Enabling reactive cities with the iFLUX middleware

Olivier Liechti
HES-SO University of Applied Sciences Western Switzerland
CH - 1401 Yverdon-les-Bains
olivier.liechti@heig-vd.ch

Laurent Prévost
HES-SO UAS-WS
CH - 1401 Yverdon-les-Bains
laurent.prevost@heig-vd.ch

Valentin Delaye
Novaccess SA
CH - 1401 Yverdon-les-Bains
valentin.delaye@novaccess.ch

Jean Hennebert
HES-SO UAS-WS
CH - 1705 Fribourg
jean.hennebert@hefr.ch

Vincent Grivel
HES-SO UAS-WS
CH - 1705 Fribourg
vincent.grivel@hefr.ch

Jean-Philippe Rey
HES-SO UAS-WS
1401 Yverdon-les-Bains
jean-philippe.rey@heig-vd.ch

Jonathan Depraz
Novaccess SA
CH - 1401 Yverdon-les-Bains
jonathan.depraz@novaccess.ch

Marc Sommer
Novaccess SA
CH - 1401 Yverdon-les-Bains
marc.sommer@novaccess.ch

ABSTRACT

This paper presents the iFLUX middleware, designed to provide a lightweight integration solution for Smart City applications. Based on three core abstractions, namely *event sources*, *action targets* and *rules*, iFLUX makes it very easy to expose sensors and actuators through REST APIs so that they can be integrated in application-level workflows. Sensors and actuators can be smart objects integrating hardware and software, but can also be pure software services. In the paper, we introduce the iFLUX programming model and describe how it has been implemented in a middleware platform. We also report on how the platform has been used and evaluated in various contexts. While iFLUX has been initially designed in the context of Smart City applications, it is generic and applicable to other domains where hardware and software components are connected through the Web.

Keywords

smart city, integration middleware, rule engine

1. INTRODUCTION

While there is no single definition for the term *Smart City* [3, 2, 9], the general idea is that ICT technologies can contribute to improve the quality of life by impacting a wide range of domains: energy, transport, safety, administration, politics, culture, etc. Because of this broad definition, very different types of applications and technologies fit in the scope of smart cities. Think of a sensor network that measures air quality. Think of an application that optimizes public transportation usage by giving incentives to travel outside peak hours. Think of services that encourage citizen to interact more actively and directly with authorities. These are only a few examples that illustrate the variety of smart city applications. Some of these applications have a strong physical dimension, when sensors and actuators are material artifacts (smart objects). Other applications have a lesser physical dimension, when the sensors and the actuators are actually software systems (e.g. mobile applications, online business services). In this paper, we consider the entire spectrum between these two cases.

Cities do not become *smart* overnight. Rather, they become smarter through an evolutionary process. New technologies and applications are introduced over a long period of time, often without an overall architecture defined a priori. Every domain of the city is managed as a silo, with its own needs, its own services and its own infrastructure. For this reason, it is often difficult to build cross-domain applications. Think of an application that would seek to optimize energy consumption by continuously adapting street lighting to the current road traffic. While the sensors (on the roads) and the actuators (in the street lights) might actually be deployed, building the application is generally a challenge because the two components live in two silos, isolated because of a mix of organizational, administrative and technical reasons.

This situation is analogous to any significant information system. Hence, it raises the usual questions: what is the best way to integrate heterogenous components deployed across a Smart City? How can we enable developers to create and deploy new services independently, while ensuring that these services can be shared, reused and combined in higher-level workflows and applications? How can we make the integration of legacy services into the new ecosystem as effortless as possible? Over the last decade, the REST architectural style and standard web technologies (HTTP, JSON, etc.) have proven to be very effective in this pursuit. Combined with the exponential growth of the Internet of Things both in consumer and industrial settings, this makes the Web of Things [4] an ideal paradigm for designing city-wide services, where there is a mix of hardware and software components.

Adopting the Web of Things as an architectural style for smart city applications, however, does not address all architectural questions. What concrete guidelines should be followed to expose components through REST APIs? What interaction patterns should be used between system components? How can we concretely ease the creation of new urban services? How do we ensure that the interaction with these services in higher-level applications is easy and manageable? It is to investigate such issues that the iFLUX project has been initiated. Our first objective was to propose a programming model, based on WoT principles, that can be applied to build citywide applications. Our second objective was to expose this programming model in a concrete middleware platform and to evaluate it in a series of illustrative applications.

In the remaining sections of this article, we first give more information about the context in which iFLUX has been created and describe the main design objectives associated with this context. We then introduce the iFLUX programming model by describing three core concepts: *event sources*, *action targets* and *rules*. We then present one aspect of the middleware implementation, by describing three main RESTful endpoints. We finally explain how we have evaluated the platform in several applications, some of which have been built by third-party developers.

2. CONTEXT AND DESIGN OBJECTIVES

The need for the iFLUX middleware has emerged in a research program dedicated to smart cities, entitled iNUIT and established at the HES-SO University of Applied Sciences Western Switzerland. The goal of iNUIT is to build a complete IoT stack for smart city applications. It is based on a layered architecture: the lower layer deal with the interconnection of physical objects (specialized sensors, low-power mesh networks, WoT gateways, etc.), the intermediate layer deal with data analysis (sensor fusion, video processing, etc.) and the upper layer hosts applications (e.g. crowd monitoring). Initially, middleware services have been provided in the iNUIT cloud for archiving and querying information (both raw sensor data and extracted information). One goal of iFLUX was to add the capability to process event streams and to enable a reactive programming style [12].

2.1 Reuse

Our first objective was to encourage the reuse of iNUIT services developed in different projects. Typically, the teams working on individual components (e.g. a new type of sensor, a video processing module) focus on a specific service. They often implement basic demonstrators to validate this service. What is often not so easy to do for them is to expose the service, so that it can be found and used by other teams (and in particular by those working at the application level). Hence, our first goal was to provide an easy solution for component developers to bring their technology in a single service catalog. From the application developers point of view, the interaction with all services in the catalog should follow the same approach and patterns. This is what is captured in the iFLUX programming model.

2.2 Decoupling

The fact that many independant teams are involved in the iNUIT program reflects what happens in most smart city environments, from an organizational point of view. In the introduction, we used the term *silo*, to suggest that it must be possible for different teams to develop atomic services and applications with as few dependencies as possible. With this in mind, iFLUX is based on a micro-services oriented architecture. Every component of the ecosystem is developed and operated independently. Web protocols provide the platform-independant glue between the components. We have used the Docker virtualization technology, based on lightweight containers, to facilitate the packaging and deployment of these independent components.

2.3 Simplicity

We wanted iFLUX to be very lightweight and easy to use, both for service and application developers. The programming model should be easy to grasp, hence with a small number of abstractions. Bringing a sensor or an actuator into the iFLUX ecosystem should not require a big effort. The APIs that have to be consumed or published should be concise. This is one reason for which we have decided to initially only support stateless rules. We will later show that state can easily be managed in *action targets*.

3. PROGRAMMING MODEL

The programming model for iFLUX was inspired from popular lightweight service integration services, such as IFTTT [11] and Zapier [14]. It follows the Event Condition Action (ECA) paradigm [1, 8, 6] and is based on three core abstractions: i) *event sources*, ii) *action targets* and iii) *rules*. *Event sources* and *action targets* are meant be developed by third-party developers, independently from any specific application (to encourage reuse). Application developers implement workflows on top of available *event sources* and *action targets*, by defining stateless rules. Essentially, they express rules such as "*if* an event with properties that match these conditions is notified, *then* trigger an action on this target, with the following properties".

3.1 Event Sources

An *event source* is an autonomous component that produces a stream of typed events. Based on this definition, there are quite different types of event sources. Here are some examples:

- *Connected hardware sensors* that emit a continuous flow of low-level events. In this case, the events are observations or measures captured by the sensors.
- *Software sensors* that capture some kind of activity in a digital system and emit related events. For instance, one can think of a software sensor embedded in a business application that emits an event whenever a business-level condition is met.
- *Data processing services* that emit higher-level events. Typically, a data processing service aggregates several streams of low level events and applies some kind of logic to produce a new stream.
- *User agents* used as proxy to emit human generated events. For instance, think of a mobile application used to report incidents.

The iFLUX middleware exposes a standard REST API that *event sources* use to stream their data. The API specifies a simple payload structure: an event is defined by a timestamp, a source, a type and a list of properties. The list of properties depends on the event type.

3.2 Action Targets

An *action target* is a component that exposes logic that can be triggered from the iFLUX middleware. Here also, there are different types of action targets:

- *Connected hardware actuators* that can be remotely controlled. A smart street light or a large public display located in a stadium are two examples.
- *Software actuators* that are typically business applications or gateways deployed for integration purposes.
- *User interaction channel gateways* that are special software actuators geared at delivering notifications to people. Examples include gateways for delivering e-mails, push notifications and social network notifications.

The developers of *action targets* must implement a simple REST API and process incoming action payloads. An action payload is defined by a timestamp, a type and a list of properties.

3.3 Rules

The third abstraction in the iFLUX model is the notion of rule. Rules are what bind events and actions together. A rule specifies that *if* an event is notified and its properties meet certain criteria *then* one or more actions has to be triggered with a list of properties (which values are computed based on the event properties).

4. IMPLEMENTATION

At the inception of the iFLUX project, we have decided to apply the *lean* and *agile* principles. We have made the choice to produce and release a first version of the system quickly. We wanted to have something that we and others could experiment with rapidly, so that we would rapidly benefit from feedback. After a few weeks, we have been able to implement our first *event sources* and *action targets*, and we were able to assemble applications by specifying rules. Since the project inception, we have relied on virtualization technologies, such as Vagrant and Docker, to make it easy for third-party developers to have a working environment on their machines. The very first user interface only provided a mechanism to create *rules* and to simulate their evaluation. Later on, we have expanded the user interface to enable the configuration of *event sources* and *action targets* as well. We now do most of our development on top of the Node.js platform, but we also SDKs in other languages.

4.1 REST APIs

The three core abstractions of the iFLUX programming model are exposed in corresponding REST endpoints (there are additional endpoints that space prevents us to fully describe here). Two endpoints, namely `/events/` and `/rules/`, are implemented by the iFLUX middleware (i.e. developers who create *event sources* or workflows act as clients for these endpoints). The third one, namely `/actions/`, has to be implemented by the developers who create *action targets*.

4.1.1 *The `/events/` endpoint*

This endpoint and the payload that it accepts is straightforward. Every iFLUX *event source* produces a stream of events. It POSTs these events on the endpoint. Notice that the payload accepts an array of events, so it is possible to send several events in a single request. For every event, there is a timestamp, a source (a link to the event source resource), a type (a link to a JSON schema) and an array of custom properties. As you can see, writing an iFLUX event source is very simple. In particular, bringing an existing component (WoT gateway, mobile app, business application, etc.) into the iFLUX ecosystem is not a burden, which was one of our main design objectives. Furthermore, this can be done on any kind of platform (software and firmware), since the only requirement is to be able to issue an HTTP request.

```
POST /events/ HTTP/1.1
Content-type: application/json

[
  {
    "timestamp" : "2015-01-12T05:21:07Z",
    "source" : "/event-sources/JI8928JFK",
    "type" : "/eventTypes/temperatureEventSchema",
    "properties" : {
      "temperature" : 22.5,
      "location" : "room 1"
    }
  },
  {
    "timestamp" : "2015-01-12T05:22:07Z",
    "source" : "/event-sources/JI8928JFK",
    "type" : "/eventTypes/temperatureEventSchema",
    "properties" : {
      "temperature" : 22.8,
      "location" : "room 1"
    }
  }
]
```

4.1.2 *The `/actions/` endpoint*

This endpoint is analogous to the previous one. It has to be implemented by every *action target*, to expose functionality that can be triggered when iFLUX rules are evaluated positively. Here again, it is possible to send several action payloads in a single HTTP request. Every action has a type (a link to a JSON schema) and a list of custom properties that depend on this type. Notice that there is no link to the action target, since this information is already part of the action resource URL (e.g. `https://myactuator.mysystem.com/actions/`)

```
POST /actions/ HTTP/1.1
Content-type: application/json

[
  {
    "type" : "/actionTypes/sendAlertViaEmailSchema",
    "properties" : {
      "email" : "user.name@iflux.io",
      "subject" : "Alert: something has happened!",
      "body" : "An event has been notified to iFLUX by a source and a rule
            states that we should inform you about it."
    }
  },
  {
    "type" : "/actionTypes/sendAlertViaEmailSchema",
    "properties" : {
      "email" : "user.name@iflux.io",
      "subject" : "Alert: something has happened!",
      "body" : "An event has been notified to iFLUX by a source and a rule
            states that we should inform you about it."
    }
  }
]
```

4.1.3 *The `/rules/` endpoint*

This endpoint is used to perform CRUD operations of iFLUX rules. Unlike the two previous endpoints, which only accept POST requests, it accepts GET, POST, PATCH and DELETE requests. In addition to the simple metadata (a description and a reference), every rule has two parts: the *if* part and the *then* part. The *if* part specifies under which conditions the rule will be fired. It is possible (but not mandatory) to specify an event source, an event type and a list of property values. In other words, it is possible to define rules that are fired "*if any sensor sends an event with* *a temperature above 30 degrees*" or "*if the particular sensor in my kitchen sends any kind of event*". The *then* part specifies which action(s) should be triggered on which *action target(s)* when the rule is fired. The property `actionSchema` contains a Handlebars [10] template, which specifies how to create the action payload based on the event properties. In the example below, `properties.temperature` is replaced with the value of the `temperature` event property.

```
POST /actions/ HTTP/1.1
Content-type: application/json

{
  "description" : "When a temperature event is received, notify Bob by
        email.",
  "reference": "TEMPERATURE-EMAIL-NOTIFICATION",
  "enabled": true,
  "if" : {
    "eventSource" : "/event-sources/JI8928JFK",
    "eventType" : "/eventTypes/temperatureEventSchema",
    "eventProperties" : {}
  }
  "then" : [{
    "actionTarget" : "https://mail-gateway.iflux.io/api",
    "actionSchema" : "{\"type\" : \"sendEmail\", \"properties\" : {\"to\" :
          \"bob@iflux.io\", \"subject\" : \"New temperature\", \"body\" :
          \"The temperature in  is now: {{ properties.temperature }}.\" }}"
  }]
}
```

5. USE CASES AND EVALUATION

As of today, we have mostly evaluated the iFLUX from a developer's point of view. We have looked at the following questions: how easy is it to grasp the programming model? How easy is it to create a new service or to expose a legacy service through iFLUX APIs? How easy is it to build an application by applying the programming model? To answer these questions, we have worked with various teams to develop both components and applications. These teams had no prior knowledge about iFLUX before starting the exercise. In all cases, we saw that it was easy and quick both to implement iFLUX APIs in existing components and to design end-to-end workflows.

5.1 PubliBike

Like in many other countries, bike sharing stations are increasingly deployed in swiss cities (see Figure 3). Customers need to acquire a smart card, with which they can unlock a bike at a station. They also use the smart card when they later return the bike. The bike stations are connected to a nation-wide information system, named PubliBike. PubliBike makes it possible to know the number of available bikes and free slots at every station, in realtime. The data can be accessed via a REST API: the returned JSON payload contains the current state of all stations.

Figure 3: A bike station in Yverdon-les-Bains

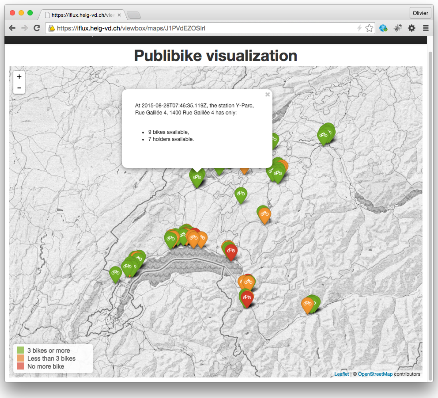

Figure 4: Bike stations displayed by an action target

As illustrated in Figure 1, we have implemented one *event source* and two *action targets* and we have defined rules in order to react to the activity monitored within the PubliBike system. These components are described in the following paragraphs.

5.1.1 Tracking bike arrival and departure: the PubliBike Event Source

Integrating PubliBike into the iFLUX ecosystem has first been achieved by defining a new *event source* and by specifying the type of events produced by this source. Note that when doing this, we did not need to think about how the information would be used. It could be used to trigger alerts, to create visual representations, to compute statistics. As developers of the *event source*, this is not something that we had to worry about (decoupling). We could have decided to define one *event source* for every bike station, but instead we have preferred to define a single *event source*: the PubliBike gateway, which is responsible for polling data via the PubliBike API and to detect state changes. In this scenario, while there are sensors and communication modules embedded in the physical stations, the iFLUX *event source* is purely implemented in a software daemon running in the cloud. With the PubliBike *event source* deployed, iFLUX receives an incoming stream of events, where every event represents a state change at a given station (i.e. either a bike has arrived or left). In addition to a timestamp, the events contain the following properties: the identifier, name and geographic coordinates of the station, the number of available bikes and the number of free slots.

5.1.2 Notifying users: the Slack action target

Someone who uses the bike sharing service to commute from the office to the train station might be interested to be notified if the number of available bikes close to the office falls below a certain threshold. This person might also be interested to receive an alert if the number of free slots at the station falls below a threshold. To implement this use case, the user needs an iFLUX *action target* that provides a bridge to a notification system (SMS, e-mail, etc.). To illustrate the idea, we have implemented an action target that makes it possible to send a notification in the Slack instant messaging platform [13]. The payloads sent to the action target via the REST API contain the message and the name of the channel where to post it.

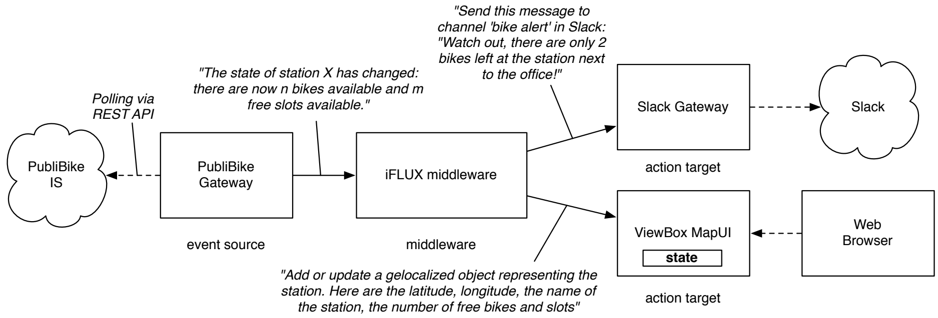

Figure 1: The PubliBike application, with one event source and two action targets

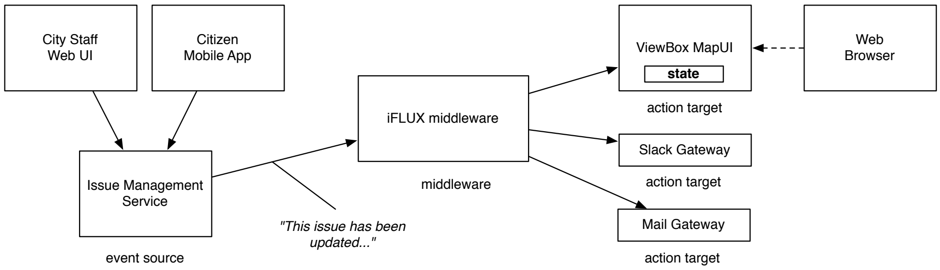

Figure 2: The Citizen Engagement application, with one event source and three action targets

This allows the user to define an iFLUX rule, which states that ***IF*** *an event is received from the PubliBike event source AND the event property 'stationId' of the event is the one of the station close to my office AND if the event property 'numberOfAvailableBike' is less than 3* ***THEN*** *send an action to the Slack action target, with the property 'channel' set to 'bike alerts' and the property 'message' set to 'WARNING: there are not many bikes left at the station!'*.

5.1.3 Map visualization: the ViewBox action target

Another idea for using the data produced by the PubliBike event source was to create a visual representation, where the current state of every bike station is shown on a geographical map (see Figure 4). This use case is interesting, because it raises the question about how to deal with application state given that iFLUX rules are stateless.

We have implemented an *action target* that we have named the *ViewBox action target*. Note that while it can be used in conjunction with the *PubliBike event source*, it is generic and can be used with other types of *event sources* (for instance, it has been used in the citizen engagement application described later). Essentially, the *ViewBox action target* is responsible for managing application state, which is defined by a collection of geolocalized objects, which can have arbitrary properties attached to them. It accepts action payloads with the following properties: the unique identifier of a geolocalized object, the current latitude and longitude and a list of application specific values (in the case of PubliBike, the name of the station and the number of available bikes and slots). When it receives an action via the REST endpoint, it creates or updates a geolocalized object with the property values in the payload. The *action target* provides its own API (outside the scope of iFLUX), so that web browsers can fetch annotated maps.

After deploying the *Viewbox action target*, we were able to configure a rule so that whenever an event was received from the *PubliBike event source*, an action would be sent to the *ViewBox action target* to update the corresponding geolocalized object.

5.2 Citizen Engagement

After a few months of work on iFLUX, we used the platform in a two-weeks undergraduate course dedicated to end-to-end mobile services. The course is project-oriented and every year, we use an application domain to provide some context to the students. This year, we explained that software platforms are increasingly deployed, so that citizen can report issues to city authorities [7, 5]. Users can report broken street lights, graffitis, dangerous areas, etc. The students were asked to design a system with two components. Firstly, they had to implement a simple issue tracking system and to expose their domain model via a RESTful API. Secondly, they had to implement a mobile app that would be provided to citizen, so that they could easily report issues and follow their resolution process.

The Citizen Engagement back-end was then transformed into an iFLUX *event source*. This was done by emitting an event whenever the state of an issue would change (created, acknowledged, in progress, resolved, etc.). Special properties were added to the event (e.g. to attach comments to state transitions). Again, since emitting an iFLUX event is not more complicated than issuing a POST request, the integration was trivial. We were then able to combine the new *event source* with existing *action targets*. It was really easy to implement a workflow to notify city staff about new issues, either via Slack or via email. It was also very easy to create a map to visualize the issue with the *Viewbox action target* described before. Figure 2 shows the end-to-end workflow in iFLUX. Several rules have been added to trigger behavior in the action targets whenever an issue is updated.

Figure 5: The event source after the festival

5.3 Parking @ Paléo

Paléo Festival is one of the largest music festivals in Switzerland. This year, had the opportunity to evaluate several iNUIT projects in a proof-of-concept deployment. One need expressed by the organizers was to get realtime information about the flow of vehicles and the occupancy of the parkings. To address this question, we created a system composed of one iFLUX *event source* and one iFLUX *action target*.

The *event source* is a smart object located at the entrance of the parking (see Figure 5), which detects vehicles with ultrasonic sensors. Connected to the Internet via a 802.15.4 mesh network and a WoT gateway, the object emits an event every time a car enters or leaves the parking. The *action target* is responsible for managing application state, which consists of the number of cars currently in the parking, as well as aggregate metrics about the flow of vehicles (number of entries and departures per minute, hour and day). The *action target* publishes this information via a custom REST API, which is used by a web dashboard.

5.4 Awareness @ Novaccess

Novaccess is a startup which develops an integrated stack (hardware, firmware, software) for the industrial Internet of Things. Novaccess has developed NovaLight, a smart lighting solution, which allows municipalities to reduce costs by lowering energy consumption and improving maintenance procedures. Measures are collected from street lights (consumption, faults, etc.) and analyzed. The platform can also dynamically and remotely adjust lighting levels.

We have worked with the Novaccess team to develop an *awareness system* with iFLUX, i.e. a system which collects various types of events and generates notifications for the Novaccess staff. Several *event sources* have been created. The first source, embedded in the NovaLight software, emits events that correspond to user actions (e.g. user has logged in, user has sent a command to a street light, etc.) or to technical issues (e.g. the database size has reached a given threshold). The second source, embedded in the NovaLight IoT gateway, emits events that correspond to measures or technical issues. The *action target* used in the system is the Slack gateway presented before. Rules have been configured on the iFLUX middleware to send the text notifications via Slack, when *interesting* events happen. The flexibility of the setup comes from the fact that it is possible to configure more than a rule. This allows the team to fine tune the amount and destination of the generated awareness messages. This simple setup could easily be extended with other notification devices (lava lamps, color LEDs, etc.).

6. CONCLUSIONS

We have introduced iFLUX, a middleware that enables a lightweight integration of loosely coupled services. Developed in the context of Smart Cities, it is applicable to other domains. iFLUX is particularly well suited to facilitate prototyping and co-design activities in WoT environments. Hopefully, the examples have shown that bringing an existing component (a smart object, a WoT gateway or a software application) into the iFLUX ecosystem is quick and easy. Even if iFLUX is currently based on stateless Event-Condition-Action rules, we have evidence that valuable workflows can already be implemented from the system. Still, we are currently extending the programming model to support stateful rules. This raises new types of issues, in terms of scalability and performance. These are topics that we have not had the time to discuss in this paper. We have however already integrated several distributed middleware technologies in our implementation. Last but not least, iFLUX is an open source project. More information is available at `www.iflux.io`.

7. REFERENCES

[1] J. Bailey, G. Papamarkos, A. Poulovassilis, and P. T. Wood. An event-condition-action language for xml. In *Web Dynamics*, pages 223–248. Springer, 2004.

[2] M. Batty, K. W. Axhausen, F. Giannotti, A. Pozdnoukhov, A. Bazzani, M. Wachowicz, G. Ouzounis, and Y. Portugali. Smart cities of the future. *The European Physical Journal Special Topics*, 214(1):481–518, 2012.

[3] A. Caragliu, C. Del Bo, and P. Nijkamp. Smart cities in europe. *Journal of urban technology*, 18(2):65–82, 2011.

[4] D. Guinard, V. Trifa, T. Pham, and O. Liechti. Towards physical mashups in the Web of Things. *2009 Sixth International Conference on Networked Sensing Systems INSS*, 7(5):1–4, 2009.

[5] D. Offenhuber. Infrastructure legibility—a comparative analysis of open311-based citizen feedback systems. *Cambridge Journal of Regions, Economy and Society*, page rsu001, 2014.

[6] G. Papamarkos, A. Poulovassilis, and P. T. Wood. Event-condition-action rule languages for the semantic web. In *SWDB*, pages 309–327. Citeseer, 2003.

[7] R. Patel. *A Guide to Open311*. Partridge Publishing, 2015.

[8] Y. Qiao, K. Zhong, H. Wang, and X. Li. Developing event-condition-action rules in real-time active database. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 511–516. ACM, 2007.

[9] J. M. Shapiro. Smart cities: quality of life, productivity, and the growth effects of human capital. *The review of economics and statistics*, 88(2):324–335, 2006.

[10] Handlebars.js web site. `http://handlebarsjs.com`.

[11] Ifttt web site. `https://ifttt.com`.

[12] The reactive manifesto. `http://www.reactivemanifesto.org`.

[13] Slack messaging platform web site. `https://slack.com`.

[14] Zapier web site. `https://zapier.com`.