

Ravel: Programming IoT Applications as Distributed Models, Views, and Controllers

Laurynas Riliskis, James Hong & Philip Levis
Stanford University
lauril@cs.stanford.edu

ABSTRACT

The embedded sensor networks are a promising technology to improve our life with home and industrial automation, health monitoring, and sensing and actuation in agriculture. Fitness trackers, thermostats, door locks are just a few examples of Internet of Things that have already become part of our everyday life. Despite advances in sensors, micro-controllers, signal processing, networking and programming languages, developing an Internet of Things application is a laborious task.

Many of these complex distributed systems share a 3-tier architecture consisting of embedded nodes, gateways that connect an embedded network to the wider Internet and data services in servers or the cloud. Yet the IoT applications are developed for each tier separately. Consequently, the developer needs to amalgamate these distinct applications together.

This paper proposes a novel approach for programming applications across 3-tiers using a distributed extension of the Model-View-Controller architecture. We add new primitive: a *space* - that contains properties and implementation of a particular tier.

Writing applications in this architecture affords numerous advantages: automatic model synchronization, data transport, and energy efficiency.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems; D.1.0 [Software]: Programming Techniques; General

Keywords

Internet of Things; Programming Distributed System; Programming Paradigm; Wireless Sensor Network

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
IoT-App'15, November 1, 2015, Seoul, South Korea.
© 2015 ACM. ISBN ACM 978-1-4503-3838-7/15/11. \$15.00.
DOI: <http://dx.doi.org/10.1145/2820975.2820977>.





	Embedded	Gateway	Cloud
			
Processor	Cortex M3/AVR	ARMv7-8	x86-64
RAM	1-16kB	1-3GB	16+GB
Network	Bluetooth/ZigBee	LTE/WiFi	10G Ethernet
Storage	few MB	few 10s of GB	Peta bytes
OS	Custom C	iOS/Android	Linux
Power supply	200 mAh Battery	3000 mAh Battery	~

Figure 1: An example of common IoT architecture.

1. INTRODUCTION

The Internet of Things (IoT) encompasses a huge variety of applications, ranging from fitness trackers, home automation to health monitoring, sensing and actuation in agricultural environments. These ultra-low-power networks bridge to the larger Internet through gateway: a mobile phone or a device running embedded Linux. Back-end servers in the cloud store data transmitted by gateways and render it to users via web interfaces.

A subset of these applications operates across three tiers of devices as shown in Fig. 1: embedded, gateway and cloud. Each tier has different storage, computation, energy resources and diverse user interfaces (e.g. a blink sequence and tapping input on Fitbit or a web page). One or more low-power embedded devices sense and actuate with the physical environment. These devices can be personal and mobile (e.g. smart watch), shared and stationary (e.g. door lock), or some mixture of the two. Embedded sensing devices often share a similar set of challenges and limitations, including ultra-low power operation, low duty cycles, sensing, actuation, and low-power personal area wireless protocols (802.15.4 or Bluetooth).

IoT apps must manage energy carefully, deal with delay-tolerant networking, run on three different processor architectures and three different operating systems. Changing a small detail in one part of the application (e.g. the format of a data value) requires propagating this change across many languages and components that run on each tier.

Programming such sensor network applications has always been difficult. While there has been tremendous progress in operating systems [13, 4], programming models [11, 5], and

networking for low-power embedded devices, these advances are only part of a much larger puzzle – an IoT programming model consisting of many different types of devices.

Despite these efforts, each tier is developed separately: converting data between different schemas manually. We notice that Model-View-Controller is a software architecture that dominates modern web application development in frameworks such as Rails, Django, and Meteor. Many desktop applications follow such an architecture too (XEmacs: synchronize multiple windows with the same buffer). As a result, MVC’s abstractions and approaches are well understood by the majority of developers today. Therefore we ask, **how can we make the development of IoT applications as easy as the modern web?**

This paper presents Ravel, a programming framework for IoT applications following the three-tier architecture. A Ravel developer writes code in a single language (in our implementation, Python) across all tiers of the system. First, an application is written with a *Distributed Model-View-Controller* architecture. Then, the developer assigns the models and views across *spaces*– the devices that comprise the three-tier application. Finally, Ravel generates static code for each of the spaces that can be compiled and deployed to the devices.

Ravel’s programming abstractions allow a developer to take a complex, distributed sensor network application and write it as a series of models with views and controllers. Ravel fills in all of the intervening pieces, such as network protocols that synchronize models across devices, storage, and scheduling. The high-level description of the system is simultaneously concise and semantically expressive.

This paper has three contributions: (1) A novel Distributed Model-View-Controller programming paradigm for IoT applications that follow a 3-tier architecture. (2) The concept of *spaces*, which allow a programmer to distribute the data processing pipeline across multiple devices, with the Ravel automatically handling data synchronization, encryption, and other networking across spaces. (3) An implementation and an evaluation of the framework based on how well the system enables simple, high-level programming of distributed applications with little overhead over native, hand-written implementations.

Paper organization follow. Section 2 provides an overview of Ravel, that is detailed in Section 3. Section 4 describes the evaluation of the framework. Section 5 gives an overview of programming paradigms for IoT and Section 6 concludes this article.

2. SYSTEM OVERVIEW

A Ravel application is a set of models, views, and controllers, distributed across different devices. Ravel’s model define the data types that an application collects, processes, stores, and displays to a user. In web frameworks, a model is typically bound to a particular database table: they represent a schema and an instance of that schema. Ravel applications are distributed; thus, a model has multiple instances across different spaces; each with a to space specific schema. For example, the model instance on the embedded device is represented as a *Cstruct*, on the Android gateway as a class, and in the backend as a table. Each instance of the model will have distinct storage sizes characteristic to the space. An embedded device, smartphone, and a backend might have the same model of heart rate. However, each of

the model instances would only store a subset of the data (e.g. the embedded device current heart rate, mobile device the latest hour, while the server maintains the full history).

Models can be durable or volatile, and apply data optimizations (e.g., allocation, caching, compression) appropriate to the limited resources of embedded and gateway devices. It is the controller of the model that create records, moves data from in-memory buffers to flash, apply data optimization, move and convert data between schemas and spaces.

Models and controllers are device-independent, but the views are space-specific: the interface of a shower sensor with a few LEDs is different from that of a mobile phone or web application. Ravel provides unified API to fetch the data from models and pass it to a template implementing the view for a particular space.

A *space* is a set of rules, templates, and code snippets for a particular platform (e.g., an Android phone, a TinyOS sensor, a Django server). For example, a *TinyOS* space has information about nesC data types (such as `nx_uint16_t`), uses ActiveMessages, runs transforms within tasks, and uses TinyOS’s storage component for durable models. Rather than individually write code for each tier, Ravel developers assign same models to spaces and adds views to them. Each space specification has all the logic necessary to translate Ravel application code into the code for that particular space. For some spaces, such as Python-based Django, this translation is straightforward because Ravel is implemented in Python, the developer writes an application similarly to the Django framework. For more constrained environments, such as TinyOS/nesC or Contiki, this translation is more complex. Ravel generates the code from space templates and code snippets: by mapping data types from internal to the particular space types and using that to create buffers, communication routines, initialize the main function.

2.1 Water Saving Application

To ground the above concepts in a concrete example, we develop a sensor network application. The application’s goal is to collect fine-grained water use data on the students living in a large dormitory. Currently, the university only knows how much the entire building uses water daily. Without finer-grained data, it is hard to formulate policies to reduce water use (a tremendous concern in California, given the severe drought).

We mount a sensor between the water pipe from the wall and the fixture to monitor water flow and temperature from the showers and sinks in the dormitories. The sensor has a rechargeable battery that is charged by harvesting energy from water flow using a small turbine. The sensor device communicates using Bluetooth Low Energy (BLE) to a smartphone gateway (either Android and iOS). When the gateway is close to a sensor, the application on the smartphone connects to and requests data from the sensor. The gateway stores this data and asynchronously sends it to a python-service (Django) running in the cloud. The sensor encrypts the data that is decrypted only on the cloud. Sensors hold onto records until they receive an end-to-end acknowledgment from the cloud.

3. PROGRAMMING WITH RAVEL

Fig. 2 shows Water Sensing Application in Ravel’s primitives. The main data flow (shown in blue) goes from two

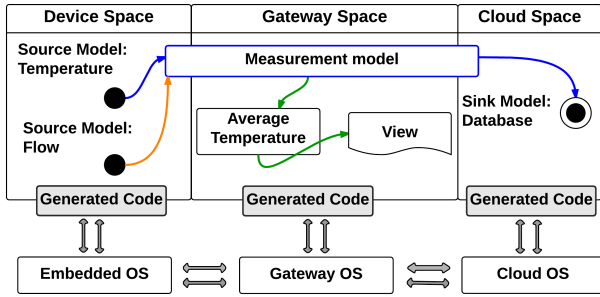


Figure 2: An application spanning three tiers of eMbedded-Gateway-Cloud architecture represented in Ravel primitives.

sources (the temperature and flow sensors) to the sink (the cloud with database) via *Measurement* model. We distinguished notation of source and sink models because they are space specific (they represent a particular hardware or software) and cannot be moved. For example source model for temperature is a sensor on the embedded device. Hence, moving it to the gateway would result in a completely different model.

The *Measurement* model is distributed over all three tiers, thus Ravel will initialize needed storage and communication links to synchronize data between the spaces. The green flow shows a *Average Temperature* model with a view; Ravel creates this model from the *Measurement* model in the gateway space. Because *Average Temperature* is in only one space, it does not need synchronization.

When the developer runs Ravel’s build command, the compiler will generate static code for each of the tiers: including, buffering, storage and communication protocols. The resulting static code can be edited or directly compiled and deployed. The resulting application will read sensors, transmits data to the gateway, show the mean values and forward the measurements to the cloud for storage.

3.1 Models

Essentially, each Ravel model is a set of fields that represents data. We use metaprogramming to override standard class initialization and add necessary fields for the system meta information, class signatures and convenience functionality. For example, each model will get automatic fields such as *node id*, *timestamp*, *sequence number*. The distributed models can be either best-effort or reliable. In the reliable case, a model synchronizing data from the embedded space to the cloud space will not delete a record until it receives an end-to-end acknowledgment from the cloud.

The fields are represented by Ravel’s data type (such as Integer, Float, TimeStamp, and so on) that compiler maps to the type used in each space. For example, if we decide to represent sensor reading as *uint32_t* rather than *uint16_t* the change needs to be performed only in one place – Ravel will propagate this change to all three tiers. Such an approach is tremendously useful when models span multiple spaces: instead of having to redefine the schema for each device and system, the developer only needs to maintain one specification. Additionally, a holistic data types prevent one of the most common causes of security vulnerabilities: wrong type-casting [12].

Ravel automatically synchronizes model instance between spaces when networking is available. Prior that, records are

stored either in memory or in flash depending on defined durability. Because changes to the model can occur on any of tiers while being disconnected, the framework enforces directed data flow: records appearing earlier in the flow are synchronized to later models but not vice-versa.

Ravel models support differing data durability requirements between applications and models within the same device tier. The standard mode is durable, persisting data on flash while the light-weight mode only keeps a RAM buffer. In the latter mode, data may be lost to a system reboot or overflowed buffer.

3.2 Views and Controllers

Views present an external interface (a GUI, JSON) for a set of records drawn from one or more models. View specifications are solely device (and therefore space) specific. On an iOS gateway, the interface might be a native application; in the cloud it might be a web page or JSON data; on the embedded device it might be a small LED or LCD. For this reason, views are the one device-specific component in Ravel. The framework provides a clean set of interfaces that translate data into local representations for feeding into a view.

Controllers respond to pushes from other controllers and interface requests (such as displaying a record). They read and modify their models and invoke views. Ravel controllers extend standard MVC by sending data across spaces; controllers typically do not modify or compute on data as this is done by the model. The developer has the flexibility to specify option such as value update time, sensor reading interval, timeout. Ravel generates controllers as series of timers and interrupt handlers. For example, the temperature sensor would be periodically read, but the data transmission would be contingent upon an available connection.

3.3 Spaces and Templates

Ravel spaces describe the properties and technical details of underlying devices. Technically, a space contains file, function, Makefile and include templates together with software components. Ravel uses templates and software components to assemble controllers, models and views that map them to the underlying OS native types in a particular language. Internally, Ravel builds a graph with the data types and structure particular to space’s programming system. For example, a temperature model component will contain initialization function, includes, build and linking dependencies. Also a controller to read the sensor and write to the *Measurement* – a model in our example.

Ravel uses a templating approach similar to that of popular modern web frameworks. To reduce runtime overhead Ravel generates static code before deployment that allows to inspect and modify code manually.

4. EVALUATION

This section describes Ravel evaluation. Firstly, we examine implicit benefits of developing IoT application in the Distributed Model-View-Controller in a data flow architecture. Secondly, using previously described the water saving application as the example, we compare the required workload (Lines of Code, LoC) to implement the sensor application manually in C to one written in Python using Ravel.

A team of 4 Ph.D. students, one undergraduate, and one post-doctoral researcher developed this application over a

period of 6 months, encountering numerous engineering challenges in the process. For example, Android and iOS have very different Bluetooth interfaces (and programming languages), forcing completely separate gateway implementations. Developers for each component of the system must decide on data representations and bridge those representations across programming languages and architectures: C to Swift, C to Java, Swift to Python, and Java to Python. Moving functionality across devices (compression or encryption) requires entirely new, and separate implementations, and changing the placement of data processing requires new data formats as well as marshaling code. As a result, early design decisions made about how to distribute the application across devices are effectively set in stone, preventing us from making several desired changes before deployment.

4.1 Benefits of the Single Data Model

Having one single data model spanning all three tiers is tremendously helpful. In our example application the data is (1) read from a sensor as `uint32t`, (2) stored in a buffer (as a C-struct containing timestamp `uint32t`, sequence number and node id both `uint16t`), (3) moved to the durable medium (flash). Upon available connection, a record is read from the flash drive, (4) converted to a radio compatible buffer (`uint8t` byte array) and transmitted. When the gateway receives the data (`uint8t` byte array), the application (5) cast byte array to the expected data types (two `uint32t` and two `uint16t` unmangled in the right order). Stored them in (6) typed class and (7) writes to a durable storage. Finally, the data is transmitted to the backend server (8); where it is cast from text/JSON to the appropriate type (9) and stored in the database fields (10).

In Ravel changing data type for from 16 to 32 bits is achieved by one line of code. Whereas manually, it would require changes in ten different places, on three different devices, in three different programming languages as described above. Additionally, such change may require splitting data packets on embedded device (because of the limited radio message size) which would complicate reliable synchronization further.

Another notable advantage of Ravel is the ability to define valid ranges or checks for data: check temperature between 0 and 100C, else alert. Alternatively, monitor invariants that data must satisfy: verify that battery percent increased while water flow is active. To apply such sanity checks and add testing code to our non-Ravel implementation across all three-tier manually (via hacks) has led to numerous bugs as parts of the system were evolving separately. In Ravel, this is automated and does not require additional effort, leading to a system that is not only easier for developers to understand, but also to debug, maintain and detect errors.

4.2 Model Synchronization and Durability

A traditional sensor application often periodically streams data to the permanent gateway. Hence, storing data values on the device is optional, which simplifies the application. In our scenario, we encountered two challenges with such approach: firstly, BLE devices have short communication range thus deploying gateway in each shower would increase cost. Secondly, to connect these gateways to the electrical outlet requires special casings and certification for electronics in wet environments, which made deployment more expensive and complicated. Hence, we decided to crowdsource

Application	O1	O3	Os	Ravel	LoC (C)
Hand written	20308	21936	17180	-	1025
Ravel	13100	13328	10300	80	1201

Table 1: Size of water saving application using Ravel as well as the hand-written implementation, both in bytes of compiled code and lines of source code.

data gathering that requires durable models, and their synchronization.

Ravel model has optional meta class parameters that allows specifying details for synchronization and durability. Enabling the durability flag will automatically initialize flash on embedded device, store data there and read it for transmission. On the gateway, for instance, it would create a database (for example SQLite on Android). Enabled synchronization produces necessary buffers and controllers to move the data from the embedded device to the gateway and further to the cloud. For example, *MeanTemperature* model is only stored on the gateway to store it on the cloud the developer needs to implement database storage on the cloud and communication handlers. A robustly and scalable implementation would require tens to hundreds of lines in Python, Java or C/C++ while inRavel it is sufficient with only one.

4.3 Complete Application

In our comparison, the Ravel developer required to write 80 lines of code for the entire application. That resulted in generated 1201 lines of C code for the embedded device, 2393 for Android (1470 XML, 908 Java and 15 IDL), and 136 Python for Django. That is fewer than the hand written version, but this difference is not significant: it is mostly due to how Ravel partitions functions and performs local variable initialization.

When compiled, the Ravel version uses significantly less code space than the hand-written one, namely 20308 B vs. 13100B as detailed in Tab. 1. The difference took us by surprise - given the somewhat equal lines of code, we would expect somewhat equal code size. The larger code size in the hand-written version is because of its build process: there were some unnecessary object files included in the final executable, left over from prior versions of the application. Once these were removed and the same compilation options were used, the hand-written version had roughly equal code size to the Ravel one. However, this points at the benefit of using a framework, which can automatically minimize not only code but also use toolchains intelligently and in an optimizing way. Ravel helps avoid many common mistakes that only many years of practice and experience prevent.

Noticeably, this evaluation is indicative because of LoC as a main metric. However, it is evident that the time to develop applications in Ravel is significantly shorter. Our approach is more feasible for applications intended for individual use rather than mission critical cyber-physical systems.

5. RELATED WORK

SQL-like query interfaces [15, 16, 2] enable data retrieval from a distributed sensor network as if it were a database. A single query can retrieve data from multiple nodes in the network. Users or applications query data from a heterogeneous sensor network via the gateway, retrieving results in

an endpoint. For example Cougar [25] adds a query proxy layer on top of the application’s runtime. The burden to write the complex code for each tier of the application: individual nodes, gateway, and backend system – is on the developer.

Rather than sending queries, macroprogramming focuses on writing a high-level app for groups of nodes [1, 18, 19, 14, 3, 8]. The paradigm hides communication, storage, and runtime complexity allowing the developer to focus on the application’s data and logic. For example, EcoCast [24] is an interactive object-oriented macroprogramming framework. The developer writes Python code that is later compiled into C code and distributed to nodes. Similar to an SQL-like approach, these systems assume that a particular OS, libraries the run-time is available on the node. Macroprogramming systems do not address cross-tier programming issues: the developer must manually program and change the gateway and cloud, ensuring that the data schemas are compatible after the changes in node applications.

Numerous systems have proposed to simplify development using data streams, which enable the developer to concentrate on data models and information flow rather than low-level programming [17]. These approaches allow users to specify computation on existing data flows and deal with a variety of particular type of stream. For example, MISSA [10] implements a middleware for provisioning generic stream-based services, while SPITFIRE [20] focuses on enabling access to the data from connected sensors as a semantic web. Yet, these approaches assume either a secondary developer [23] (who implements the system according to the specification) or the existence of the enterprise infrastructure with deployed code for each tier.

Other systems, for example, SNACK [7], propose a configuration language, with a library and compiler for the development of Wireless Sensor Networks. Instead of creating a new language, WuKong [21, 22] uses a flow-based programming paradigm in a familiar Java environment. The developer constructs an application from logical components in WuKong’s library and later distributes the application as binary executable for embedded Java VM.

To address three-tier programming, Exemplar [9] allows the developer to demonstrate the desired physical interaction and bind it to a resulting action. The system analyzes the sensor traces and classifies different interactions (tilting a head left or right). Similarly, Fabryq [6] allows a developer to write an embedded-gateway-cloud application as a centralized Javascript application that interacts with the cloud and embedded device through RPC function calls. These and many other techniques allow a developer to explore quickly and prototype an MGC application. However, they lack deployability: the implementation is tied to a particular hardware that supports the subset scripting language. Also, the primary goal is prototyping. Thus, there is no way to transition an Exemplar or Fabryq prototype to a working system (scale it, move to different hardware, or add new sensors or actuators) besides developing it again from scratch.

The exiting programming paradigms have a set of limitations for the IoT. Firstly: existing frameworks assume access to the data (e.g. nodes) through a central always-connected gateway. Secondly: they require that the data schemas, communication protocols, other components and data types are static in the different tiers of the architec-

ture. For example, SQL-like programming assumes that the system output will have correct and homogeneous data types or that they will be cast correctly even without reprogramming. Thirdly: the newly uploaded programs, scripts or executed queries have to be “standardized” or they will require the gateway and cloud to be reprogrammed, which is not addressed by these systems. Finally, these systems do not address application requirements for user interaction, networked and distributed functionality at each device tier: embedded, gateway, and cloud.

In contrast, Ravel addresses current limitations. Changes in Ravel model are propagated to the code for each of three tiers. Ravel generates static code from templates for each tier in the architecture. Firstly this reduces dependencies on a particular OS, libraries or runtimes. Secondly, it permits the developer to implement and use any existing system and its desired functionality. Finally, Ravel retains the benefits of developing a single application for a three-tier architecture.

6. CONCLUSIONS

This paper presented Ravel, a novel programming framework allowing developers to program 3-tier architecture explicitly in a manner very similar to web applications today, via models, views, and controllers. Ravel introduces concept of *space*, which bind particular models, controllers and views to a specific devices. The networking complexities between devices are hidden by the distributed model that automatically synchronizes whenever possible.

Programming an entire embedded sensor network in a high-level language has been a long-term goal of sensor network research. Ravel suggests that perhaps we should consider programming at an even larger scale, encompassing the gateways and cloud that are part of almost every application.

Acknowledgement

This work was supported, in part, by generous gifts from Ericsson, VMware, SAP, and Panasonic, as well as the Okawa Foundation and the Wallenberg Foundation. This material is based upon work supported by the National Science Foundation under Grant No. 1505728 (CNS: CPS Security) as well as the Intel Corporation.

7. REFERENCES

- [1] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using cosmos. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys ’07, pages 159–172, New York, NY, USA, 2007. ACM.
- [2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In K.-L. Tan, M. Franklin, and J.-S. Lui, editors, *Mobile Data Management*, volume 1987 of *Lecture Notes in Computer Science*, pages 3–14. Springer Berlin Heidelberg, 2001.
- [3] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys ’03, pages 187–200, New York, NY, USA, 2003. ACM.

- [4] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.
- [5] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. Acm, 2006.
- [6] M. Etemadi, W. McGrath, B. Hartmann, and S. Roy. Fabryq: Using phones as smart proxies to control wearable devices from the web. 2014.
- [7] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems, SenSys '04*, pages 69–80, New York, NY, USA, 2004. ACM.
- [8] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairo. In V. Prasanna, S. Iyengar, P. Spirakis, and M. Welsh, editors, *Distributed Computing in Sensor Systems*, volume 3560 of *Lecture Notes in Computer Science*, pages 126–140. Springer Berlin Heidelberg, 2005.
- [9] B. Hartmann, L. Abdulla, M. Mittal, and S. R. Klemmer. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '07*, pages 145–154, New York, NY, USA, 2007. ACM.
- [10] S. Kang, Y. Lee, S. Ihm, S. Park, S.-M. Kim, and J. Song. Design and implementation of a middleware for development and provision of stream-based services. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 92–100. IEEE, 2010.
- [11] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *SenSys*, volume 9, pages 127–140, 2009.
- [12] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 81–96, Washington, D.C., Aug. 2015. USENIX Association.
- [13] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.
- [14] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Trans. Embed. Comput. Syst.*, 5(3):543–576, Aug. 2006.
- [15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, Dec. 2002.
- [16] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.
- [17] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the 1st International Workshop on Data Management for Sensor Networks: In Conjunction with VLDB 2004, DMSN '04*, pages 78–87, New York, NY, USA, 2004. ACM.
- [18] A. Pathak and M. K. Gowda. Srijan: A graphical toolkit for sensor network macroprogramming. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09*, pages 301–302, New York, NY, USA, 2009. ACM.
- [19] A. Pathak, L. Mottola, A. Bakshi, V. Prasanna, and G. Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on*, pages 255–260, March 2007.
- [20] D. Pfisterer, K. Romer, D. Bimschas, O. Kleine, R. Mietz, C. Truong, H. Hasemann, A. KroÏller, M. Pagel, M. Hauswirth, M. Karnstedt, M. Leggieri, A. Passant, and R. Richardson. Spitfire: toward a semantic web of things. *Communications Magazine, IEEE*, 49(11):40–48, November 2011.
- [21] N. Reijers, K.-J. Lin, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu. Design of an intelligent middleware for flexible sensor configuration in m2m systems. In *SENSORNETS*, pages 41–46, 2013.
- [22] N. Reijers, Y.-C. Wang, C.-S. Shih, J. Hsu, and K.-J. Lin. Building intelligent middleware for large scale cps systems. In *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*, pages 1–4, Dec 2011.
- [23] S. Tranquillini, P. SpieÅ§, F. Daniel, S. Karnouskos, F. Casati, N. Oertel, L. Mottola, F. Oppermann, G. Picco, K. RÄümer, and T. Voigt. Process-based design and integration of wireless sensor network applications. In A. Barros, A. Gal, and E. Kindler, editors, *Business Process Management*, volume 7481 of *Lecture Notes in Computer Science*, pages 134–149. Springer Berlin Heidelberg, 2012.
- [24] Y.-H. Tu, Y.-C. Li, T.-C. Chien, and P. Chou. Ecocast: Interactive, object-oriented macroprogramming for networks of ultra-compact wireless sensor nodes. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 366–377, April 2011.
- [25] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, Sept. 2002.