Abstractions for Distributed Event-Driven Applications

Position Paper

Christophe De Troyer,
Jens Nicolay, Wolfgang De Meuter
Vrije Universiteit Brussel
Pleinlaan 2
Etterbeek, Belgium 1040
cdetroye;jnicolay;wdemeuter@vub.ac.be

Christophe Scholliers
Universiteit Gent
Krijgslaan 281
Gent, Belgium
Christophe.Scholliers@UGent.be

ABSTRACT

The Internet of Things (IoT) requires us to rethink the way distributed event-driven applications are programmed. IoT applications differ from traditional distributed applications on a number of points. First, they are comprised of an order of magnitude more devices that operate within a dynamic network. Second, failure in large dynamic networks is no longer an exceptional state but a given and thus needs to be part of the core semantics when programming such networks. Third, the hardware in these networks is not homogeneous so that a common software stack is impossible. We believe that contemporary event-driven languages do not offer appropriate abstractions to write IoT applications. We propose a novel computational model for programming IoT applications by identifying four key abstractions for designating network nodes and handle failures that facilitate writing large-scale IoT applications.

CCS CONCEPTS

•**Computer systems organization** →**Distributed architectures;**
•**Software and its engineering** →*Runtime environments;*

KEYWORDS

Internet of Things, Distributed Programming, Runtimes

ACM Reference format:
Christophe De Troyer,
Jens Nicolay, Wolfgang De Meuter and Christophe Scholliers. 2017. Abstractions for Distributed Event-Driven Applications. In *Proceedings of Programming 2017, Brussels, Belgium, April 2017 (PROGRAMMING'17),* 2 pages.
DOI: 10.475/123_4

1 DISTRIBUTED EVENT-DRIVEN APPLICATIONS

Internet of Things (IoT) is an umbrella term for several types of distributed, event-driven applications. In this paper we restrict ourselves to networked applications that have dynamic network topologies in which the nodes of the network can range from battery-operated microcontrollers to mainframes.

PROGRAMMING'17, Brussels, Belgium

DOI: 10.475/123_4

We exemplify our focus by means of a simple application called *walking lights* that runs on a user's smartphone and communicates with its environment by means of bluetooth. When a user walks along a trail lined with tens of thousands of LED lights, the application continuously scans for bluetooth devices in the vicinity of the user's smartphone. Nearby LED lights turn on, while more distant LEDs turn off. Every LED is controlled by a battery-operated microcontroller equipped with GPS and bluetooth. This enables each LED to be aware of its position and broadcast its location and state of the LED light (on/off).

Besides *walking lights*, another example of an even-driven distributed application is a smart store in which shopping carts, smartphones, shelves, boxes, etc. are all equipped with microcontrollers. Based on these and other example applications we discern three key properties of distributed event-driven applications.

Dynamic Network Some nodes in the network are static (e.g., the LEDs microcontroller), while other nodes are nomadic (e.g., the user's smartphone).
Failures In distributed event-driven programs at the scale of *walking lights* or smart stores, failures can be caused by for example dead batteries or interference on the radio communication. Failure in these kinds of application therefore is a given instead of an exceptional state, and the language runtime should be able to deal with these failures adequately.
Heterogenous Network In IoT networks devices are either simple and energy-constrained such as sensors, or they are comparatively powerful such as smartphones and mainframes. To facilitate homogeneous communication between these devices, all devices should communicate using a common interface.

Programming distributed event-driven applications therefore requires the right abstractions to adequately deal with these properties.

2 RELATED WORK

Wireless sensor networks can be considered as a precursor to IoT.Declarative SQL-like languages such as Cougar[1], TinyDB[3], and SINA[4] offer a form of query language to gather data from the sensor network. While the declarative approach is similar to the idea we propose, the communication is unidirectional in the sense that data always flows from the sensors to the sinks, while the applications we envision require bidirectional communication.

AmbientTalk[2] is an event-driven programming model based on a distributed actor language that allows the programmer to hook into the network and discover nearby devices that run the same

application. For the applications we envision, AmbientTalk lacks three features. First, it can only communicate with applications running the same software stack. Second, retroactively applying logic over a designated network is impossible. Third, failures are hidden from the programmer, which means there is no explicit behavior possible upon communication failure.

Contemporary runtimes such as the JVM and Erlang BEAM offer no out-of-the-box possibilities to communicate with devices through other protocols than Ethernet. To develop the applications like *walking lights* it is required that the runtime supports multiple communication protocols such as radio, bluetooth, WiFi, and Ethernet. To the best of our knowledge no runtime exists that allows the programmer to abstract away from the communication protocols and program with a homogeneous first-class network.

3 ABSTRACTIONS FOR DISTRIBUTED EVENT-DRIVEN APPLICATIONS

In this paper we argue for novel abstraction mechanisms to reduce boilerplate code and programmer effort to implement large-scale distributed event-driven applications. We identify four properties of distributed event-driven applications, and for each property we propose a novel language abstraction.

3.1 Intensional designation

Most programming languages designate endpoints in a network in an extensional way by creating a list of endpoints and then iterating over them to execute an expression. Instead we propose *intensional designation* of network nodes, in which the programmer creates a subnetwork based on a set of constraints such as "in vicinity". Constraints allow the programmer to create subnetworks and address them as a whole, or execute logic for each individual member. Consider the example program below.

```
def nearbyLEDs =
{e in Led | e.inRange(myLocation(),20)}
```

In this example all the elements in the current network that expose the interface `Led` are selected. This interface publishes the runtime property `location`, which is used to select only nearby elements.

3.2 Retroactive Designation

In many event-driven applications all events of interest, e.g., a device joining or leaving the network, must be explicitly handled in order to keep the network in a consistent state with reality. However, in a massively distributed application this becomes tedious and error-prone. We therefore introduce the notion of *retroactive designation*. Consider the example below from *walking lights* that turns on all LED lights in the vicinity of the user's smartphone.

```
with led:{e in Led | e.inRange(myLocation(),20)}:
  led.on()
```

Whenever a LED light enters the vicinity of the user's smartphone, the runtime will automatically re-evaluate the body of the designation so that the LED's light is turned on.

3.3 Failure Handling

The IoT is often comprised of battery-powered devices that communicate over several different channels such as radio, bluetooth, and ethernet. This introduces several points of failure, and disconnects and communication failures are no longer an exceptional state in the program but a given. We therefore propose to incorporate a *programmer-managed failure threshold* to allow the program to tolerate a certain percentage of failures from devices. Additionally, we propose to introduce *optional failure functionality* to be specified by the programmer. The example below shows how to address all the LEDs in the vicinity of the user's smartphone using a failure threshold of 90%.

```
with led:{e in Led | e.inRange(myLocation(),20)} require
     90\%:
  led.on()
onfail do:
  display "led failed: " led
```

In case less than 90% of the devices acknowledge the request to turn on their LED, the expression is considered a failure and traditional exception handling can be applied. If at least 90% of the devices acknowledges, the expression is considered a success. All the devices that have not replied will be handled by the expression following the `onfail` keyword.

3.4 Compensating Actions

Once an expression has been executed for a designated device, we optionally want to compensate that action when the device leaves our designation. We propose to use *compensation actions* to achieve this. In *walking lights*, for example, we want to turn off the LED light when the microcontroller is no longer in the vicinity.

```
with led:{e in Led | e.inRange(myLocation(),20)} require
     90\%:
  led.on()
onfail do:
  display "led failed: " led
compensation:
  led.off()
```

The compensating action is specified using the `compensation` keyword.

4 CONCLUSION

The Internet of Things requires us to rethink the way distributed systems are programmed. We argued that contemporary event-driven languages are not at the proper level of abstraction to write IoT applications. In this paper we proposed four novel language abstractions for designating network nodes and handle failures that facilitate writing large-scale IoT applications.

REFERENCES

[1] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. 2000. Querying the physical world. *IEEE personal Communications* 7, 5 (2000), 10–15.
[2] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo DfiHondt, and Wolfgang De Meuter. 2006. Ambient-oriented programming in ambienttalk. In *European Conference on Object-Oriented Programming*. Springer, 230–254.
[3] Sam Madden, Joe Hellerstein, and Wei Hong. 2003. TinyDB: In-network query processing in tinyos. (2003).
[4] Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen. 2000. Sensor information networking architecture. In *Parallel Processing, 2000. Proceedings. 2000 International Workshops on*. IEEE, 23–30.