Aggregate Programming for the Internet of Things

Jacob Beal, Danilo Pianini, Mirko Viroli

Abstract—Embedded pervasive devices are difficult to effectively coordinate using traditional programming methods due to their rapidly increasing number and density, close ties between functionality and spatial proximity, and the open and ever-changing nature of the network and its applications. Aggregate programming raises the level of abstraction, bringing bulk programming capabilities and a sound bottom-up engineering approach to the ill-controlled and heterogeneous environment of complex Internet of Things applications.

Index Terms—Aggregate programming, pervasive computing, field calculus, distributed systems, domain-specific languages

1 Introduction

The "Internet of Things" (IoT) is ushering in a dramatic increase in the number and variety of networked devices. Personal smart-devices, car control systems, intelligent public displays, drones, digital signs, electronic tags, sensors of various kinds, etc. are all increasingly pervading our everyday working and living environment. Proximity-based interactions between neighboring devices (Figure 1) play a major role in IoT visions, whether intermediated by fixed infrastructure (e.g., [1]), or using peer-to-peer interactions (e.g., [2]), which lower latency and increase resilience to inadequate fixed network infrastructure, e.g., during mass public events or civic emergencies. But are software development methods ready to support such complex and large-scale interactions in an open and ever-changing environment?

Traditionally, the basic unit of computing has been an individual device, only incidentally connected to the physical world through inputs and outputs. This legacy continues to pervade development tools and methodologies, causing many aspects of device interaction—efficient and reliable communication, robust coordination, composition of capabilities, search for appropriate cooperating peers, etc.—to become closely entangled in the implementation of distributed applications. When such applications grow in complexity, they tend to suffer from design problems, lack of modularity and reusability, deployment difficulties, and serious test and maintenance issues.

Aggregate programming provides an alternative that dramatically simplifies the design, creation, and maintenance of complex IoT software systems. Here, the basic unit is no longer a single device, but instead a cooperating collection of devices: details of behaviour,

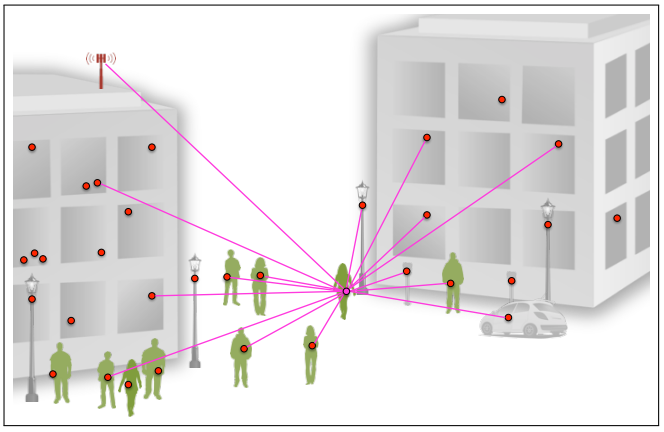


Fig. 1. In a world filled with smart devices and networked objects, every device has the possibility of many opportunistic wireless interactions with other nearby devices, both mobile and stationary. Some of these are elements of network infrastructure, but the vast majority are a heterogeneous mixture of peers.

position and number of devices are largely abstracted away, replaced with a space-filling computational environment. Hence, the IoT environment of many heterogeneous devices becomes less a concern and more an opportunity to increase the quality (e.g., soundness, stability, efficacy) of application services. This is accomplished by a layered approach to programming complex services, building on foundational work on the composition of distributed systems, then on general mechanisms for robust and adaptive coordination, to ultimately provide engineers with a relatively simple programming API that still implicitly guarantees safety and resilience.

Such a framework is particularly useful for large-scale scenarios with inadequate fixed network infrastructure, such as crowd management at mass public events. In these environments, opportunistic interactions between devices (e.g., people's smart-phones)

- *Jacob Beal is with Raytheon BBN Technologies, Cambridge, Massachusetts, USA.Email:* jakebeal@bbn.com
- *Danilo Pianini and Mirko Viroli are with Universita di Bologna, ItalyEmail:* [danilo.pianini,mirko.viroli]@unibo.it

can smoothly support services such as crowd detection, crowd dispersal, and crowd-aware navigation. We illustrate the power of aggregate computing by showing how simply some examples of such crowd services can be implemented and composed, empirically demonstrating resilience and adaptivity of the resulting services using data gathered from an actual mass public event.

2 Aggregate Programming

The limits of the single-device viewpoint have been widely recognised, motivating work toward aggregate programming in many different domains, as surveyed in [3]. The main strategies taken are generally: making device interaction implicit (e.g., TOTA [4]), composing geometric and topological constructions (e.g., Origami Shape Language [5]), summarizing data over space-time regions and streaming it to other regions (e.g., TinyDB [6]), automatically splitting computations for cloud-style execution (e.g., MapReduce [7]), and providing generalizable constructs for space-time computing (e.g., Protelis [8]). The last are particularly well suited for an IoT environment, being explicitly designed for distributed operation in a physical environment filled with embedded devices.

The successes and pitfalls of the many prior efforts suggest some key observations about programming large-scale situated systems: *(i)* mechanisms for robust coordination should be hidden "under-the-hood" where programmers are not *required* to interact with them, *(ii)* composition of modules and subsystems must be simple and transparent, and *(iii)* different subsystems need different coordination mechanisms for different regions and times. Aggregate programming aims to address these issues using the following three principles:

1) the "machine" being programmed is a region of the computational environment whose specific details are abstracted away (perhaps even to a pure spatial continuum);
2) the program is specified as manipulation of data constructs with spatial and temporal extent across that region; and
3) these manipulations are actually executed by the individual devices in the region, using resilient coordination mechanisms and proximity-based interactions.

For example, consider the two diagrams of smartphone-hosted crowd safety services in Figure 2. In this example, smart-phones interact to estimate crowd density and distribution, which is used as input for several services: one warns people of nearby dangerously dense regions (where there is risk of panic or trampling), another provides advice for dispersing from such regions, and a third helps others navigate through the crowd while avoiding dangerous

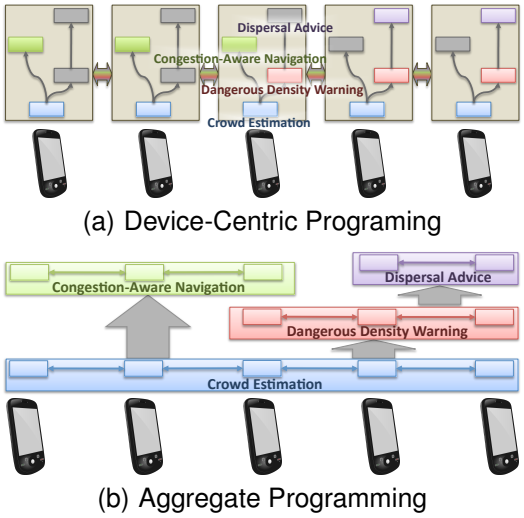


Fig. 2. Device-centric programming of distributed algorithms (a) versus aggregate programming (b): with aggregate programming, algorithmic building blocks can be scoped and composed directly for the aggregate.

areas. With traditional device-centric approaches (Figure 2(a)), the programmer needs to focus on the protocol for device interactions while simultaneously reasoning about how that local interaction will produce the desired complex global behavior. With aggregate programming, on the other hand, one instead naturally reasons in terms of incremental construction from continuum-like data structures and services (Figure 2(b)): in this example, crowd estimation produces as output a distributed data structure—a "computational field" [9], [4]—mapping from location to crowd density. This then serves as an input for crowd-aware navigation, which outputs vectors of recommended travel, and for the warning service, which produces a map of warnings that are in turn an input for producing dispersal advice. From this composition of data structures and services, the precise protocol details can then be generated automatically. By thus separating service composition from details of coordination and interaction protocols, aggregate programming promotes construction of more complex, reusable and composable distributed services.

3 Building up to Aggregate APIs

Aggregate programming hides the complexity of distributed coordination in IoT network environments using several layers of abstraction (Figure 3). Its foundation is *field calculus* [9], a core set of constructs modeling computation and interaction amongst large numbers of spatially embedded devices (in particular, this paper uses Protelis [8], a Java-based field calculus implementation with support for first-class aggregate functions). Upon this foundation, we can identify key "building blocks" for resilient coordination, then combine these to produce APIs for common application

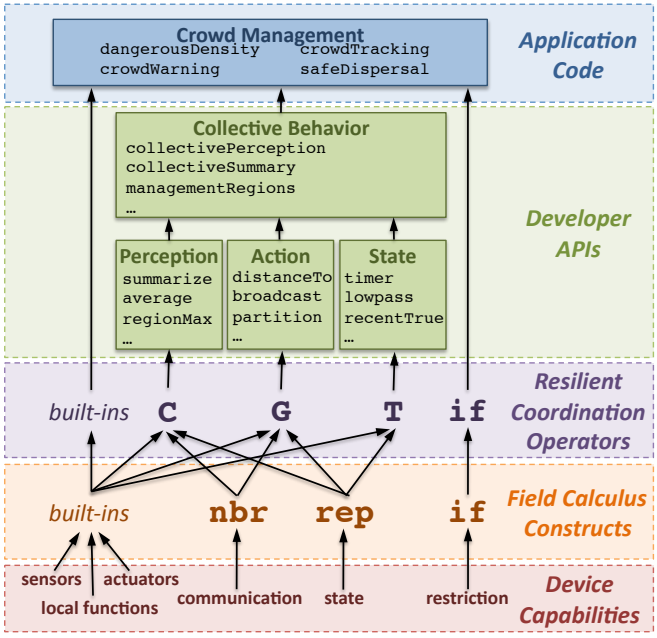


Fig. 3. Layers implementing practical aggregate programming: the software and hardware capabilities of particular devices are abstracted by using them to implement a small universal calculus of aggregate-level field calculus constructs. This calculus is then used to implement a limited set of "building block" coordination operations with provable resilience properties, which are then wrapped and combined together to produce a user-friendly API for developing situated IoT systems.

needs like sensing, decision, and action, creating a collective behavior API for transparent implementation of complex networked services and applications [10].

This framework enables simple specification of complex, resilient distributed systems, as we will see in Section 4. As such a specification is realized, implicit details are made explicit: first which resilient coordination operators are used, then how those operators are implemented, how aggregate specification maps to actions by individual IoT devices, and finally how those devices actually implement capabilities like sensing, communication, and localization.

3.1 Field Constructs

Certain interaction patterns appear across many different aggregate programming approaches. *Field calculus* [9] captures these essential features in a tiny universal language suitable for mathematical analysis. This layer (second lowest in Figure 3) is also where aggregate programming interfaces with the open world of device infrastructure and non-aggregate software services (together comprising the lowest layer).

The unifying abstraction of field calculus is a *field*, inspired by physical concepts like magnetic fields, which maps each networked device to some local value. In field calculus every expression, value, or variable is a field: for example, a collection of temperature sensors produce a field of ambient temperatures, smart-phone accelerometers produce a field of movement directions, and a notification application produces a field of messages displayed on phones. Fields are constructed and manipulated using four program constructs:

- **Functions:** $\texttt{b}(\texttt{e}_1,\ldots,\texttt{e}_n)$ applies function `b` to arguments $\texttt{e}_1,\ldots,\texttt{e}_n$. Such "built-in" functions are stateless mathematical, logical, or algorithmic functions, sensors or actuators, or user-defined or imported library methods.
- **Dynamics:** $\texttt{rep}(\texttt{x}\texttt{<-}\texttt{v})\{\texttt{s}_1;\ldots;\texttt{s}_n\}$ defines a local state variable `x` initialized with value v and periodically updated with the result of executing its body statements $\{\texttt{s}_1;\ldots;\texttt{s}_n\}$, thereby defining a field that evolves over time.
- **Interaction:** $\texttt{nbr}(\texttt{s})$ gathers a map at each device (actually, a field) from all neighbors (including itself) to their latest value of `s`. Built-in "hood" functions then summarize such maps, e.g., $\texttt{minHood}(\texttt{m})$ finds the minimum value in map `m`.
- **Restriction:** $\texttt{if}(\texttt{e})\{\texttt{s}_1;\ldots;\texttt{s}_n\}\ \texttt{else}\ \{\texttt{s}'_1;\ldots;\texttt{s}'_m\}$ partitions the network into two regions: where `e` is true $\texttt{s}_1;\ldots;\texttt{s}_n$ is computed, elsewhere $\texttt{s}'_1;\ldots;\texttt{s}'_m$ is computed instead. Importantly, partition implies branches are encapsulated and cannot have effects outside their subspace.

Each construct can be interpreted equivalently as either aggregate-level field manipulation or into protocols for individual devices implementing such manipulations. Field calculus is also universal [11], supporting any causal, approximable space-time computation. As will be demonstrate in Section 4, the field calculus can express distributed services safely and predictably composed and modulated.

These constructs also support portability, infrastructure independence, and interaction with non-aggregate services. In fact, aggregate programming can incorporate any device or infrastructure implementing them, including heterogeneous mixtures of devices with different sensor, actuator, computation, and communication capabilities. Likewise, complementary non-aggregate software services, whether local or cloud-based, can be integrated simply by importing their APIs into the aggregate programming environment [8].

3.2 Building Blocks for Resilient Coordination

The next level of abstraction adds resilience, identifying a collection of general "building block" operators for resilient coordination applications. This layer (middle in Figure 3) comprises coordination mechanisms that are *(i)* self-stabilizing, meaning they reactively adjust to changes in network structure or input values, *(ii)* scalable to large networks, and *(iii)*

```
def G(source, initial, metric, accumulate) {
  rep(dv <- [Infinity, initial]) {
    mux(source) {
      [0, initial]
    } else {
      minHood([nbr(dv.get(0)) + metric.apply(),
               accumulate.apply(nbr(dv.get(1)))])
    }
  }.get(1)
}
```

Fig. 4. Protelis implementation of operator G

preserve these resilience properties when composed with one another. Any service constructed from these "building blocks" is thus implicitly resilient as well.

One such collection is identified in [10]: three generalized coordination operators plus field calculus' `if` and built-ins. The three operators are:

- `G(source,init,metric,accumulate)`: a "spreading" operation generalizing distance measurement, broadcast, and projection, executes two tasks: it computes a field of shortest-path distances from a `source` region (indicated as a Boolean field) using the supplied `metric`, then propagates values up the distance gradient, beginning with value `initial` and accumulating along the gradient with `accumulate`.
- `C(potential,accumulate,local,null)`: accumulates information to the `source` down the gradient of a `potential` field. Beginning with an idempotent `null`, the `local` value is combined with "uphill" values using a commutative and associative function `accumulate`, producing a cumulative value at the `source`.
- `T(initial,floor,decay)`: flexible countdown with a potentially time-varying rate: function `decay` strictly decreases its input value, starting at `initial` and stopping at `floor`.

These few operators are general enough to cover, individually or in combination, many of the common coordination patterns used in large-scale systems. Implemented in field calculus (e.g., Figure 4), these operators provide an expressive programming environment with strong guarantees of resilience and scalability. Furthermore, the composability proof is modular, allowing expansion of the operator collection by proving a new candidate operator satisfies the same resilience properties as those already in the collection.

3.3 Pragmatic General-purpose APIs

To better meet "day-to-day" programming needs, libraries developed using "building block" operators can apply and combine them to form a pragmatic and user-friendly API that still retains the same properties. Such libraries form the penultimate layer in Figure 3, upon which application code is written.

For example, many distributed action and information diffusion functions can be based on G. One such common computation is estimating distance to one or more designated "source" devices, which can be implemented using G initialized to zero and a metric (nbrRange) of estimated device-to-device distance:

```
def distanceTo(source) {
  G(source, 0, () -> {nbrRange}, (v) -> {v + nbrRange})
}
```

Another common pattern, broadcasting a value from a source, can be implemented:

```
def broadcast(source, value) {
  G(source, value, () -> {nbrRange}, (v) -> {v})
}
```

Other G-based operations include Voronoi partition and a "path forecast" marking paths crossing an obstacle or region of interest.

Similarly, C supports functions related to information perception, such as accumulating the sum of all values of a variable in a region:

```
def summarize(sink, accumulate, local, null) {
  C(distanceTo(sink), accumulate, local, null)
}
```

or, alternately, computing the variable's average or maximum. Likewise, T enables functions of state and memory, such as remembering a value until a specified timeout (relying on the dt built-in to track passage of time):

```
def limitedMemory(value, timeout) {
  T([timeout, value], [0, false],
    (t) -> {[t.get(0) - dt, t.get(1)]}).get(1)
}
```

or implementing a timer or a low-pass filter.

As with any other software library, these API functions can be further combined to create higher level libraries. For example, a summary shared throughout a region can be implemented by applying `broadcast` to `summarize`, and state and partition functions can be combined to organize space into "management regions" by balanced partition into clusters.

Such developer APIs form a practical interface for a typical engineer to develop IoT services using distributed coordination, while building APIs atop resilient operators and field calculus ensures these services are also resilient and safely composable. In parallel, development at lower layers can improve and extend available coordination mechanisms, improve efficiency of field calculus abstractions, and improve interface efficacy with particular device hardware or with non-aggregate applications and services. Layered aggregate programming thus offers the prospect of an efficient software ecosystem for engineering distributed IoT services, analogous to existing ecosystems for web or cloud development.

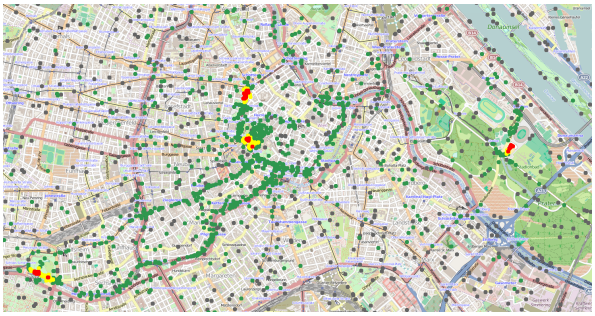
4 Engineering Large-Scale, Opportunistic IoT Services: The Crowd Detection Case

These foundations of aggregate programming and reusable "building block" APIs can greatly simplify construction and composition of resilient applications for IoT scenarios. On the one hand, individual distributed services can be built simply by composition of API functions; on the other hand, the mathematical foundations of aggregate programming, particularly restriction and distributed first-class functions (the ability to pass and call functions just as any other kind of data), enable such services to be dynamically deployed, safely composed and preemptively modulated, just as threads and virtualization enable composition and modulation of services in individual machines and data centers.

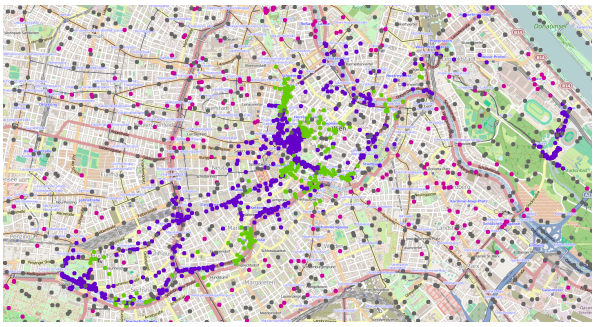
For example, consider how an IoT environment might provide services for crowd safety at mass public events, such as civic festivals, outdoor concerts, or marathons. Such events pose challenging safety issues, because the movement of people in crowded and constrained environments often creates emergent zones of dangerous overcrowding where any small incident can create a panic or stampede that injures or kills people [12], [13]. Moreover, the large number of people and large spatial extent often locally overwhelm the available infrastructure: cell phone networks drop calls, data communications becomes unreliable, public safety personnel are not near an emergent event, etc. In an IoT environment, however, more people means more personal smart-devices, which might coordinate with each other and other IoT devices embedded in their environment, requiring neither cloud services nor centrally deployed infrastructure.

Figure 5(a) shows an ALCHEMIST [14] simulation of such a crowd safety service running in an IoT environment. In particular, we simulate 1000 stationary devices embedded in the environment plus 1479 mobile personal devices, each following a smart-phone position trace collected at the 2013 Vienna marathon [15], as discussed in [16], [13], all communicating via once per second asynchronous local broadcasts with 100 meters range.

This example uses a simple conservative estimate of dangerous crowding via "level of service" (LoS) ratings [17] with LoS D (>1.08 people/m^2) indicating crowds and LoS E (>2.17 people/m^2) in a group of at least 300 people indicating potentially dangerous density. Density is estimated as $\rho = \frac{|nbrs|}{p \cdot \pi r^2 \cdot w}$ where $|nbrs|$ counts neighbors within range r, p estimates the proportion of people with a device running the app (about 0.5% of marathon attendees), and w estimates fraction of walkable space in the local urban environment. Given this estimate, potential crowding danger can be detected and warnings



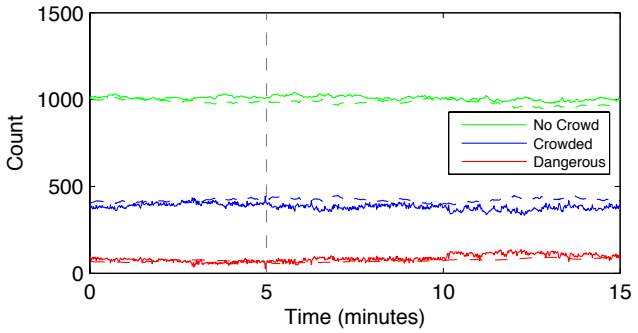
(a) Crowd Density Warnings



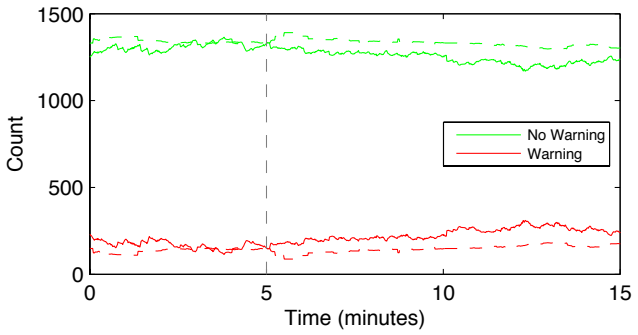
(b) Upgrade In Progress

(c) Priority Modulation

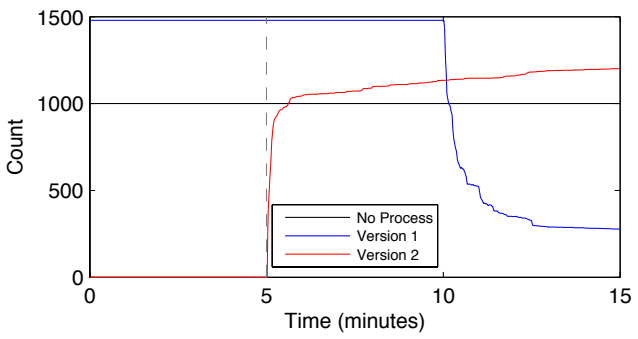
Fig. 5. Snapshots from simulation of crowd safety services in an IoT environment on approximately 2500 personal and embedded devices: (a) A service restricted to run on personal devices (colored) detects regions of dangerous crowd density (red) and disseminates warnings to nearby devices (yellow). (b) Non-disruptive upgrade of a running service disseminates replacement code from injection points: devices running the old version only (red) receive the new version and run encapsulated versions of both (purple) until the new version is ready to take over entirely (green). Note the spatial correlations in color, caused by the progressive spread of the new version outward from several points of injection (now centers of green regions) (c) An external policy composed with the running service prioritizes network resources for the crowd safety service (hotter colors are higher priority) near potential emergency situations for all devices, not just those running the crowd safety service. Note the correspondence of "hot" regions in (c) with "hot" regions in (a).



(a) Crowding LoS across upgrade



(b) Crowd warnings across upgrade



(c) Shift in Versions

Fig. 6. Aggregate programming enables lightweight construction of resilient IoT services, such distributed crowd estimation (a) and warnings of dangerous crowding (b). Despite executing on a large number of highly mobile devices, both services produce estimates (solid lines) that track closely to the true values (dashed lines). Aggregate-level manipulation of distributed services further enables a disruption-free upgrade of these services while running: (c) in this simulation, a new version of crowd management services is injected at the 5-minute mark (vertical grey dashed line), but both versions are encapsulated to run concurrently until the new version is ready to take over smoothly from the old, thereby ensuring that there is no significant disruption in either service.

disseminated robustly with just a few lines of Protelis code dynamically deployed and executed on individual devices by a middleware app [8], [9]. The coordination code is realized using aggregate programming API elements:

```
def dangerousDensity(p, r) {
  let mr = managementRegions(r*2, () -> { nbrRange });
  let danger = average(mr, densityEst(p, r)) > 2.17 &&
               summarize(mr, sum, 1 / p, 0) > 300;
  if(danger) { high } else { low }
}

def crowdTracking(p, r, t) {
  let crowdRgn = recentTrue(densityEst(p, r)>1.08, t);
  if(crowdRgn) { dangerousDensity(p, r) } else { none };
}

def crowdWarning(p, r, warn, t) {
  distanceTo(crowdTracking(p,r,t) == high) < warn
}
```

Using the aggregate programing API ensures that this short program is resilient and adapative, allowing it to effectively estimate crowding and distribute warnings (`none`, `low`, `high`) while executing on a large number of highly mobile devices. Figure 6(a) and 6(b) compare number of crowded and warned devices against ideal values across a 15-minute simulation: crowding level tracks very closely, while warnings have a small overestimate, primarily due to brief persistence of warnings after devices leave a warned region.

Beyond resilience, aggregate programming also supports unanticipated composition of processes, a critical need in the open and dynamic environment of typical IoT applications. In our approach, a complex distributed service can be encapsulated and managed as a single aggregate object, which can be modulated and composed with other services [9].

For example, crowd estimation can be "wrapped" with another service for non-disruptive distributed upgrades, which spreads a new version from peer to peer from one or more devices where it is injected. To prevent disruptions, the new version runs alongside the old for some period of time, each safely encapsulated using field calculus' restriction and alignment semantics, and switching over when some criterion is met (e.g., a specified elapsed time). Figure 5(b) shows such an upgrade in progress. This allows a switchover from one version to another without disrupting services, as shown in Figure 6, without building any upgrade capability into the services.

Similarly, encapsulation allows management of service composition with dynamically specified policies. For example, Figure 5(c) shows the effect of a policy prioritizing crowd safety services near dangerous crowd situations. Again, the policy has been wrapped around distributed services not designed to support it, and furthermore acts not just on devices running crowd estimation, but also on other nearby embedded devices, ensuring that unrelated services on those devices do not interfere with emergency communication requirements. Just as with upgrades and resilience,

adopting an aggregate programming model simplifies engineering of complex coordination of services in an open IoT environment.

5 Future Directions

We have seen how aggregate programming may help unlock the true potential of the "Internet of Things" that is coming to permeate our environment. Field calculus and resilient "building block" APIs allow complex distributed services to be specified succinctly, as well as allowing such services to be treated as coherent objects to be safely encapsulated, modulated, and composed with one another.

Aggregate programming thus invites a fundamental change in how we think about engineering IoT systems, as well as a plethora of new investigations. First, hybrid models are needed that take advantage of the complementary capabilities of aggregate programming and cloud-based architectures. Security also needs consideration, since IoT environments are open and involve many actors with different motivations and capabilities. The "building block" APIs discussed here also need to be further developed against real applications to ensure they support a sufficiently wide range of IoT services. Finally, mechanisms for composition and modulation need to be further developed towards a general IoT "operating system," including support on various devices, and encapsulation methods for integrating legacy applications. Together, these all lead towards a future where complex distributed systems are just as simple to engineer as individual computers.

Acknowledgments

This work has been partially supported by the EU FP7 project "SAPERE - Self-aware Pervasive Service Ecosystems" under contract No. 256873 (Viroli), by the Italian PRIN 2010/2011 project "CINA: Compositionality, Interaction, Negotiation, Autonomicity" (Viroli), and by the United States Air Force and the Defense Advanced Research Projects Agency under Contract No. FA8750-10-C-0242 (Beal). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, opinions, and/or findings contained in this article are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release; distribution is unlimited.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012.

[3] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, M. Mernik, Ed. IGI Global, 2013, ch. 16, pp. 436–501, a longer version available at: http://arxiv.org/abs/1202.5509.

[4] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The tota approach," *ACM Trans. on Software Engineering Methodologies*, vol. 18, no. 4, pp. 1–56, 2009.

[5] R. Nagpal, "Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, MIT, 2001.

[6] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *Workshop on Mobile Computing and Systems Applications*, 2002.

[7] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] D. Pianini, J. Beal, and M. Viroli, "Practical aggregate programming with PROTELIS," in *ACM Symposium on Applied Computing (SAC 2015)*, 2015, to appear.

[9] F. Damiani, M. Viroli, D. Pianini, and J. Beal, "Code mobility meets self-organisation: A higher-order calculus of computational fields," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. Lecture Notes in Computer Science, S. Graf and M. Viswanathan, Eds. Springer International Publishing, 2015, vol. 9039, pp. 113–128. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-19195-9_8

[10] J. Beal and M. Viroli, "Building blocks for aggregate programming of self-organising applications," in *Workshop on Foundations of Complex Adaptive Systems (FOCAS)*, 2014.

[11] J. Beal, M. Viroli, and F. Damiani, "Towards a unified model of spatial computing," in *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France, May 2014.

[12] G. K. Still, *Introduction to Crowd Science*. CRC Press, 2014.

[13] B. Anzengruber, D. Pianini, J. Nieminen, and A. Ferscha, "Predicting social density in mass events to prevent crowd disasters," in *Social Informatics*, ser. Lecture Notes in Computer Science, A. Jatowt, E.-P. Lim, Y. Ding, A. Miura, T. Tezuka, G. Dias, K. Tanaka, A. Flanagin, and B. Dai, Eds. Springer International Publishing, 2013, vol. 8238, pp. 206–215. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-03260-3_18

[14] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with Alchemist," *Journal of Simulation*, 2013. [Online]. Available: http://www.palgrave-journals.com/jos/journal/vaop/full/jos201227a.html

[15] F. Zambonelli, A. Omicini, B. Anzengruber, G. Castelli, F. L. D. Angelis, G. D. M. Serugendo, S. Dobson, J. L. Fernandez-Marquez, A. Ferscha, M. Mamei, S. Mariani, A. Molesini, S. Montagna, J. Nieminen, D. Pianini, M. Risoldi, A. Rosi, G. Stevenson, M. Viroli, and J. Ye, "Developing pervasive multi-agent systems with nature-inspired coordination," *Pervasive and Mobile Computing*, vol. 17, Part B, no. 0, pp. 236 – 252, 2015.

[16] D. Pianini, M. Viroli, F. Zambonelli, and A. Ferscha, "HPC from a self-organisation perspective: The case of crowd steering at the urban scale," in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, July 2014, pp. 460–467.

[17] J. Fruin, *Pedestrian and Planning Design*. Metropolitan Association of Urban Designers and Environmental Planners, 1971.

Jacob Beal is a scientist at Raytheon BBN Technologies. His research focuses on the engineering of robust adaptive systems, and particularly on the problems of aggregate-level modeling and control for spatially distributed systems like pervasive wireless networks, robotic swarms, and natural or engineered biological cells. He is an associate editor of “ACM Transactions on Autonomous and Adaptive Systems,” on the steering committee of the IEEE SASO conference, and a founder of the Spatial Computer Workshop series.

Mirko Viroli is Associate Professor at DISI, the Computer Science Department of the University of Bologna. He is an expert in programming languages, computational models, and engineering of self-adaptive and self-organising systems, and has written over 200 articles on such topics. He is member of the Editorial Board of the “Knowledge Engineering Review” (Cambridge University Press), and was program chair of ACM SAC 2008 and 2009, IEEE SASO 2014, and IFIP COORDINATION 2015.

Danilo Pianini is post-doctoral researcher at DISI, the Computer Science Department of the University of Bologna. There, he conducts research on pervasive computing and self organization, with a strong focus on engineering, tools, and simulation. He is the chief architect of the Alchemist simulator, and one of the designers of Protelis.