploying sensors and analyse the information recorded. A prototype smart campus sensor system has been deployed that uses a conventional tiered Python-based architecture on Raspberry Pi Super Sensors (PRSS)[1] [9]. The PRSS sensor nodes have a typical set of smart room sensors: air quality, ambient light, motion, sound, temperature and humidity. Approximately 12 rooms are currently equipped with PRSS sensor nodes (Section 2).

We use a tierless language for IoT programming that comprises two shallowly-embedded DSLs hosted in Clean: iTask to program the web server, and mTask to program the sensor nodes. The entire IoT software stack is a single Clean program. The program fragments to be executed on sensor nodes are compiled at runtime to bytecode and transmitted to dynamically selected sensor nodes for execution.

The same UoG smart campus functionality as PRSS is implemented using iTask/mTask. It is deployed on lighter weight, and more conventional sensor node hardware, namely ESP8266X based Wemos D1 mini nodes. We denote this system the CWSS[2] (Section 2).

We undertake an operational comparison of CWSS and PRSS. To ensure that we are comparing software stacks with equivalent functionality we demonstrate that CWSS and PRSS meet the functional requirements for UoG smart campus. As a tierless language implementation generates the code to be executed on the sensor node, this code may not be as power and memory efficient as hand-written code, so we also measure these two aspects (Section 3).

We undertake a systematic programming comparison of the CWSS and PRSS software architectures. Code size is widely recognised as a measure of the development effort and maintainability of a software system [16]. We examine aspects like code size, number of paradigms used, and interoperation issues like type safety, communication, and the dynamic management of failed sensor nodes (Section 4).

[1]Available at https://bitbucket.org/jsinger/anyscale-sensors

[2]Available at ftp://ftp.cs.ru.nl/pub/Clean/mTask/IOT2020/CWSS.tgz

2 BACKGROUND

IoT software stacks and interoperation

Traditional IoT application stacks involve the interoperation of software components distributed over multiple distributed physical devices. Physical devices include hardware like embedded sensors, actuators and transceivers while software components typically include web interfaces, web servers and databases [18].

These stacks are generally deployed using a layered or tiered architecture. The number of tiers for each IoT application differs depending on the functional requirements and complexity [18]. Here we consider a four-tier IoT stack comprising perception, network, application and presentation layers. Sometimes a fifth, business layer, is added [14].

Specifically we study the lower four tiers of the PRSS and CWSS stacks, as illustrated in Figure 1. These comprise the following tiers. (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 – responsible for the communication 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. (4) Presentation Layer – utilises web components as the interface between the human and devices where application services are provided.

IoT stacks commonly use architectural design patterns like client-server, peer-to-peer, layered (tiered) and microkernel [14]. While both PRSS and CWSS use a client-server architecture, PRSS is tiered and CWSS is tierless.

UoG smart campus

The UoG is partway through a ten-year campus upgrade programme, and a key goal is to embed smart sensing infrastructure into the new physical fabric. As a prototyping exercise, we use low-power 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

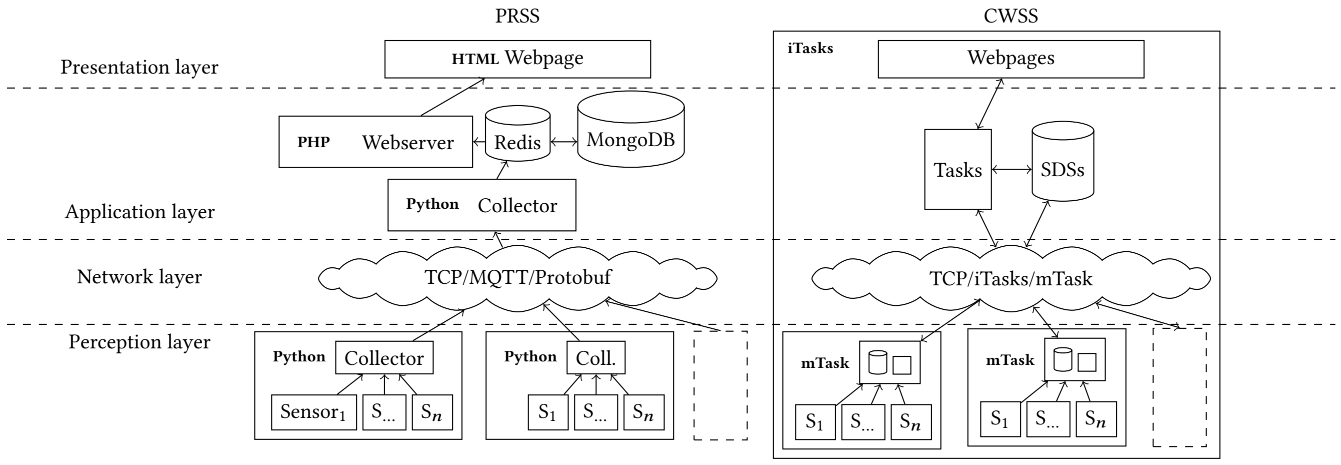


Figure 1: PRSS and CWSS mapped to a four-tier IoT architecture.

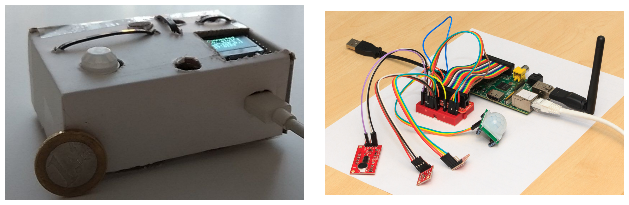


Figure 2: A prototype CWSS (left) and PRSS (right) sensor node.

and campus utilization maps [9]. A longitudinal study of sensor accuracy has also been conducted [8].

PRSS

The PRSS sensor nodes are Raspberry Pi 1 Model Bs with a range of commodity sensors connected using GPIO, I^{2}C and SPI (Figure 2). There is a simple object-oriented Python collector for configuring the sensors and reading their values.

The collector daemon service marshalls the sensor data and transmits using MQTT to the central monitoring server at a preset frequency. Each sensor node also sends a regular heartbeat signal to the measurement server. All connections are initiated by the nodes. The collector caches sensor data locally when the server is unreachable. The only well-known network name in the system is the measurement server, which must be configured for each sensor node.

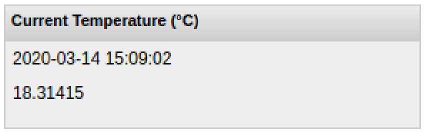
The measurement server is a commodity PC, which stores incoming sensor data in two database systems, i.e. Redis (in-memory data) and MongoDB (persistent data). The real-time sensor data is made available via a streaming websockets server, which hooks into Redis. There is also an HTTP REST API for polling current and historical sensor data, which hooks into MongoDB.

CWSS

CWSS is implemented using two DSLs hosted in the pure functional programming language Clean [3]. The DSLs adopt a novel declarative Task Oriented Programming paradigm for modeling interactive systems where tasks are the basic blocks [15].Tasks represent work and their progress is observable by other tasks and can be acted upon. Moreover, they can be combined to form compound tasks using task combinators arising from workflow modelling. Many implementation details are abstracted away from the programmer such as the user interface, the communication and the sharing of data. Tasks are implemented as event-based rewrite systems which means automatic parallelisation is possible. Alongside task values, Shared Data Sources (SDSs) are a way of communicating data between tasks. One can create an SDS for any type of data. An SDS can also provide an automatically updated view on other SDSs. The iTask system is equipped with a lean publish/subscribe system to automatically update tasks if an SDS they are watching changes.

Listing 1 shows the code for a temperature monitor written in iTask and mTask. Figure 3 shows the user interface generated by the code. The `tempSDS` stores the temperature date in a persistent SDS. The `latestTemp` provides a view to read and write the most recent value in `tempSDS`. The first line of the `mainTask` sets up the connection to the device. The mTask `devTask` is executed on the device by `liftmTask`. In parallel, the latest value of `tempSDS` is shown to the user. The mTask `devTask` continuously reads the temperature and, when it changes, writes it to `localSDS`, a lifted SDS from the server. Hence `latestTemp` and `tempSDS` are updated automatically. As a result the webpage is automatically updated by `viewSharedInformation`.

Crucially, when an SDS changes on the client or the server, any tasks watching the SDS are automatically updated. So


Current Temperature (°C)

2020-03-14 15:09:02

18.31415

Figure 3: The webpage for the Clean temperature monitor.

```
tempSDS :: SimpleSDSLens [(DateTime, Real)]
tempSDS = sharedStore "temperatures" []

latestTemp :: SimpleSDSLens (DateTime, Real)
latestTemp = mapReadWrite (hd, λx xs→Just [x:xs]) Nothing tempSDS

mainTask :: TCPSettings → Task Real
mainTask spec = withDevice spec False λdev →
        liftmTask devTask dev
    -|| Title "Current Temperature (°C)"
        @>> viewSharedInformation [ViewAs listToMaybe] tempSDS
where devTask = DHT D4 DHT11 λdht =
                liftsds λlocalSDS =
                    mapRead snd (dateTimeStampedShare latestTemp)
                In fun λtemp = (λoldtemp →
                    temperature dht
                    >>*. [IfValue ((!=.) oldtemp) (setSds localSDS)]
                    >>=. temp)
                In {main = temp (lit 0.0)}
```

Listing 1: The Clean code for the temperature monitor.

when the temperature changes, the change is automatically propagated to the server and also to the webpage.

The application layer in CWSS uses iTask, and once the tasks on the sensor nodes are running, they can interact with resources on the server and vice versa. The perception layer uses mTask, a multi-backend TOP language for programming IoT devices. The layers communicate via SDSs and task results. The nature of TOP facilitates writing the device programs at a high level of abstraction, keeping it close to the design. It offers multitasking on even the smallest of devices. Specialised IoT tasks written in the mTask language are constructed and compiled at runtime to be sent to the device. The bytecode backend is tightly integrated with iTask through a communication method agnostic interface [13]. The devices are programmed only once with a runtime system that includes a task interpreter. The runtime system is lightweight, written in portable C and supports devices as small as an Arduino UNO. Once an IoT task is integrated in iTask, it functions like a regular iTask task, i.e. the progress can be observed and data can be shared with it.

The combination of iTask and mTask allows the expression of IoT applications spanning all traditional tiers in a single source, and in a single paradigm, namely TOP. This results in less semantic friction, i.e. no mismatches in types, protocols and abstraction level between the client and server.

The UoG sensor nodes use the Wemos D1 Mini powered by the ESP8266X microcontroller that boasts integrated WiFi. It has 1 analog and 11 digital GPIO pins and there are several shields available to extend the capabilities. Figure 2 shows an assembled prototype box containing the hardware itself and the attached sensors.

Other tierless IoT systems

The iTask/mTask system is a unique IoT framework in providing a combination of a single declarative paradigm across all tiers in the stack, automatic communication and bidirectional data sharing between sensor node and server, and runtime provisioning on tiny devices. Nevertheless, there are other tierless IoT systems that share some of these capabilities.

Haskino is one of the few tierless IoT languages [6]. It is integrated in the Haskell functional language and uses runtime provisioning to control Arduino devices. Haskino supports multithreading, untethered execution, and type safety. In contrast to the functional mTask programs, Haskino executes imperative sensor node code, moreover it lacks automatic bidirectional communication between sensor nodes and server.

Functional Reactive Programming (FRP) is a declarative paradigm used for implementing IoT stacks. Within this class of languages Potato stands out as a tierless stack using hardware similar to PRSS, and leveraging the Erlang Virtual Machine [20]. TOP allows for more complex collaboration patterns than FRP [19].

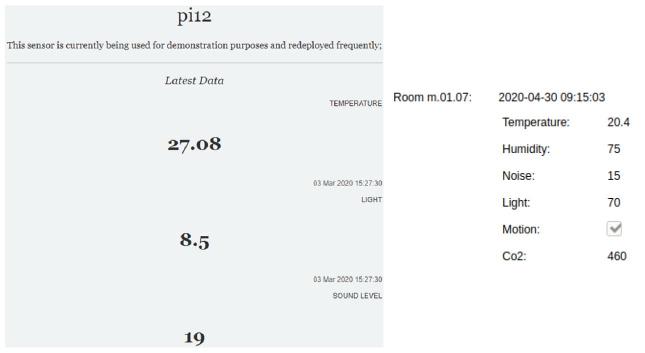
Baccelli et al. provide a single language IoT system based on the RIOT OS that allows runtime deployment of code snippets called containers [2]. Both client and server are written in Javascript, a multi-paradigm dynamically typed language. There is no integration between the client and the server other 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 runtime provisioning [11]. Alternatively, Web of Things is an initiative to standardize layered architectures to overcome impedance problems leveraging web technologies [7]. The hardware requirements are a higher than mTask and the server and clients are not as integrated.

3 OPERATIONAL COMPARISON

To ensure that the comparison reported in the following sections is based on IoT stacks with equivalent functionality we demonstrate that CWSS, like PRSS, meets the functional requirements for the UoG smart campus sensor system. We further compare CWSS and PRSS sensor node power consumption and memory footprint.

Functional validation

The main goal of the UoG smart campus project is to provide a testbed for sensor nodes and potentially other devices


pi12

This sensor is currently being used for demonstration purposes and redeployed frequently;

Latest Data

TEMPERATURE

27.08

03 Mar 2020 15:27:30

LIGHT

8.5

03 Mar 2020 15:27:30

SOUND LEVEL

19

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

Figure 4: The web interfaces of PRSS (left) and CWSS (right).

Table 1: Sensor node memory footprint and executable segment size comparison. Sizes are in KiB.

	Memory residency		Executable size	
Segment	PRSS	CWSS	PRSS	CWSS
Text	180.36	42.60	36.15	1.64
Data	0	0.90	3.39	1.24
BSS	3.82	49.47	83.34	42.36
Total	184.18	92.04	122.89	44.25

to act as a data collection and computation platform for the smart campus at the UoG. These devices 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 distributed nodes like updating software or detecting addition of new devices.

Both PRSS and CWSS meet all of these requirements. Figure 4 illustrate the web display of the information collected by each system.

Memory and power consumption

Sensor nodes have limited memory, and it is important to minimise their power consumption. As a tierless language generates the code to be executed on the sensor node, this code may not be as power and memory efficient as hand written code for the sensor nodes.

Table 1 compares the memory footprints, or residencies, of the CWSS and PRSS sensor node code. Memory is split into the standard Text (for code), Data, and BSS (Basic Service Set: statically-allocated variables). It shows that the total footprint of the runtime system of mTask is approximately half that of the corresponding handwritten Python code.

Often the memory footprint of an application is primarily determined by the number and size of shared libraries/classes loaded [1]. The PRSS Python code base loads many third party libraries like MQTT, and only uses a small part of the functionality provided. In contrast, the perception layer CWSS code is specifically generated to perform the required functionality, and hence is far smaller.

Moreover, Table 1 also compares the sizes of the CWSS and PRSS perception layer executables. While PRSS's Python is normally interpreted, for comparison purposes an executable is produced using the Nuitka compiler. Here, Python code is converted to binary and the resulting executable is linked against the Python runtime. The results show that the runtime system of mTask compiler is approximately one third of the size of the executable for the corresponding handwritten Python code.

The PRSS Python executable is larger primarily because it represents more lines of code than the CWSS executable. The contrast in executable size is even starker knowing that the mTask runtime system contains all drivers, for example the WiFi, as well while the Python executable still needs a host Operating System (OS).

The Wemos sensor node of the CWSS has the low power consumption of a typical embedded device. With all sensors enabled, it consumes around 0.2W. The Raspberry Pi sensor node of the PRSS uses more power as it has a general purpose ARM processor and runs mainstream Linux. With all sensors enabled, it consumes 1–2W, depending on ambient load. So each CWSS sensor node consumes an order of magnitude less power than a PRSS sensor node.

4 PROGRAMMING COMPARISON

This section addresses the central questions of the paper. That is, does a tierless language improve the quality of an IoT software stack, and will it simplify the development and maintenance of the stack?

Code size

Code size is widely recognised as a measure of the development effort and maintainability of a software system [16]. Table 2 enumerates the Source Lines of Code (SLOC) required to implement the UoG smart campus functionalities in both PRSS and CWSS. SLOC only counts the lines of actual code, omitting comments and blank lines.

Requiring only 172 SLOC, CWSS requires 394 fewer lines, or 70% less code than PRSS (566 SLOC). We attribute this to three main factors: (1) being tierless CWSS requires fewer languages and less interoperation; (2) CWSS automates communication between the perception, application and presentation layers; (3) the iTask and mTask DSLs used in CWSS are

Table 2: UoG smart campus code size comparison.

Functionality	PRSS	CWSS
Manage Device	178	45
Device Output	18	5
Web Interface	52	9
Database Interface	102	73
User Authentication	25	15
Hardware Interface	49	21
TCP Communication	56	2
Server Communication	86	2
Total	566	172

declarative, unlike the imperative PRSS languages (Table 3), and hence provide concise high level abstractions. We analyse each of these aspects in the following sections.

Interoperation

A major reason that CWSS is simpler and shorter than the PRSS implementation is that it uses fewer programming languages and paradigms. The multiple tiers in the PRSS stack use five very different languages and programming paradigms. In contrast, CWSS uses a single paradigm and just two conceptually-similar DSLs embedded in the same host language (Table 3).

In comparison with the single CWSS source file, PRSS comprises some 21 source and configuration files. Interoperating multiple components in multiple languages and paradigms introduces semantic friction, i.e. mismatches in types and paradigms between the components.

An example of semantic friction is the loss of *type safety*. This is where two components, possibly implemented in different programming languages, attribute different types to a data value. Such type errors can lead to runtime errors or the application silently reporting erroneous data, and can be hard to debug.

```
message SensorData {
    enum SensorType { TEMPERATURE = 1; . . . }
    SensorType sensor_type   = 1;
    uint64     timestamp     = 2;
    double     float_value   = 3;
}
.................................................................
channel = 'sensor_status.%s.%s' % (hostname,
    sensor_types.sensor_type_name(s.sensor_type))
        self.r.publish(channel, s.SerializeToString())
```

Listing 2: PRSS sends a `double` and stores a `string`.

The Clean compiler guarantees type safety as the entire CWSS software stack is generated from a single source. Type errors are identified and reported at compile time. In contrast PRSS loses type safety. For example Listing 2 first shows a `double` sensor value sent from the sensor node followed by the data being stored in Redis as a `string` on the server.

Automated communication

The PRSS developer must write and maintain MQTT [12] communication code between the perception and the application layer. Listing 3 shows the sensor node code to upload the sensor readings to the Redis store on the server.

Requiring the developer to write communication code is not the only challenge. Communication in distributed systems is intricate as sender and receiver must be correctly configured, correctly follow the communication protocol through all execution states, and deal with potential failures. For example in Listing 3, the line `redis host = config.get('Redis', 'Host')` will fail if the host or IP is not correct; likewise the following line fails if the port is incorrectly assigned.

```
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 %s" % message
            . . .
```

Listing 3: MQTT communication fragment in PRSS.

In contrast, the tierless CWSS communication is not only automated, but also automatically correct because matching sender and receiver code is generated by the compiler. The temperature application (Listing 1) uses only the following three communication functions. The `withDevice` function integrates a device with the server, allowing tasks to be sent to it. The `liftmTask` integrates an mTask task in the iTask runtime by compiling it and sending it for interpretation to the device. The `liftsds` integrates SDSs from iTask in mTask, allowing mTask tasks to interact with data from the iTask server. The exchange of data, user interface generation, and communication is all automatically generated.

High level IoT programming

For comprehensibility, the simple temperature sensor illustrated in Listing 1 is used to compare the expressive power of Clean and Python-based IoT programming abstractions. Table 4 compares the SLOC required for the two implementations: Clean Wemos Temperature Sensor (CWTS) and Python Raspberry Pi Temperature Sensor (PRTS)[3]. Recall that the CWTS Clean program is explained in Section 2.

In comparison with the single CWTS source file, PRTS comprises some 21 files that use Python, HTML, PHP, JSON,

[3]Available at https://bitbucket.org/latent12/temp_code.

Table 3: Implementation languages and paradigm comparison.

	Languages		Paradigms	
Functionality	PRSS	CWSS	PRSS	CWSS
Manage Device	Python	iTask	imperative	declarative
Device Output	HTML, PHP	iTask	declarative, imperative	declarative
Web Interface	HTML, PHP	iTask	declarative, imperative	declarative
Database Interface	Python, JSON, Redis	iTask	imperative, declarative, key/value store	declarative
Hardware Interface	Python	mTask	imperative	declarative
TCP Communication	Python	both	imperative	declarative
Server Communication	Python	both	imperative	declarative
Total	5	2	3	1

Table 4: Temperature sensor code size comparison.

Functionality	Python	Clean	Listing 1 lines
Device Output	3	-	for free by 11
Web Interface	17	2	10, 11
Database Int.	87	5	1, 2, 4, 5, 14
Hardware Int.	31	7	7, 12, 15–19
TCP Comm.	56	1	8
Server Comm.	86	2	9, 13
Total	280	17	

and Redis queries. Implementing such a small application as a conventional IoT stack requires a significant amount of configuration and other machinery that can be reused in a larger application. Hence the ratio between total PRTS and CWTS code sizes (280:17) is far greater than for realistic applications like PRSS and CWSS (566:172).

There are several ways that high-level abstractions make the CWTS and CWSS implementations much shorter than the PRTS and PRSS implementations. Firstly, functional programming languages are in general more concise than most other programming languages because their powerful abstractions require less code to describe a computation [5]. Secondly, the TOP paradigm used in iTask and mTask reduces the code size further by making it easy to specify IoT functionality concisely. For instance, the step combinator `>>*.` is an implicit repeat until one of the steps is enabled and the `viewSharedInformation` part of the UI will be automatically updated when the value of the SDS shared data store 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 [10].

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

Listing 4: An mTask failover combinator.

There is, however, no free lunch and CWSS developers must understand and effectively exploit the powerful abstractions provided by Clean and iTask/mTask. Moreover the (tierless) iTask/mTask programmer is bound to the predefined semantics of the constructs provided. For example the generated communication protocols are predetermined. Implementing unsupported behaviour requires workarounds or changes to the iTask/mTask libraries.

Failure management

Some IoT applications, e.g. room monitoring, require high uptimes for their sensors. 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 would be alerted to replace the failed device.

In many IoT architectures, including PRSS, 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 CWSS sensor node fails in CWSS, the iTask/mTask combinator to interact with a sensor node will throw an iTask exception. This exception propagates up to be caught higher up to perform act upon it, e.g. rescheduling the task on a different device in the room or to order a manager to replace the device. The declarative nature of iTask allows actions to be taken upon such events to be described succinctly.

In the UoG smart campus application, this is done by creating a pool of sensor nodes for each room and when a sensor node fails, assign another one to the task. Listing 4 shows an example of such a failover combinator for executing an mTask task on a pool of sensor nodes. If a sensor node unexpectedly disconnects, the next sensor node is tried until there are no sensor nodes left. When other errors occur, the error is propagated as usual.

Currently PRSS uses heartbeats to confirm that the sensor nodes are operational, and reports failures. At the cost of complicating the codebase, failover to an alternate sensor node could be provided.

5 CONCLUSION

We have reported the first ever systematic comparison of tiered and tierless IoT software architectures based on two implementations of a non-trivial smart campus application. PRSS has a conventional tiered Python-based architecture. CWSS has a novel tierless architecture based on the iTask and mTask DSLs hosted in Clean.

An operational comparison of CWSS and PRSS demonstrates that they have equivalent functionality, and that both meet the UoG smart campus functional requirements. We investigate the power and memory efficiency of the runtime system for the sensor nodes in CWSS (Table 1). For example, showing that CWSS's perception layer memory footprint is approximately half that of the PRSS handwritten Python, primarily due to using fewer and smaller libraries.

We report a systematic programming comparison of the CWSS and PRSS software architectures (Section 4). The tierless CWSS stack requires far less code, i.e. 70% fewer source lines, than the tiered PRSS stack (Table 2). We analyse the impact of the following three main factors. (1) Tierless developers need to manage less interoperation: CWSS uses two DSLs in a single paradigm where PRSS uses five languages and three paradigms (Table 3). Thus a tierless stack minimises semantic friction, and preserves type safety. (2) Tierless developers benefit from automatically generated, and hence correct, communication (Listings 1 and 3). (3) Tierless developers can exploit powerful high-level declarative and task-oriented IoT programming abstractions (Table 4).

The primary conclusions are (1) that *the far smaller and conceptually simpler codebase should dramatically reduce development time and improve the maintainability of tierless IoT stacks* like CWSS. (2) That *a tierless approach significantly improves the software quality of IoT stacks*, for example by preserving type safety and by automatically generating correct communication between tiers.

While promising, tierless IoT technologies also raise some challenges. Programmers must master the novel tierless iTask/mTask formalisms, and the semantics of these automatic multi-tier behaviours are necessarily relatively complex. Specifying a behaviour that is not already provided by the iTask/mTask DSLs requires either a workaround, or extending a DSL, although replicating PRSS required no such adaption. Finally, the tierless technology is very new, and the tool support and community have yet to mature.

Ongoing work explores (1) applying a tierless approach to additional tiers. For example replicating some of the business layer applications that PRSS currently supports (Section 2); (2) Using MicroPython on the Wemos microcontrollers to enable a comparison of tiered and tierless stacks on identical sensor node hardware. Other potential avenues include investigating the capabilities of tierless IoT technologies to manage additional failure modes, dynamic sets of deployed sensor nodes, and changes in sensor node configuration.

ACKNOWLEDGMENTS

Thanks to Kristian Hentschel and Dejice Jacob who developed and maintain PRSS and to funders: Royal Netherlands Navy and the Radboud-Glasgow Collaboration Fund. Furthermore we would like to thank the anonymous reviewers for their invaluable comments.

REFERENCES

[1] Ali-Reza Adl-Tabatabai, Brian T Lewis, Vijay Menon, Brian R Murphy, Bratin Saha, and Tatiana Shpeisman. 2006. Compiler and runtime support for efficient software transactional memory. *ACM SIGPLAN Notices* 41, 6 (2006), 26–37.

[2] 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, 1–6.

[3] Tom Brus, Marko van Eekelen, Maarten Van Leer, and Marinus Plasmeijer. 1987. Clean – a language for functional graph rewriting. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 364–384.

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web programming without tiers. In *International Symposium on Formal Methods for Components and Objects*. Springer, 266–296.

[5] Joe Davidson and Greg Michaelson. 2018. Expressiveness, meanings and machines. *Computability* 7, 4 (2018), 367–394.

[6] Mark Grebe and Andy Gill. 2019. Threading the Arduino with Haskell. In *Trends in Functional Programming*, David Van Horn and John Hughes (Eds.). Springer, Cham, 135–154.

[7] 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.

[8] Natascha Harth, Christos Anagnostopoulos, and Dimitrios Pezaros. 2018. Predictive intelligence to the edge: impact on edge analytics. *Evolving Systems* 9, 2 (2018), 95–118.

[9] 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, 58–62.

[10] John Hughes. 1989. Why functional programming matters. *The computer journal* 32, 2 (1989), 98–107.

[11] Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. *ACM Sigplan Notices* 37, 10 (2002), 85–95.

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

[13] Mart Lubbers. in-press. Writing Internet of Things applications with Task Oriented Programming. In *Central European Functional Programming School*. Springer, 51.

[14] Henry Muccini and Mahyar Tourchi Moghaddam. 2018. Iot architectural styles. In *European Conference on Software Architecture*. 68–85.

[15] 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*. ACM, 195–206.

[16] Jarrett Rosenberg. 1997. Some misconceptions about lines of code. In *Proceedings fourth international software metrics symposium*. IEEE, 137–142.

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

[18] Pallavi Sethi and Smruti R Sarangi. 2017. Internet of things: architectures, protocols, and applications. *Journal of Electrical and Computer Engineering* 2017 (2017).

[19] 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.

[20] 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)*. 185–192.