First-Class Reactive Programs for CPS

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

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

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

Abstract

Cyber-Physical Systems (CPS) are comprised of a network of devices that vary widely in complexity, ranging from simple sensors to autonomous robots. Traditionally, controlling and sensing these devices happens through API communication, in either push or pull-based fashion. We argue that the computational power of these devices is converging to the point where they can do autonomous computations. This allows application programmers to run programs locally on the sensors, thereby reducing the communication and workload of more central command and control entities.

This work introduces the Potato framework that aims to make programming CPS systems intuitively easy and fast. Potato is based on three essential mechanisms: failure handling by means of leasing, distribution by means of first-class reactive programs, and intentional retroactive designation of the network by means of capabilities and dynamic properties.

In this paper we focus on the reactive capabilities of our framework. Potato enables programmers to create and deploy first-class reactive programs on CPS devices at run time, abstracting away from the API approach. Each node in the network is equipped with a minimal actor-based middleware that can execute first-class reactive programs. We have implemented Potato as a library in Elixir and have used it to implement several small examples.

CCS Concepts • **Hardware → Emerging languages and compilers**; • **Software and its engineering → Distributed programming languages**;

Keywords reactive programming, distribution, actors

ACM Reference Format:
Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. 2017. First-Class Reactive Programs for CPS. In *Proceedings of 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3141858.3141862

REBLS'17, October 23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5515-5/17/10…$15.00
https://doi.org/10.1145/3141858.3141862

1 Introduction

CPS devices in a Cyber-Physical System (CPS), or "things" in the Internet of Things (IoT), are smart devices [5] that typically form a hierarchy. The generic pattern that emerges from CPS architectures is that there are central entities (*master devices*) that continuously react to the new data coming in from sensors (*slave devices*) in the system, and send out commands to the smaller devices.

Consider the example of a smartphone application that controls the light bulbs in a house. The smartphone application will continuously monitor its environment for nearby light bulbs. In case a light bulb is nearby, the smartphone will turn on the light bulb. Upon departure, the light bulb is turned off again. Departure in this case implies disconnection. Therefore, the light bulb must autonomously decide to turn itself off.

From an engineering standpoint, programming CPS and IoT systems is intuitively difficult. These systems do not map well on the mental models of functional or imperative programming languages. The interactive scenarios (e.g., the light bulbs example) are based on perpetual actions such as continuous scanning of the environment, classifying nearby devices according to their capabilities, and monitoring connections to these devices. Furthermore, these systems are comprised of heterogeneous devices, which each have their own API. The programmer often has to resort to implementing his own abstraction layer.

The nomadic traits of these devices means that communication happens over challenged connections; the devices disconnect from the network to return shortly thereafter. The hierarchical construction of these systems means that the master devices explicitly control the slave devices. However, the challenged networks prohibit continuous control by a master.

The scale of these devices implies that the programmer can no longer individually address these devices. Hence, there is no need for the individual semantics of a device, but rather the semantics of an entire group. E.g., "all the devices in my vicinity", "all the devices that are currently in an erroneous state", or "all the light bulbs that are currently turned on". There is a need for a declarative designation of devices.

Finally we argue that the distributed reactive paradigm is an excellent fit for CPS and IoT. Often the smaller devices in the network such as temperature sensors and actuators merely push their value to the master nodes in the network. Actuating slave nodes only act on instructions coming from the master nodes. The model of acting and reacting based on signals perfectly embodies the intuition behind CPS systems.

In this work we introduce Potato, a framework to program CPS and IoT systems. Potato is built on the premise that within a system, several slave and master devices communicate using heterogeneous API's over challenged connections. Potato provides a middleware based on first-class reactive programs that allows programmers to deploy first-class reactive programs on remote nodes such that the nodes can behave autonomously without permanent control of a master. Communication between slaves and masters happens by means of reacting to distributed signals.

Contributions The contributions of Potato are as follows.

- Potato translates each newly discovered device into a set of generic capabilities.
- A potato program allows the programmer to send reactive programs using a small programming language that integrates with the on-device available capabilities.
- The host language of Potato allows the programmer to address the network using declarative intentional retroactive addressing.
- Challenged communication between a client and server device is handled by the middleware, while the programmer remains in full control of the leasing algorithm.

The Potato framework is tailored to the use cases of CPS and IoT systems in which the user can use the Smart Objects in its surroundings as extensions of the application on its smartphone or other server device. The problems posed by these scenarios such as intermittent connections, heterogeneous API's and intuitive programming model are addressed by Potato.

Overview We give an overview of the principles and design of Potato (section 2) and discuss our implementation of its middleware in Elixir (section 3). We then present related work (section 4) and some avenues for future work (section 5).

2 Potato

We describe the architecture of the Potato middleware. Potato is a proof-of-concept middleware written in Elixir with an embedded DSL to program clients in the network dynamically by means of first-class reactive programs.

The Potato network consists of four components: network discovery, DDF management, reactive capabilities, and designation management. Below we discuss each of these components.

2.1 Distributed Reactivity

In a Potato system, the slave and master nodes are equipped with an actor middleware that supports first-class reactive programs. This results in devices being able to send and deploy reactive programs on remote nodes in the network.

Each node is aware of its local signals (e.g., time, temperature, humidity, location, etc.). These signals can be subscribed to, as is done in contemporary reactive languages, by lifting and applying functions. Regular functions can be lifted and applied to a signal to produce new signals, to which in turn new functions can be applied to generate a complex reactive DAG.

Whenever a node has registered a new signal locally, all nodes in the network are notified of this. From that point on, programs deployed on this node can lift and apply functions on this newly created signal.

Additionally, the programmer can lift functions on signals on remote nodes as if they were local signals.

This results in a distributed web of signals spread across the network. The main goal of this approach is that much of the logic (e.g., averaging signals, minimum and maximum values, etc.) no longer has to be computed on master nodes, but immediately can be computed on slave nodes. This reduces traffic on the network and reduces resource consumption on the master nodes.

Additionally, if more than one master device needs a new signal based on a remote signal, duplicate calculations on master devices is avoided. This also reduces the complexity of the program code on the host. E.g., it can be argued that signal manipulating operations , such as Celsius to kelvin conversion, are concerns for the sensor, and not for the master node.

2.2 Network Management

Each device in the network must manage its own view on the network, which is a snapshot of the network at any point in time. We now discuss four abstractions related to network management: discovery, device description, leasing, and designation.

2.2.1 Discovery

The first layer of abstraction is the network discovery abstraction. Both master and slave devices in a Potato system continuously listen on the network for new devices. If a device boots, the Potato middleware will automatically notify all its peers in the network of its presence. In doing so, all devices that are in reach are aware of each other on the network level.

At this point in the architecture the only knowledge a device has about its peers is their unique location in the network (e.g., IP address).

2.2.2 Device Description File

The second layer of abstraction is formed by Device Description Files (DDF). Each device that wants to participate in a Potato application needs to be identified to its peers by means of a DDF. A DDF contains all the relevant information for the to set up communication with the device, and also contains the set of instructions the device understands.

As soon as the discovery layer has notified the DDF layer of a new reachable device, the DDF layer will contact the DDF layer on the remote device. First the device sends its own DDF to the discovered device, and awaits the DDF of the discovered device to be sent back. The DDF service then maps the the DDF files to their corresponding remote identifier (e.g., IP).

Listing 1. Device Description for a Lamp

```
{
  "uuid": "8c2117b0-51cd-11e7-bcdb-67
     c26160338f",
  "name": "Lamp 123",
  "location": "office",
  "control": "WIFI",
  "type": {
    "name": "lamp",
    "capabilities": [
      {"switch": {"ops": ["on", "off"]}
    ],
    "signals": [temperature, humidity]
  }
}
```

Listing 1 shows an example DDF. A DDF is composed out of meta-data concerning the participating device. It contains information such as a globally unique identifier, a human-readable name, location, communication channel (e.g., Ethernet or Bluetooth), a type, a list of capabilities, and a list of signals available on the device.

As soon as a device is discovered, the middleware on each device on the network creates a proxy data structure that facilitates all the communication with the physical device. This means that the programmer can communicate with the data structure as if it were a plain old data structure. Furthermore, this gives the framework more flexibility in managing the actual connection between devices.

2.2.3 Leasing

The challenged connections in the systems we have discussed up to this point can not be easily managed. Connections can no longer be managed individually, due to intentional designation (see Section 2.2.4) to catch failures. Additionally, connections are challenged and thus fail periodically. We argue that a short disconnect should not affect the semantics of the system, nor should be of concern to the application programmer. Leasing, as for example found in AmbientTalk [8], is crucial building block of Potato. Leasing allows a programmer to gracefully handle intermittent connections. For example, if a user walks around with a smartphone to read out values from sensors in its vicinity, a temporary disconnect does not mean that the smartphone should no longer await updates of the value of that sensor. Leasing maintains a time-bound *logical* connection, even if the *communication link* is temporarily broken.

Note that the leasing algorithm must be fine-tuned based on the application in which it is deployed. Consider comparing the value signal by a temperature sensor with the control switch of an industrial robot. In the former case, no longer receiving values from the sensor could either be a temporary disconnect, or it could be that there is an issue with the temperature sensor. Either way, the impact on the workings of the application is probably minimal. The control of an industrial machine however, is very crucial. If we suddenly lose connection we want the application to immediately be able to take action accordingly.

Hence, we propose a leasing model without default lease times, such that every connection must be configured to have a specific timeout. In traditional AmbientTalk there is a default leasing timeout. However, this means that if the programmer omits one explicit lease timeout it can break the entire application. We argue that default lease timeouts are the worse of both options.

2.2.4 Designation

The final layer of abstraction on top of the network is the designation layer. The previous layer (i.e., DDF layer) managed all the currently approachable devices. The designation layer builds on top of this by managing designations of several devices.

Intentional designation is defined as grouping together parts of a network by means of a predicate without resorting to individual enumeration of devices in that set. An example of the intentional is show in Listing 2.

Listing 2. Intentional designation of all light bulbs.

```
with(e is Lightbulb and e.turned_on)
do
 // code here
end
```

Listing 2 designates all the reachable devices in the network that are of the type `Lightbulb`, and that are currently turned on. While the code snippet seems trivial, the middleware is doing a lot of background work. The execution of the following code can be broken down in the following steps.

1. List all the devices currently present in the application
2. Filter out all the devices that are not a `Lightbulb`.
3. Filter out all lamps who's signal says they are off.
4. Serialize the program on line 2 and send it to the devices.
5. Perpetually monitor the network for new participants. If a new participant joins, execute that code for the new participants.

Note that the first predicate is known as soon as the device has broadcast its DDF on the network; the type of a device is static. The second predicate requires us to have runtime information about the device. To be able to do so the DDF layer will keep the network updated about its current DDF. Most parts of the DDF remain static, such as the name, human readable name, unique id, etc. Runtime values such as "location" in case of a GPS-equipped device are periodically transmitted to the rest of the network.

3 Implementation

The implementation of Potato is a work in progress. Currently we have implemented a proof of concept in Elixir. While the eventual goal is to deploy Potato on challenged devices that require multiple protocols, we currently only use Elixir and deploy Potato on Raspberry Pis.

We argue that Elixir is a good platform to build a prototype because the language runs on the BEAM VM. BEAM is a VM geared towards actor or process based languages. The process paradigm fits perfect on the different network process we have discussed in Section 2. Each component of the Potato mentioned before is implemented using a process, and distribution is almost free in BEAM. Overall this helps to reduce the development cost of our framework.

In this section we discuss how we implemented the concepts of Section 2 in Elixir.

3.1 Reactivity

The code that is sent over to remote nodes are small scripts. We call these scripts Potato Scripts. The basis of our Potato Scripts is an Elixir library named "Reactivity". Reactivity allows the programmer to write small reactive programs using four primitives; `source`, `register`, `lift`, and `apply`. These primitives suffice to implement common reactive programs.

Each device in the network is assumed to have a set of built-in signals. For example, a temperature sensor has a "temperature" signal etc. Based on these input signals the programmer can create derived signals.

Consider the example in Listing 3.1.

```
source(:temperature)
|> liftapp(fn(t) ->
             if t > 24 then:
               :too_hot
             else
               :too_cold
             end
           end)
|> register(:hotcold)

source(:hotcold)
|> liftapp(fn(x) ->
             if x == :too_cold then
               puts "It's too hot!"
             then
               puts "It's too cold!"
             end
           end)
```

The first part of the example applies a function to the `:source` signal and create a new signal that spits either a `:too_hot` or a `:too_cold` symbol.

The second part of the program subscribes to this new signal and prints out whether it is too hot or too cold.

3.2 Signals

The DDF of any device contains a list of its *signals*. These are streams of data that update regularly. Examples are the "temperature" signal on a temperature sensor, and the "location" signal on a GPS tracker. Devices can send reactive programs to slave devices that capture these signals and act upon them.

Listing 3 depicts an example of such a reactive program. Line 1 designates all the devices in the network that are a Tracker *and* have a GPS signal. On all these devices we deploy a reactive program that transmits its value to the host device. Upon deployment on the node, the `master` keyword in this program is replaced by a remote reference to the master device that sent the program to the slave node.

Listing 3. Sending all GPS locations back to the master device.

```
with(e is Tracker and e has GPS) do
  signal(:GPS)
  |> liftapply(&(send(master, &1)))
```

Previously we hinted at creating new signals at runtime. We enable this by using the `register` keyword. Consider the example in Listing 4, in which the location signal is translated into a publicly available signal.

Listing 4. Creating a new signal on all GPS devices.

```
with(e is Tracker and e has GPS) do
  signal(:GPS)
  |> liftapply(fn(_) -> :moving end)
  |> register(:moving)
```

Any device that has a running designation which requires a device to have a signal will be triggered as soon as the updated DDF has propagated the network.

3.3 Network Discovery

Potato relies on UDP broadcasting to realize network discovery. On top of this discovery primitive we use Erlang's node monitor system. This is a built-in functionality that enables the monitoring of the liveness of devices such that we can handle the events of new node that connecting and nodes that disconnect.

The Network process thus receives `nodeup` and `nodedown` signals, meaning the can only interact at this granularity. However, Erlang enables tweaking the timeout for the node connections, which in turn enables Potato to implement its own timeouts. This means that the network discovery service can delay the notification of disconnect to the rest of the system by means of leasing.

3.4 Device Description Files

The DDFs of the devices are managed by a separate `Designator` actor. The DDF actor is notified by the Network actor in case a new device has been discovered. The DDF actor will then send a message at the Erlang level to the other IP. If the device replies with its DDF the DDF is stored in the current network and can be used for designations.

If the Network manager sends a disconnect message for that particular node, the DDF is again removed.

3.5 Designation

The designation logic is handled by the Designator actor. This actor listens to the client's application for designations. During evaluation of the clients program the Designator is sent tuples containing a predicate and a program. The Designator will deploy the program on all the remote nodes for whose DDF the predicate succeeds.

This of course means that every new node that joins the network has to be checked against all the pre-existing designations.

4 Related Work

Existing frameworks do not address the same set of problems as Potato or have a different computational model.

Node-RED [1] and DDF [3] are a distributed data-flow modeling web applications. Node-RED is aimed at extracting data from an IoT system by means of deploying the Node-RED software on all the nodes in the network, or configuring an API for each node in the network. Our system has a similar textual approach to programming the network, but has been built with failure in mind from the start. While in Node-RED failure is still not handled gracefully.

nesC [2] which runs on TinyOS [6] is a C-like programming language that supports reactive distributed networked applications. However, nesC is targeted solely low-power highly constrained devices. Potato aims to incorporate these devices into its network in the future, but its end-goal is to have a clear master-slave architecture platform that incorporates both constrained devices as well as high-power devices such as smartphones and mainframes. This makes the language design aspect of potato different from nesC. Finally, Potato is distributed reactive first, while nesC's event-driven model is aimed at wireless sensor networks.

Esperanto [4] is a framework that creates Object-Oriented-style objects that act as a gateway to the device. This approach means that generic commonalities between devices can be moved up in the OO-hierarchy. However, the architecture of Esperanto differs significantly from Potato. Esperanto is a form of tierless programming that offers a holistic view of the entire application. Potato offers the same form of holistic programming, but by means of intentional designation. This means that distribution is by no means hidden from the programmer and failures are still apparent to the programmer and can be handled more elegantly.

Tomlein et. al [7] have published a model to automatically deploy software on a network of nodes. Each node in the system has metadata which is embedded in the software that needs to be deployed. During the deployment phase of the software the system figures out, using semantic reasoning, which software should be deployed on which device. These rules are encoded using the Notation3 language. While the designation of these devices happens in a declarative way, the designation happens once, before the actual deployment. Potato designates each device that joins at an undefined time during the execution of the software. Furthermore, not only static meta-data is considered by Potato, but also the middleware signal values, which allows for a more real-time designation.

5 Future Work

5.1 Finer-grained Designation

The second open research question is how we can deal with more fine-grained designations, while also dealing with failure. For example, if we wish to designate only 5 of the nearby light bulbs and deploy a blinking program on them. The problems with this arise due to the retroactive designation. Once we have deployed a program on a remote node, there is, for now, no way to cancel that. As soon as one of those light bulbs fail, we should cancel the blinking on all 4 remaining light bulbs.

On the other hand, consider the disconnection between the master device and the failing light bulb to be temporary. The light bulb is autonomous and will likely still be blinking. Here, too, the possible solution is to use two-way leasing.

5.2 Leasing In Reactive Systems

The first open research question is how we can properly integrate leasing within the functional reactive paradigm. Again, consider the example of a master device applying a local function on a remote signal on a slave device. After a temporary disconnect the leasing paradigm might send a batch of values from the slave device to the master device. If we process all these signals we might create glitches in the system. On the other hand, if we discard the values we might miss important information. A possible solution to this problem is using two-way leases. This means that slave nodes monitor their masters and vice versa. At any point in time, modulo lease time, every device is this is new aware of failed communication links.

If the devices have noticed the communication link is offline, the reactive program can safely be unscheduled until the connection link is back. That way, neither the slave nor the master will produce new values on their signal and expect them to be consumed by the remote node.

5.3 Declarative Signal Designation

The third open research question concerns the declarative designation of signals. Consider the following application.

```
with(s is Sensor and s has Celsius) do
  signal(:Celsius)
  |> liftapply(fn(c) -> c + 237.15 end)
  |> register(:kelvin)
```

Listing 5.3 creates a new signal on tdfdfhe slave nodes which are Sensors. If the master device wants to compute the mdfdfaximum of these values, the programmer would need to subscribe to all these signals. We would like to use a declarative approach to apply a function to all the signals in the network, and compute it on the master node. A fold operadfdtion over this signal would be one possible way to approach this. Listing 5.3 exemplifies a possible implementation.

```
fold_signal(signals(:temperature),
            fn(t, acc) ->
              max(t, acc)
            end)
```

6 Conclusion

Potato is a framework that enables application prodfdfgrammers to express their intuitive ideas about interactive CPS and IoT applications in a natural way. We believe that it bends the train of thought of the programmer in the right direction.

Failures in the distributed applications are mitigated using two approaches. First, we introduce leasing to handle partial failures in the distributed reactive programming. Second, we deploy reactive programs on remote nodes as soon as there is a connection available. This allows the devices to work autonomously in the absence of a controlling master.

We have discussed a proof-of-concept implementation of Potato in Elixir.

References

[1] Michael Blackstock and Rodger Lea. 2014. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *Proceedings of the 5th International Workshop on Web of Things (WoT '14)*. ACM, New York, NY, USA, 34–39. https://doi.org/10.1145/2684432.2684439

[2] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. *SIGPLAN Not.* 38, 5 (May 2003), 1–11. https://doi.org/10.1145/780822.781133

[3] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor CM Leung. 2015. Developing iot applications in the fog: A distributed dataflow approach. In *Internet of Things (IOT), 2015 5th International Conference on the*. IEEE, 155–162.

[4] Gyeongmin Lee Seonyeong Heo Bongjun Kim and Jong Kim Hanjun Kim. [n. d.]. Integrated IoT Programming with Selective Abstraction. ([n. d.]).

[5] Gerd Kortuem, Fahim Kawsar, Vasughi Sundramoorthy, and Daniel Fitton. 2010. Smart objects as building blocks for the internet of things. *IEEE Internet Computing* 14, 1 (2010), 44–51.

[6] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. *Ambient intelligence* 35 (2005), 115–148.

[7] Matú Tomlein and Kaj Grønbæk. 2016. Semantic model of variability and capabilities of iot applications for embedded software ecosystems. In *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*. IEEE, 247–252.

[8] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. 2007. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC'07. XXVI International Conference of the*. IEEE, 3–12.