Building IoT Systems Using Distributed First-Class Reactive Programming

Christophe De Troyer
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
cdetroye@vub.ac.be

Jens Nicolay
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
jnicolay@vub.ac.be

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
wdmeuter@vub.ac.be

***Abstract*—Contemporary IoT systems are challenging to develop, deploy, and maintain. This is because of their ever-increasing scale, dynamic network topologies, heterogeneity and resource constraints of the involved devices, and failures that may occur as a result of these characteristics. Existing approaches are either not at the right level of abstraction, require developers to learn specialized languages, or miss certain key features to address all these challenges in a uniform manner. In this paper we leverage reactive programming and code mobility to support the entire life-cycle of large-scale IoT systems. Our approach is based on existing programming technologies and offers simple and composable abstractions to developers. We implemented our approach in a middleware called Potato and used it to develop and deploy an IoT application on a Raspberry Pi cluster. We found that using Potato reduces much of the accidental complexity associated with developing and deploying IoT systems, resulting in clean and maintainable programs.**

***Index Terms*—iot, code mobility, observables, reactive programming**

I. Introduction

IoT systems tightly integrate physical and computational processes, and cover a wide range of applications including building automation, personalized healthcare, intelligent transport, sustainable environment, and disaster recovery [12]. Because of their characteristics, and despite technological software and hardware advances, large-scale IoT applications are still challenging to develop, monitor, and maintain.

- IoT systems have grown to a massive *scale* [10]. For example, every room in a smart office building now is equipped with temperature sensors, air conditioning units, motion sensors, electronic access control, etc. At such a massive scale a node-centric approach [16], in which nodes are programmed and addressed in a largely individual fashion at a low level of abstraction, becomes unwieldy.
- IoT systems usually involve a *dynamic network topology*, in which nodes may join or leave the network at any moment, for example when they represent mobile devices such as smartphones. Therefore, developers need elegant abstractions to deal with frequent connections and disconnections.
- Compared to mainstream computing platforms, the *resource constraints* of devices in an IoT system are much more stringent. IoT devices often operate for long periods on a limited or less reliable power supply, causing them to (temporarily) disconnect from the network.
- Most IoT systems consist of *heterogeneous device types*, including any combination of micro-controllers, embedded devices, smartphones, and full-fledged servers. Developers should, however, not be concerned with every technical detail of how devices operate and communicate, but instead should be able to abstract over these devices and primarily focus on their control and data handling.
- The combination of all previous characteristics results in a complex and dynamic environment in which *failures* are not merely possible, but rather a given. This means developers need a more elegant way than a `try/catch` construct, for example, to handle every potential failure, especially when it is transient in nature.

A. Problem

State-of-the-art programming approaches for IoT applications already attempts to address most or all of the challenges outlined above, but not always in a satisfactory way. We identified three shortcomings in existing related work.

1) *Wrong level of abstraction.* Model-driven development approaches (e.g., [6, 5, 8, 13]) and declarative query languages (e.g., [14, 18]) offer high levels of abstraction, but introduce much overhead and are less flexible and extensible. Implementing new application logic in these systems is not directly modeled or requires extensive modifications to the infrastructure (e.g., parsers, query engines, ...) [11]. On the other hand, using a (low-level) general purpose language may offer too little abstraction, requiring developers to write much boilerplate code (e.g., [7]).
2) *Use of single-purpose, specialized languages.* Model-driven and macroprogramming approaches require developers to learn one or more single-purpose, specialized languages to develop IoT applications (e.g., [15, 6]). Developer tools, libraries, and documentation for these languages may be lacking and represent a considerable development effort to create and maintain. Requiring developers to learn and understand a new language may also hamper adoption.

3) *Lack of features to support the entire life-cycle in a uniform way.* Libraries and frameworks that are suitable to express application logic may miss certain features such as code mobility to support the entire life-cycle of an IoT system (including deployment and maintenance) in a uniform manner. Some approaches that offer a network-centric view of an IoT application (e.g., [11, 15]) still rely on compilation from network-centric to node-centric code, which reduces flexibility.

B. Overview of our approach

This paper presents an approach to facilitate the development, deployment, and maintenance of large-scale IoT applications based on distributed reactive programming. In this approach, IoT systems are considered to be a dynamic set of observable event streams. All important abstractions, such as the network, nodes, and channels over which they communicate, are observables. IoT applications can then be expressed as a collection of mostly functional programs that transform and compose data streams produced by these observables. Furthermore, both programs and observables are first-class and mobile, meaning that they can be stored, copied, and transmitted between nodes. We implemented this approach in a middleware called Potato.

Like the existing approaches, Potato helps developers in overcoming the challenges of developing and deploying large-scale IoT applications. More in particular, Potato has the following set of features to overcome the shortcomings we identified in existing work.

- *Simple and composable abstractions.* Potato offers simple and composable abstractions to developers. On the one hand, developers can use rich, high-level abstractions provided by Potato to reduce boilerplate code. On the other hand, developers can also define and compose their own abstractions.
- *Use of existing and established programming technology.* Potato is built using existing and well-established programming technology, so that developers are not required to learn single-purpose, specialized languages.
- *Support for the entire life-cycle in a uniform manner.* Potato contains the right feature set to support the entire life-cycle of large-scale IoT systems, including deployment and maintenance, making it suitable for use in a DevOps context.

The goal of Potato is to reduce much of the accidental complexity associated with the construction of IoT systems. This should result in clean and maintainable code and scripts to develop, deploy, and maintain IoT applications.

Our main conceptual contribution is the design of a distributed reactive programming approach combining event streams and code mobility for building IoT applications. Our technical contribution consists of the implementation of Potato based on existing and established programming technology, and its validation for the construction and deployment of IoT applications.

TABLE I
OVERVIEW OF THE USE OF OBSERVABLES AND SUBJECTS IN POTATO.

	Observable	**Subject**
Network	Observe network events	Broadcast events
Remote node	Observe remote node events	Deploy code
Local node	(Implicit evaluation)	Emit node events

C. Overview of this paper

We first introduce the main features of Potato (Section II), and then demonstrate how applications are typically developed and deployed in Potato (Section III). We then discuss how we evaluated Potato (Section IV) and discuss related work (Section V). To conclude the paper, we identify its current limitations and avenues for future research (Section VI).

II. OVERVIEW OF POTATO

Potato is a middleware, implemented in Elixir [2], for developing and deploying IoT applications. Elixir runs on BEAM [3], the Erlang VM. Erlang and related technologies are proven technologies in the realm of large-scale distributed programs. In addition to devices supporting a generic x86 platform, Elixir also runs on small devices such as Raspberry Pis, Lego EV3, and BeagleBone boards. Even smaller, more resource-constrained devices like micro-controllers that run C can also participate in an Elixir application by supporting only parts of Potato's functionality. Such devices could, for example, (only) support data transmission if they implement the Erlang distribution protocol [17], which is available as a library.

In the remainder of this section we present the basic building blocks of Potato.

A. Reactive programming

Reactive programming is a programming paradigm that maps well on the event-driven nature of IoT, as IoT applications can be regarded as an infinite feedback loop between event generators and event handlers. Potato implements the ReactiveX API [1] for asynchronous programming with observable streams. ReactiveX enables the composition, transformation, and consumption of asynchronous data streams and events. The API is well documented and has been implemented for popular programming languages, such as JavaScript, Java, C#, and Scala.

B. Observables and subjects

An *observable* is a value in the host programming language (Elixir, in case of Potato) that produces discrete values over a period of time. An *observer* can subscribe to an observable to react to these values by implementing three methods: `onNext` whenever the observable emits an item, `onError` when the observable has encountered an error, and `onCompleted` when it no longer produces any more values and no error has occurred. Composition of observables happens by means of operators to create a dedicated stream. A *subject* is both an observable (it emits values) and an observer (it can subscribe to other

```
network()
|> filter(fn {event, n} ->
             n.type == :thermometer
   end)
|> map(fn {event, n} ->
     case event do
       :join ->
         IO.puts "Temp present: #{n.uuid}"
       :leave ->
         IO.puts "Temp left: #{n.uuid}"
     end
   end)
```

Fig. 1. Print out all network events for temperature sensors on the network.

observables), and additionally implements a `next` method to put a value on the stream.

Observables and subjects form the foundation of any Potato application, and most of the values offered by Potato assume one or both of these roles. Table I gives an overview of how observables and subjects are used in Potato, while the remainder of this section presents the details.

C. Network

The `network` observable provides the starting point of most Potato applications. It emits an event as soon as a new node either joins or leaves the network. A network event therefore is a tuple of values `(event, node)` consisting of a remote `node` object and either a `:join` or a `:leave` event.

The `network` observable is a so-called *hot observable*, which means it starts producing values as soon it is created. However, this does not allow a program to obtain an overview of the current state of the network (i.e., all connected devices). Potato therefore exposes the network as a function `network`, which is a thin wrapper around the `network` observable that prepends all the nodes to the network event stream that are currently connected to the system, using ReactiveX's `starts_with` operator.

The program in Figure 1 prints out all the network events of thermometers in the network. The `filter` operator is used to filter out all the nodes of the type `:thermometer`, and operator `map` to print the network event type and the unique identifier of a thermometer to the console. The "pipe operator" in Elixir (`|>`) calls the function on its right with the value of the left side as first argument. For example, `1 |> add(2)` is equivalent to `add(1, 2)`.

The network can also be addressed as a subject to broadcast events to all nodes in the network, but due to space constraints we omit discussions and examples of this functionality in this paper.

D. Nodes

Every device that participates as a *node* in a Potato application must run an instance of Potato, and therefore nodes represent Potato runtimes in a network. A node is local if it represents the underlying Potato runtime in which it appears, and is obtained by calling function `myself()`. A

```
%{
        name:     "Wolf's phone",
        uuid:     "d4384f68..",
        type:     :phone,
        hardware: android,
        location: :roaming
}
```

Fig. 2. Example node descriptor for a smartphone.

```
thermometers =
  network()
  |> filter(fn {event, n} ->
       n.type == :thermometer
       && event == :join
     end)
  |> map(fn {_, thermometer} ->
     thermometer
     |> Obs.map(fn e ->
       case e do
         {:temperature, t} ->
           ... # process temperature values
       end
     end)
  end)
```

Fig. 3. Observe and process a stream of temperatures.

node is remote if it represents another Potato runtime than the underlying one, independent of where this runtime is (physically) executing. Consequently, all nodes obtained from `network` are remote nodes. Every node can assume the role of a subject, while a remote nodes can additionally assume the role of an observable.

A node is represented as an immutable struct of static properties. In fact, this set of static node properties is the only configuration a Potato runtime expects at startup. Although the static properties are user-defined and depend on the application, typically they will include properties such as a unique identifier, the node name, the device type, and the type of hardware.

Figure 2 shows an example of a node descriptor for a smartphone. Structs are an extension of maps in Elixir, and the `%{...}` syntax denotes a struct literal. The `:sym` syntax denotes atoms, whose value is their name.

1) Observing remote node events: Remote nodes on the network are observables that emit node events. Suppose that `thermometers` is an observable that emits thermometer nodes. Figure 3 illustrates how remote nodes can be used as observables to obtain their emitted values.

2) Emitting local node events: Each Potato application node has a single local subject, which is visible as an observable to the rest of the network. The local subject is used to emit events to the other nodes in the network that are explicitly subscribed to this subject. A node is free to emit any type and amount of events. Emitting events is typically done by wrapping the data in a tuple. For example, temperature sensors could emit events of the form `(:temperature, t)` on the net-

```
1  devices =   # all joining devices
2    network()
3    |> Obs.filter(f&(match?({:join, _}, &1))
4
5  devices     # print all node events
6  |> Obs.map(fn {_, n} ->
7      n |> Obs.map(&IO.puts/1)
8    end)
9
10 # deploy program that emits :helloworld
     event
11 devices
12 |> Obs.map(fn {_, n} ->
13   Subject.next(n,
14     program do
15       Subject.next(myself(), :helloworld)
16     end)
17   end)
18 end
```

Fig. 4. "Hello world" example of remote code deployment.

work, where `:temperature` is the tag and `t` is a temperature value. As another example, the program in Figure 4 emits a `:helloworld` notification (line 14) by accessing the local node as a subject.

3) Code deployment: Code deployment is the act of sending an expression or program from one Potato runtime to another. Any Elixir expression can be sent to a remote node by means of calling `next()` on its remote subject. The receiving node will evaluate the expression in its local Potato environment (if it supports this functionality).

The program in Figure 4 first subscribes an observer that prints all node events to every node that joins the network (lines 5–8). Then, a program that emits a `:helloworld` event is deployed on every joining node (lines 11–18). Although they represent the same underlying node, in this program there are two different subjects accessed. The subject on line 13 is a remote node and is used to deploy a program, while the subject on line 15 is the same but a local node from the perspective of the node on which the program was deployed, and therefore emits an event for that node.

The current implementation of Potato passes the data context by copy. This means that all the values captured in the scope of a first-class expression are copied at the moment the expression is sent to a remote destination. This approach presents no additional difficulties for captured observable values such as nodes, because they are essentially represented as a process identifier by the underlying Elixir runtime and are distributed transparently.

Because observables are first-class and distributed in Potato, nodes can set up and expose dedicated observable data streams themselves and notify observers of this fact using "regular" node events described so far. In this scenario a more clear separation of concerns is achieved by using different observables for different purposes (e.g., sensor data streams vs. other notifications). This may also avoid congestion in applications that transmit large amounts of data because less events need

```
1  raspberrypis =
2    network()
3    |> Obs.filter(&(match?({:join, _}, &1)))
4    |> Obs.filter(fn {:join, n} ->
5                    n.type == :pi2 end)
6    |> Obs.distinct(&Preds.same_node?/2)
```

Fig. 5. Designate all the Raspberry Pis in the network at most once.

to be broadcasted to the entire network.

Observables are implemented as Elixir actors, and therefore enjoy the same guarantees offered by the underlying VM. In particular, messages sent to observables are guaranteed to arrive in the order they are sent, or not delivered at all. Furthermore, if a node in the network fails, a program is bound to be notified of this failure.

III. Building Applications With Potato

In this section we explain how to build a small but complete application in Potato that computes the average temperature based on the readings of a *arbitrary* and *dynamic* number of temperature sensors in a network, each one connected to a Raspberry Pi (RPI). Similar to cloud-based applications, a Potato application is run on a single so-called "master" node. All the code for the application can therefore be developed using a holistic view of the application and its network on the master node, while other nodes are programmed by means of remote code deployment from within the application itself.

In what follows we discuss five important issues that IoT applications typically have to deal with: designation of nodes, deployment of programs, observation of data, and handling reconnects and failures.

A. Designation of nodes

The first part of our application is to designate all the RPIs using an observable, as illustrated in Figure 5. To deploy a program on all the currently connected RPIs, only the `:join` events from the network are of interest (line 3). Additionally, all devices that are not RPIs are filtered out (line 4). Finally, because a node can reconnect to the system at any point in time, thereby producing a `:join` event, these duplicate events are filtered out in order to designate each node only once (line 6).

B. Deployment of programs

The next step is deploying logic on all RPIs. Figure 6 deploys a program that continuously reads out the temperature value from the sensor and publishes this value to the network.

First, the binary file that actually reads out the value from a connected temperature sensor is read into memory (lines 1–2). The remainder of the program deploys (line 6) a first-class program (lines 7–15) that first writes the binary file (line 8) and then continuously emits temperature values (lines 10–13). The latter is accomplished by applying operator `each` over the infinite observable created on line 10. The temperature is read out by calling the binary using function `read_t()` (code

```
1  name = "pi2_temp.bin"
2  {:ok, bin} = File.read(name)
3
4  designated = raspberrypis
5    |> Obs.map(fn {_, node} -> node end)
6    |> Subject.next(node,
7      program do
8        write_binary_to_file(bin, name)
9
10       Obs.range(1, :infinity)
11       |> Obs.each(fn _ ->
12           Subject.next(myself(),
13                        {:temp, read_t()})
14         end)
15     end)
16   end)
```

Fig. 6. Deploy a program on all RPIs to read and publish temperature values.

```
1  network()
2  |> snapshot()
3  |> Obs.map(&Obs.merge_n/1)
4  |> Obs.switch()
5  |> Obs.filter(&Kernel.match?({:temp, _},&1))
6  |> Obs.map(&Kernel.elem(&1, 1))
7  |> Obs.chunk(5000)
8  |> Obs.map(&Enums.average/1)
```

Fig. 7. Observing and averaging temperature values.

omitted for brevity). Every temperature value is emitted as a node event by calling `next()` on the local node subject (lines 12–13).

C. Observation of events

When the program for reading and emitting temperature values is deployed on all RPIs, the produced temperature values have to be observed and aggregated to compute the average. Figure 7 shows the code required to turn observed temperature values into a single local observable that streams the average temperature every 5 seconds.

First, `snapshot()` turns a stream of joining and leaving devices into an observable that emits *sets* of currently connected devices (line 2). Every time the network updates, a new set of currently connected nodes will be emitted.

Then, all nodes in every snapshot are merged into a single observable (line 3). Merging combines the output of multiple input observables into a single observable. Next, the function `switch` is called to transform the higher-order observable resulting from the merge into an observable that emits the values emitted by the last created merged observable (line 4). From this observable all values that are not tagged with `:temp` are filtered out and stripped of their tag (lines 5–6). Finally, the temperature values are chunked into lists every 5 seconds (line 7), and the average of the resulting sets is computed (line 8).

```
1  disconnects =
2    network()
3    |> Obs.filter(&(match?({:part, _}, &1)))
4    |> Obs.map(fn {_, n} -> n end)
5  joins =
6    network()
7    |> Obs.filter(&(match?({:join, _}, &1)))
8    |> Obs.map(fn {_, n} -> n end)
9
10 seen_once =
11   joins
12   |> Obs.scan(fn v, acc ->
13       Enum.uniq_by([v] ++ acc, &L.uuid/1)
14     end, [])
15
16 reconnects =
17   joins
18   |> Obs.withLatestFrom(seen_once)
19   |> Obs.map(fn {n, seen} ->
20       if any?(seen, &same?(n, &1)) do
21         n
22       end)
23   |> Obs.filter(&(&1 = nil))
```

Fig. 8. Observe reconnecting nodes.

D. Handling reconnects

Figure 8 shows how to construct an observable `reconnects` in Potato that emits devices that reconnect instead of connecting for the first time, illustrating that observables in Potato are powerful enough to build higher-level abstractions upon. Additionally, the created observable can be used to implement more complex failure handling techniques, as we will see later.

First, observables `disconnects` and `joins` emit joined nodes and disconnected nodes , respectively (lines 1–8). Next, observable `seen_once` emits a *set* of nodes that have connected at least once during program execution, and does so whenever this set grows (lines 10–14). The three observables created so far are combined into an observable `reconnects` that emits reconnecting nodes. Operator `withLatestFrom` combines source observable `joins` with input observable `seen_once`. The result is an observable that updates only when `joins` updates, emitting the joining node coupled to the last value emitted by `seen_once`. A function is then mapped over the resulting stream of tuples of joining nodes with the set of already seen nodes to checks whether the joining node is in the set of already seen nodes. If this is the case, `nil` is emitted; otherwise the joining node is emitted. Finally, `nil` values are filtered out of the resulting stream.

E. Handling failures

Let's assume in our example application that if a node reconnects within a fixed interval of 30 seconds after deploying, the deployment has failed. Figure 9 shows the code needed to monitor for this kind of failure. The example assumes that observable `designated`, which emits every node to which a program has been deployed and was created in Figure 6, is in scope (line 1) Observables `l` and `r` stream sets of designated

```
1  l = Obs.scan(designated, &([&1 | &2]), [])
2
3  r = Obs.scan(reconnected, &([&1 | &2]), [])
4          |> Obs.map(fn ns ->
5           Enum.uniq_by(ns, &L.uuid/1) end)
6
7  timeouts = Obs.combinelatest(l, r)
8  |> Obs.merge(Obs.from([:timeout]) |> Obs.
       delay(30000))
9  |> Obs.map(fn v ->
10         case v do
11         {deployed, reconnected} ->
12                 Enums.intersect(deployed,
                       reconnected, &P.same
                       ?/2)
13                 |> Enum.map(fn node ->
14                 handle_failure(node)
15                 end)
16         :timeout ->
17                 Observable.unsubscribe()
18         end
19 end)
```

Fig. 9. Observe deployment timeouts.

nodes and reconnected nodes, respectively. These sets are updated as soon as a new node reconnects, or as soon as a new node is designated. Observables `l` and `r` are then merged with an observable that will stream a single value, `:timeout`, after 30 seconds (line 8). This merge results in an observable that streams sets of designated nodes that are paired with lists of reconnected nodes, for 30 seconds. Intuitively, as soon as a node is emitted by `designated`, and it is also present in `reconnected`, the deployment will have failed. The failure can then be handled accordingly (line 14). Finally, if the observable emits the `:timeout` value, the monitoring for reconnects should stop, because the window has closed. We garbage collect the observable by means of `unsubscribe` (line 17).

This code example shows how we can construct elaborate failure handling approaches, by means of `:leave` and `:join` messages. Another approach, for example, could be to monitor all nodes that start emitting temperature values. As soon as a timeframe has passed in which no temperature value was received, a deployment can be considered to be a failure. The failure handling can be catered towards each use case, while generic failure handling can be hidden behind abstractions.

IV. Validation

In this section we discuss our validation of Potato by implementing a Smart Office use case commonly found in literature (e.g., [6, 15, 4]). After presenting our Smart Office use case (Section IV-A, we analyze accidental and essential complexity (Section IV-B1) of the resulting application in Potato.

A. Use Case

The use case consists of modeling a large office building that is equipped with smart technology in every room. This use case is representative because it is deployed on a large scale, consists of a network of actuators and sensors, and has at least one general-purpose computing device serving as a control unit in addition to a large number of heterogenous devices that have to communicate with each other.

TABLE II
Listing of all the hardware used in the Smart Office use case.

Type	Component	Interaction	Hardware
Sensor	Temperature (AM2302)	Stream	Raspberry Pi
Sensor	Smoke (MQ-X)	Stream	Raspberry Pi
Actuator	Heating	Command	Raspberry Pi
Actuator	Alarm	Command	Raspberry Pi
Webservice	Yahoo Weather	Req/Resp	External
Sensor	RFID (RC522)	Event	Raspberry Pi
Control	Authorative Unit	N/A	Unix Server

1) Description: We assume every room in the building is equipped with a temperature sensor, a humidity sensor, a smoke detector, an alarm, and an air conditioning unit. All these devices are connected through the LAN network of the building. There is a central general-purpose computing unit that serves as the controller for the network. The controller publishes the data of the system in JSON by means of a REST endpoint to be able to create a dashboard to display the status of the network. Once the temperature in a particular room exceeds a given threshold set by the central controller, the room's air conditioning unit will attempt to bring the temperature back within a specified range. In case smoke is detected in any of the rooms, the alarms in all rooms will sound, and the control unit is also notified of this event to send out the appropriate notifications to the outside world. Each door of each room is equipped with an RFID reader to unlock its doors. Once a door is unlocked, the central controller is notified of the presence of a given person.

We implemented and executed the Smart Office application on a cluster of 160 Raspberry Pis (RPIs), in which each node represents a single RPI running an instance of the Potato runtime. The source code for our application is publicly available[1]. Table II summarizes the hardware we used to run our experiments.

2) Data flow: The use case has a very specific data flow in which data is exchanged between specific recipients, instead of broadcasting it to the system. Furthermore, the data being sent from an observable to an observer is directly routed between these two nodes. Therefore an application between two nodes forms an autonomous system, which has no need for a network connection with the master node other than for code deployment.

Once the initial application has been executed on the master node, all the components of the network will either start working autonomously to produce data for other components, or will listen to sensors to perform actions.

Thermometers, hygrometers, and smoke detectors offer their sensory readings to the network once every second. Data is

[1] https://gitlab.soft.vub.ac.be/cdetroye/potato

only sent to the nodes that are subscribed to the observable. The RFID reader works in a similar fashion, producing a user identifier every time a person moves a badge over the reader.

HVAC systems determine which thermometer is in their physical vicinity, based on the static properties of the devices, and will then subscribe to that thermometer its readings. Once the temperature exceeds a certain threshold, the HVAC system will bring the temperature back within the allowed limits. After deployment by the master node, the HVAC system and the thermometer form an autonomous system, and only require a connection between the two of them.

For sounding the alarms a program is deployed that listens to a specific "alarm" subject on the master node. The master node listens to all the smoke detectors and thermometers in the building. The system continuously checks whether or not there is any room that exceeds a smoke and temperature threshold. If this is the case, the alarm value is put on the alarm subject.

B. Implementation and evaluation

The Smart Office application is comprised of 8 modules that each handle one specific concern of the application. We manually reviewed the code of each module to determine the total lines of code and to count the lines of code that coincide with accidental complexity and essential complexity. The results of this review are listed in Table III.

1) Complexity: We consider designation, data routing, and application logic to be essential complexity, because they are important concerns in every IoT application and are specific to the application at hand. We found a high percentage of essential complexity in each of the modules. This indicates that using observables and other abstractions provided by Potato enables expressive but succinct programs that focus on application-specific concern.

Potato programs can still be verbose in certain areas, however, due to the designation and data routing using observables. Most of the accidental complexity stems from the fact that the network is producing raw tagged data, and the programmer has to filter out the relevant parts of the data by means of queries and filters. Another source of accidental complexity is the programmer dealing with the difference between `:join` and `:leave` messages, and thus has to manually keep track of currently connected nodes, previously designated nodes, etc. However, we argue that this is a reasonable trade-off as our approach results in a system that is reactive, expressive, and scalable. Furthermore, abstractions can be easily built on top of Potato, thereby offering the opportunity to further reduce accidental complexity while maintaining its advantages. We did not yet apply these abstractions in this application.

2) Conclusion: We have validated Potato by implementing a typical IoT application, which is often used as the example application in related work. Potato enables developers to write reactive programs with a holistic view of the network, without contaminating the code base with a significant amount of accidental complexity. Furthermore, the application code for the Smart Office use case shows that Potato applications are not impacted by the size of the network, and the usability of the abstractions scales up without requiring additional effort on the part of developers.

V. Related work

Our approach is inspired by Regiment [11], a functional reactive programming language for wireless sensor networks (WSN) that focuses on spatiotemporal macroprogramming. Regiment features a network-centric approach, which facilitates dealing with larger-scale networks.The programming language itself is a limited low-level reactive language with a few primitives such as `smap`, `sfilter`, and `sfold` to process data coming from the network. In Regiment, a programmer can create subsets of the network based on its topology (e.g., a node and all nodes which are maximum n hops away from that node). In Potato, each node identifies itself to the network by means of a static *node descriptor*, and nodes can dynamically emit events and data, all of which can be used to designate nodes and create subnetworks.

Several Model-Driven Development (MDD) approaches have been proposed [6, 5, 9] to address development effort and platform heterogeneity of IoT systems. MDD can hide much of complexity from the user by means of extreme high-level concepts, resulting in a DSL geared towards non-expert programmers. However, these approaches also tend to be less flexible compared to using a general-purpose programming language such as Elixir. In Potato, application and deployment logic is contained within the first-class reactive programs, which are plain Elixir programs. The abstractions offered by Potato offer are comprised of a small set of key language constructs that do not reduce generality, and enables programmers to build more specific abstractions and DSLs on top of it.

Cloud-based applications such as IBM's Node-RED [14] feature a visual programming environment for creating data flow of their entire network. A visual programming approach improves accessibility, also for non-experts. On the other hand, developers are still required to write low-level software to integrate functionality that is not supported out of the box. Furthermore, these approaches are rigid in their network topology and limited in scalability because they require the programmer to know the participants of the network beforehand.

VI. Conclusion

We presented an approach for facilitating the construction of IoT systems that combines distributed reactive programming with first-class mobile reactive programs and event streams. We found that the use of functional reactive programs to transform and compose event streams enables a declarative style of programming that is less sensitive to the scale of the underlying application. Working with event streams facilitates dealing with transient failures and dynamic network topologies. Code mobility also helps in this regard, in addition to facilitating code deployment, DevOps scenarios, and flexible edge-computing scenarios.

We implemented our approach in Potato, a library and middleware based on existing and established programming technology, and used it to evaluate its use for the construction

TABLE III
OVERIEW OF THE ACCIDENTAL VERSUS ESSENTIAL COMPLEXITY PER MODULE.

Module	LOC	Designation	Routing	Logic	Accidental Complexity	Essential Complexity
Rest Endpoint	56	0 %	35,7 %	60,7 %	3,5 %	96 %
Alarms	17	23,5 %	5,8 %	58,8 %	11,2 %	88 %
Humidity	31	16,1 %	16,1 %	54,8 %	12,9 %	87 %
HVAC	18	33,3 %	5,5 %	44,4 %	16,6 %	83 %
RFID	45	0 %	8,8 %	88,8 %	0 %	97 %
Smoke	33	21,2 %	15,1 %	48,4 %	15,1 %	84 %
Temperature	29	20,6 %	20,6 %	48,2 %	10,3 %	89 %
Weather	11	0 %	9 %	90,9 %	0 %	90 %

and deployment of IoT applications. We conclude that Potato is able to reduce much of the accidental complexity associated with constructing and deploying IoT systems.

Future Work

In this work we elaborated mostly on the conceptual approach and the foundations of Potato, and demonstrated that distributed reactive programming with first-class mobile reactive programs and event streams is a good fit for building and deploying IoT applications. However, a real-world implementation of our approach still requires other important concerns to be addressed, the most obvious of which is security. We believe that security can be implemented in the internals of our middleware by means of a public/private key infrastructure or code signing. For example, code signed and deployed by the master node could be authenticated independently by each node. This idea can also be extended to the subscription to observables.

Potato is currently also lacking a library of built-in abstractions. For example, although we illustrated how failures can be handled by creating and orchestrating observables, this code can be tedious and verbose. We therefore propose to introduce abstractions to make failure handling and other important aspects of IoT applications more straightforward, while retaining the flexibility of Potato.

Finally, Potato is built on top of the Erlang VM, and therefore currently only runs on full OS nodes such as a Raspberry Pi. Given these technological foundations, it should be possible to integrate Android devices and others into the Potato framework by using C nodes or Java nodes without much difficulty. This would also enable us to conduct a thorough performance analysis of Potato applications that are deployed in IoT environments that also comprise constrained devices.

REFERENCES

[1] ReactiveX: Observables done right. https://reactivex.io/, 2014. [Online; accessed 19-April-2018].

[2] Elixir Programming Language . https://hexdocs.pm/elixir/Kernel.html, 2018. [Online; accessed 19-April-2018].

[3] The Erlang Runtime System . https://happi.github.io/theBeamBook/, 2018. [Online; accessed 19-April-2018].

[4] Mussab Alaa, AA Zaidan, BB Zaidan, Mohammed Talal, and MLM Kiah. A review of smart home applications based on internet of things. *Journal of Network and Computer Applications*, 97:48–65, 2017.

[5] Damien Cassou, Julien Bruneau, Charles Consel, and Emilie Balland. Toward a tool-based development methodology for pervasive computing applications. *IEEE Transactions on Software Engineering*, 38(6):1445–1463, 2012.

[6] Saurabh Chauhan, Pankesh Patel, Flávia C Delicato, and Sanjay Chaudhary. A development framework for programming cyber-physical systems. In *Proceedings of the 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems*, pages 47–53. ACM, 2016.

[7] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 49(4):41–51, July 2014.

[8] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[9] Vinay Kulkarni and Sreedhar Reddy. Separation of concerns in model-driven development. *IEEE software*, 20(5):64–69, 2003.

[10] Edward A Lee. Cyber physical systems: Design challenges. In *Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on*, pages 363–369. IEEE, 2008.

[11] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 489–498. IEEE, 2007.

[12] Calton Pu. A world of opportunities: Cps, iot, and beyond. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 229–230. ACM, 2011.

[13] Estefanía Serral, Pedro Valderas, and Vicente Pelechano. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing*, 6(2):254–280, 2010.

[14] IBM Emerging Technology Services. Node-RED. https://nodered.org/, 2018. [Online; accessed 19-April-2018].

[15] Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. Building internet of things software with eliot. *Computer Communications*, 89:141–153, 2016.

[16] Ryo Sugihara and Rajesh K Gupta. Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):8, 2008.

[17] Seved Torstendahl. Open telecom platform. *Ericsson Review(English Edition)*, 74(1):14–23, 1997.

[18] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In Kay Römer, Holger Karl, and Friedemann Mattern, editors, *Wireless Sensor Networks*, pages 5–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.