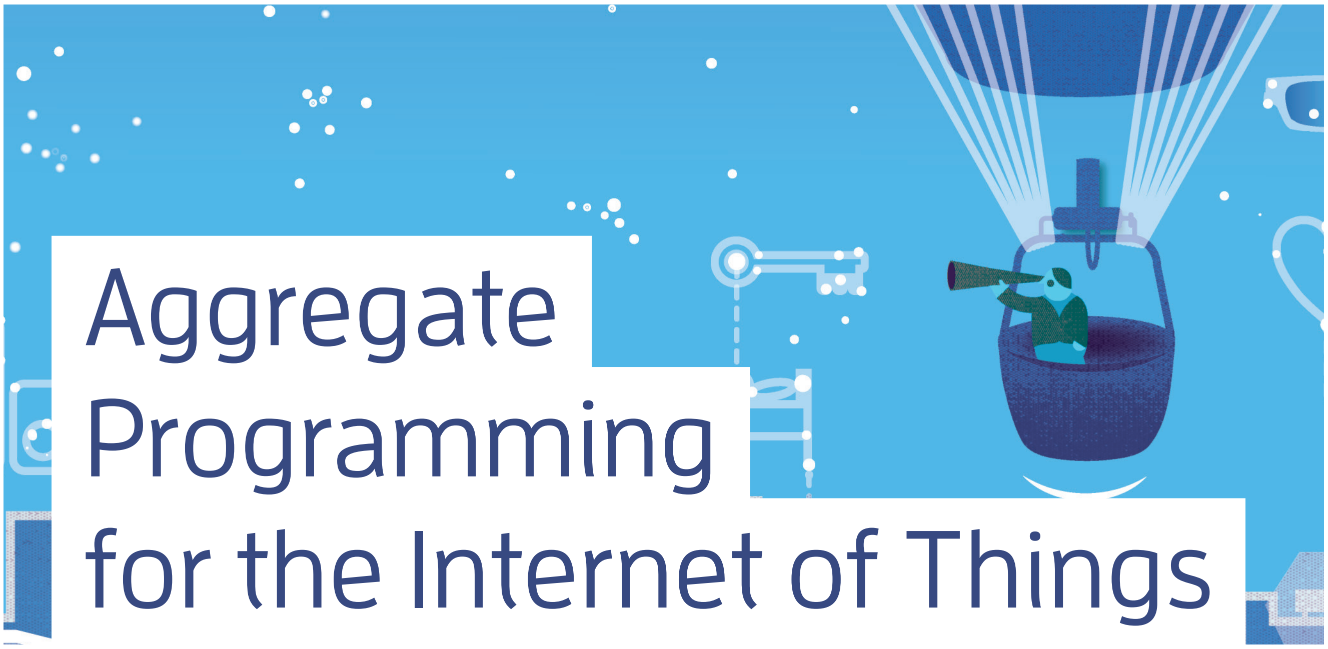

Aggregate Programming for the Internet of Things

Jacob Beal, Raytheon BBN Technologies
Danilo Pianini and Mirko Viroli, University of Bologna

Through field calculus constructs and building-block APIs, aggregate programming could help unlock the IoT's true potential by allowing complex distributed services to be specified succinctly and by enabling such services to be safely encapsulated, modulated, and composed with one another.

The Internet of Things (IoT) is ushering in a dramatic increase in the number and variety of networked objects. Personal smart devices, vehicular control systems, intelligent public displays, drones, electronic tags, and all types of sensors pervade our everyday working and living environments. As Figure 1 shows, proximity-based interactions between neighboring devices play a major role in IoT visions, whether intermediated by fixed networks[1] or using peer-to-peer communications,[2] which lower latency and increase resilience to inadequate infrastructure during, for example, 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 inform development tools and methodologies, causing many aspects of device interaction—efficient and reliable communication, robust coordination, composition of capabilities, search for appropriate cooperating peers, and so on—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 test and maintenance issues.

Aggregate programming provides an alternative that dramatically simplifies the design, creation, and maintenance of complex IoT software systems. With this technique, the basic unit of computing is no longer a single device but instead a cooperating collection of devices: details of their behavior, position, and number are largely abstracted away, replaced with a space-filling computational environment. Hence, the IoT paradigm of many heterogeneous devices becomes less a concern and more an opportunity to increase the quality—for example, soundness, stability, and efficacy—of application services. This is accomplished through a layered approach to programming complex services that builds on foundational work on the composition of distributed systems as well as on general mechanisms to provide robust and adaptive coordination, ultimately providing 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 large public gatherings. In these environments, opportunistic interactions between devices such as people's smartphones can smoothly support services including crowd detection, dispersal advice, and crowd-aware navigation. To illustrate the power of aggregate computing, we provide examples of how these crowd safety services can be implemented and composed, empirically demonstrating the resulting services' resilience and adaptivity using data gathered from an actual mass public event.

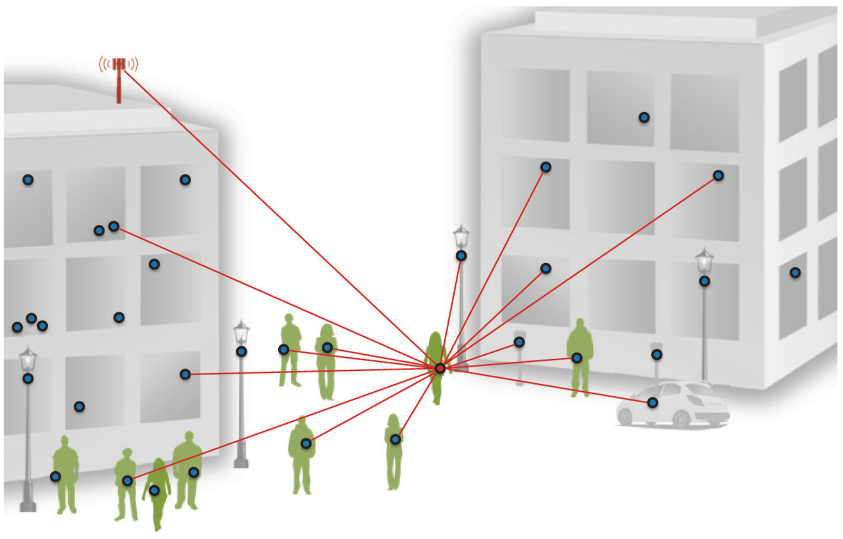

FIGURE 1. In a world filled with smart networked objects, every device has the opportunity to wirelessly interact with other nearby devices, both mobile and stationary. Some of these interactions exploit fixed network infrastructure, but the vast majority involve a heterogeneous mixture of peers.

AGGREGATE PROGRAMMING

The widely recognized single-device viewpoint's limits have motivated work on aggregate programming in many domains.[3] Generally, the main strategies are making device interaction implicit (for example, TOTA[4]), composing geometric and topological constructions (for example, the Origami Shape Language[5]), automatically splitting computations for cloud-style execution (for example, MapReduce[6]), summarizing data over space–time regions and streaming it to other regions (for example, TinyDB[7]), and providing generalizable constructs for space–time computing (for example, Protelis[8]). The last two approaches are particularly well suited for the IoT, as they are explicitly designed for distributed operation in a physical environment with embedded devices.

The successes and pitfalls of these many prior efforts suggest some key observations about programming large-scale situated systems. First, mechanisms for robust coordination should be hidden "under the hood," where programmers are not required to interact with them. Second, composition of modules and subsystems must be simple and transparent. Third, different subsystems need different coordination mechanisms for different regions and times.

Aggregate programming aims to address these issues using the following three principles:

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

To illustrate the advantages of aggregate versus device-centric programming, consider a service that leverages interactions among users' smartphones to estimate crowd density and distribution. One component service warns people of nearby regions where there is risk of panic or trampling, another provides advice on dispersing from such regions, and a third helps users navigate through the crowd while avoiding dangerous areas.

As Figure 2a shows, a device-centric programmer must focus on the protocol for device interactions while simultaneously reasoning about how local interactions will produce the desired complex global behavior. In contrast, as Figure 2b shows, an aggregate programmer naturally reasons in terms of incremental construction from continuum-like data structures and services. Crowd estimation outputs a distributed data structure—a computational field[4,9]—mapping from

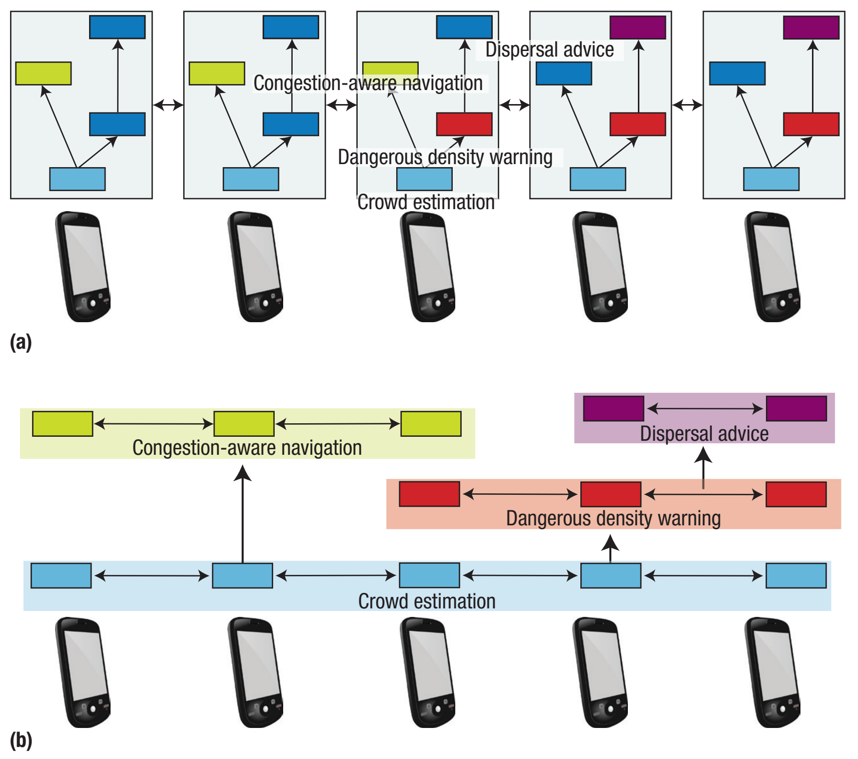
Dispersal advice
Congestion-aware navigation
Dangerous density warning
Crowd estimation
(a)
Congestion-aware navigation
Dispersal advice
Dangerous density warning
Crowd estimation
(b)

FIGURE 2. Two approaches to programming for the Internet of Things (IoT)—in this case, a smartphone-hosted crowd safety service. (a) Traditional device-centric programming of distributed algorithms. (b) Aggregate programming, which enables algorithmic building blocks to be scoped and composed directly for the aggregate.

location to crowd density. This serves as an input for crowd-aware navigation, which outputs vectors of recommended travel, and for the warning function, 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 protocol specifics can be generated automatically. By thus separating service composition from details of coordination and interaction protocols, aggregate programming promotes the construction of more complex, reusable, and composable distributed services.

TOWARD AGGREGATE APIs

Aggregate programming hides the complexity of distributed coordination in IoT network environments using several layers of abstraction, as Figure 3 shows. The foundation of aggregate programming is *field calculus*,[9] a core set of constructs modeling computation and interaction among large numbers of spatially embedded devices (in particular, we use 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, and then combine these to produce APIs for common application needs like sensing, decision, and action, creating a collective behavior API for transparent implementation of complex networked services and applications.[10]

This framework enables the simple specification of complex, resilient distributed systems. As this 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.

Field calculus constructs

Certain interaction patterns appear across many 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 nonaggregate 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 produces a field of ambient temperatures, smartphone accelerometers produces a field of movement directions, and a notification application produces a field of messages displayed on phones.

Fields are built and manipulated using four program constructs:

- Functions—`b(e1,...e`$_n$`)` applies function `b` to arguments `e1...e`$_n$. Such "built-in" functions are stateless mathematical, logical, or algorithmic functions, sensors or actuators, or user-defined or imported library methods.
- *Dynamics*—`rep(x<-v) {s`$_1$`;...;s`$_n$`}` defines a local state variable `x` initialized with value `v` and periodically updated with the result of executing its body statements `{s`$_1$`;...;s`$_n$`}`, thereby defining a field that evolves over time.
- *Interaction*—`nbr(s)` gathers a map at each device (actually, a field) from all neighbors as well

as from itself to the latest value of s. Built-in "under the hood" functions then summarize such maps—for example, minHood(m) finds the minimum value in map m.

- Restriction—if(e){s1;...;s_n} else {s1';...;s_m'} partitions the network into two regions: where e is true, s1;...;s_n is computed; elsewhere, s1';...;s_m' is computed instead. Importantly, partition implies that branches are encapsulated and cannot have effects outside their subspace.

Each construct can be interpreted as aggregate-level field manipulation or used by protocols for individual devices implementing such manipulations. Field calculus is also universal,[11] supporting any causal, approximable space–time computation. As we will see, field calculus can express distributed services that are safely and predictably composed and modulated.

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

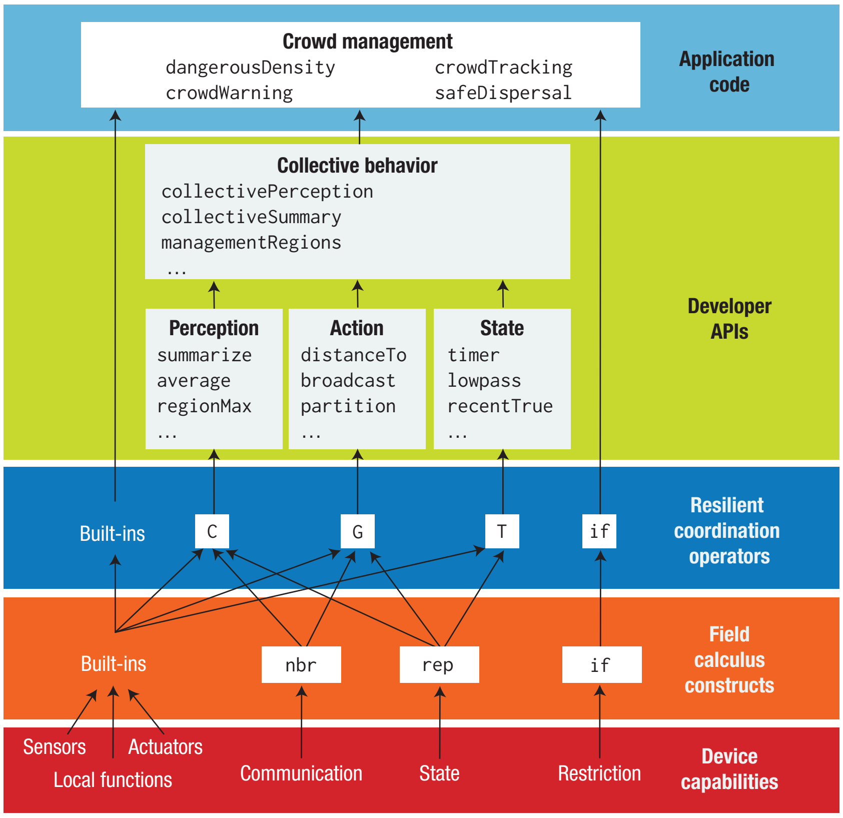

FIGURE 3. Aggregate programming abstraction layers. The software and hardware capabilities of particular devices are used to implement aggregate-level field calculus constructs. These constructs are used to implement a limited set of building-block coordination operations with provable resilience properties, which are then wrapped and combined together to produce a user-friendly API for developing situated IoT systems.

Building blocks for resilient coordination

The next level of abstraction in the aggregate programming framework adds resilience, identifying a collection of general building-block operators for resilient coordination applications. This layer (middle in Figure 3) consists of coordination mechanisms that are self-stabilizing, meaning they reactively adjust to changes in network structure or input values; are scalable to large networks; and 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 of building blocks[10] contains 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 that executes two tasks: it first 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.

```
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)
}
```

FIGURE 4. Protelis implementation of operator G.

- `T(initial, floor, decay)`—flexible countdown with a potentially time-varying rate: the 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 (see, for example, 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 that satisfies the same resilience properties as those already in the collection.

Pragmatic general-purpose APIs

To better meet day-to-day programming needs, libraries developed using building-block operators can apply and combine the operators to form a pragmatic, user-friendly API that 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 estimates 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 that cross 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 throughput 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.

These developer APIs form a practical interface for a typical engineer to develop IoT services using distributed coordination. Building APIs atop resilient operators and field calculus constructs ensures that these services are also resilient and safely composable. In parallel, development at lower layers of the framework can improve and extend available coordination mechanisms, improve efficiency of field calculus abstractions, and improve interface efficacy with particular device hardware or with nonaggregate 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.

EXAMPLE: ENGINEERING LARGE-SCALE, OPPORTUNISTIC CROWD SAFETY SERVICES

Field calculus and reusable building-block APIs can greatly simplify the construction and composition of resilient applications for IoT scenarios.

On one hand, individual distributed services can be built simply by composing 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, similar to how threads and virtualization enable the composition and modulation of services in individual machines and datacenters.

For example, consider how an IoT environment might provide crowd safety services at mass public events such as civic festivals, outdoor concerts, or marathons. The high concentration of people in constrained areas often creates emergent zones of dangerous overcrowding where even a small incident can create a panic or stampede that leads to injuries or deaths.[12,13] Moreover, the large number of people and spatial extent often overwhelm the available local infrastructure: cellphone networks drop calls, data communications become unreliable, public safety personnel cannot quickly attend to every incident, and so on. In an IoT environment, however, more people means more personal smart devices, which might coordinate with one another and other IoT devices embedded in their environment, requiring neither cloud services nor centrally deployed infrastructure.

Figure 5a shows an Alchemist[14] simulation of such a crowd safety service running on 1,000 embedded stationary devices plus 1,479 mobile personal devices, each following a smartphone position trace collected at the 2013 Vienna City Marathon.[15] The devices

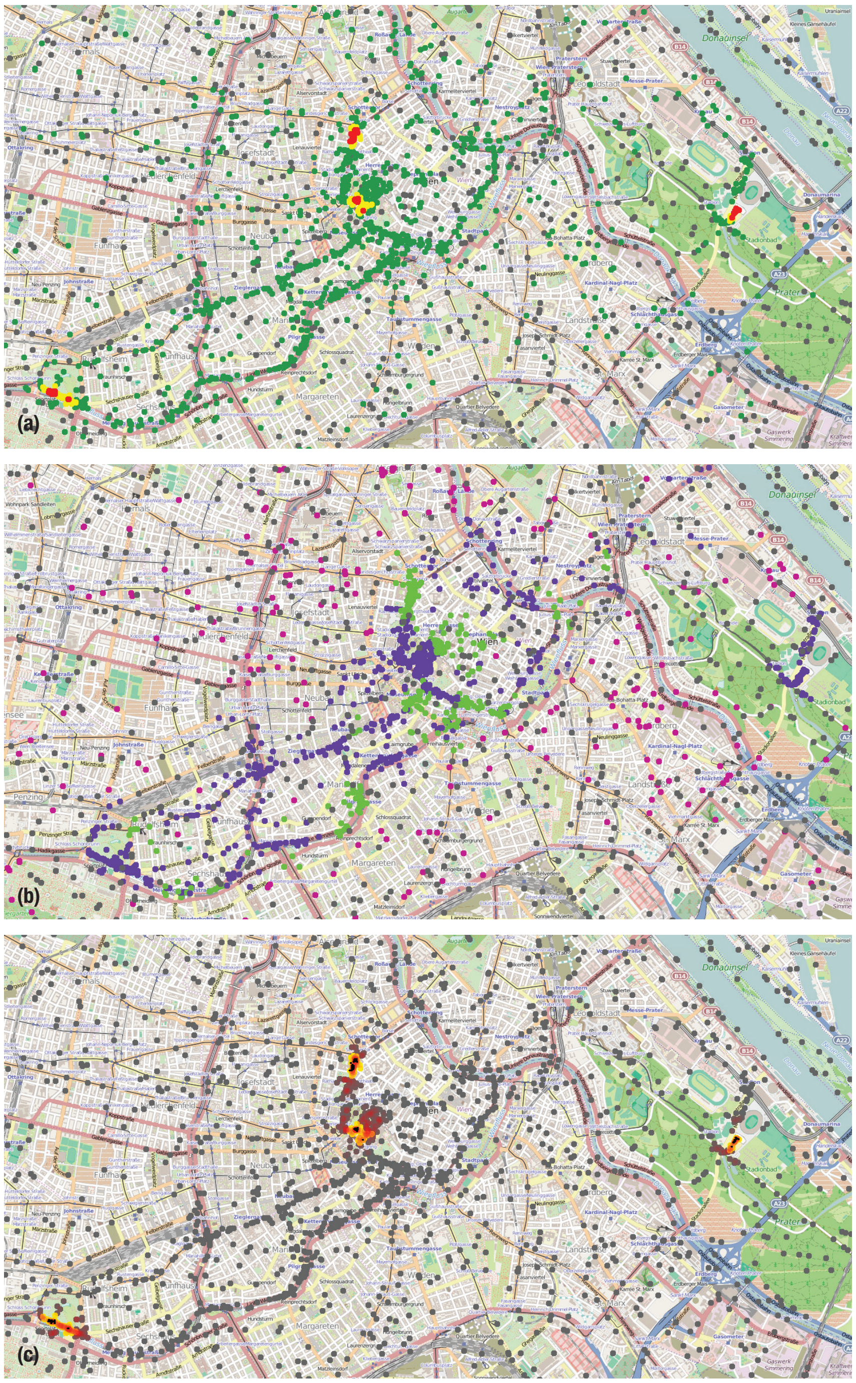

FIGURE 5. Snapshots from an Alchemist simulation of a crowd safety service in an IoT environment on approximately 2,500 personal and embedded devices. (a) Restricted to run on personal devices (colored), the service detects regions of dangerous crowd density (red) and disseminates warnings to nearby devices (yellow). (b) Nondisruptive upgrade of the running service disseminates replacement code from injection points: devices running the old version only (pink) 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 (hotter colors are higher priority) near potential emergency situations for all devices, not just those running the service. Note the correspondence of hot regions in (c) with those in (a).

```
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
}
```

FIGURE 6. Protelis implementation of crowd estimation and warning dissemination.

communicate via once-per-second asynchronous local broadcasts with a range of 100 meters.[13,16]

Our 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 a crowd and LoS E (>2.17 people/m^2) in a group of at least 300 people indicating potentially dangerous density. Density is estimated as $\rho = |nbrs|/p\pi r^2 w$, where $|nbrs|$ counts neighbors within range r, p estimates the proportion of people with a device running the app (about 0.5 percent of marathon attendees), and w estimates the fraction of walkable space in the local urban environment.

Given this estimate, potential crowding danger can be detected and warnings 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 as shown in Figure 6. This short program is resilient and adaptive, enabling it to effectively estimate crowding (`none`, `low`, `high`) and distribute warnings while executing on numerous mobile devices. Figures 7a and 7b compare the number of crowded and warned devices against ideal values across a 15-minute simulation: crowding level estimations track very closely to the true level of crowding, 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 the unanticipated composition of processes, a critical need in IoT environments. With our approach, a complex distributed service can be encapsulated and managed as a single aggregate object that can be modulated and composed with other services.[9] For example, crowd estimation can be wrapped with another service for nondisruptive distributed upgrades, which spreads a new version peer to peer from one or more devices where it is injected. The new version runs alongside the old version for some period of time, each safely encapsulated using field calculus's restriction and alignment semantics, and switching over when some criterion is met—for example, a specified elapsed time. Figure 5b shows such an upgrade in progress. This allows a switchover from one version to another without disrupting services, as Figure 7c shows, and without building any upgrade capability into the services.

Similarly, encapsulation allows management of service composition with dynamically specified policies. For example, Figure 5c shows the effect of a policy prioritizing crowd safety services near dangerously crowded 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 the engineering and complex coordination of services in an open and dynamic IoT environment.

Through field calculus constructs and building-block APIs, aggregate programming could help unlock the IoT's true potential by allowing complex distributed services to be specified succinctly, as well as by enabling such services 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 that take advantage of the complementary capabilities of aggregate programming and cloud-based architectures are needed. Security must also be considered, as IoT environments are open and dynamic, and involve many actors with different motivations and capabilities. In addition, building-block APIs need to be further developed against real applications to ensure that they support a sufficiently wide range of IoT services. Finally, mechanisms for composition and modulation must be expanded into a general IoT "operating system," including support on various devices and encapsulation methods created for integrating legacy applications. Together, these could lead toward a future where complex distributed systems are just as simple to engineer as individual computers. **C**

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 US 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 US government. Approved for public release; distribution is unlimited.

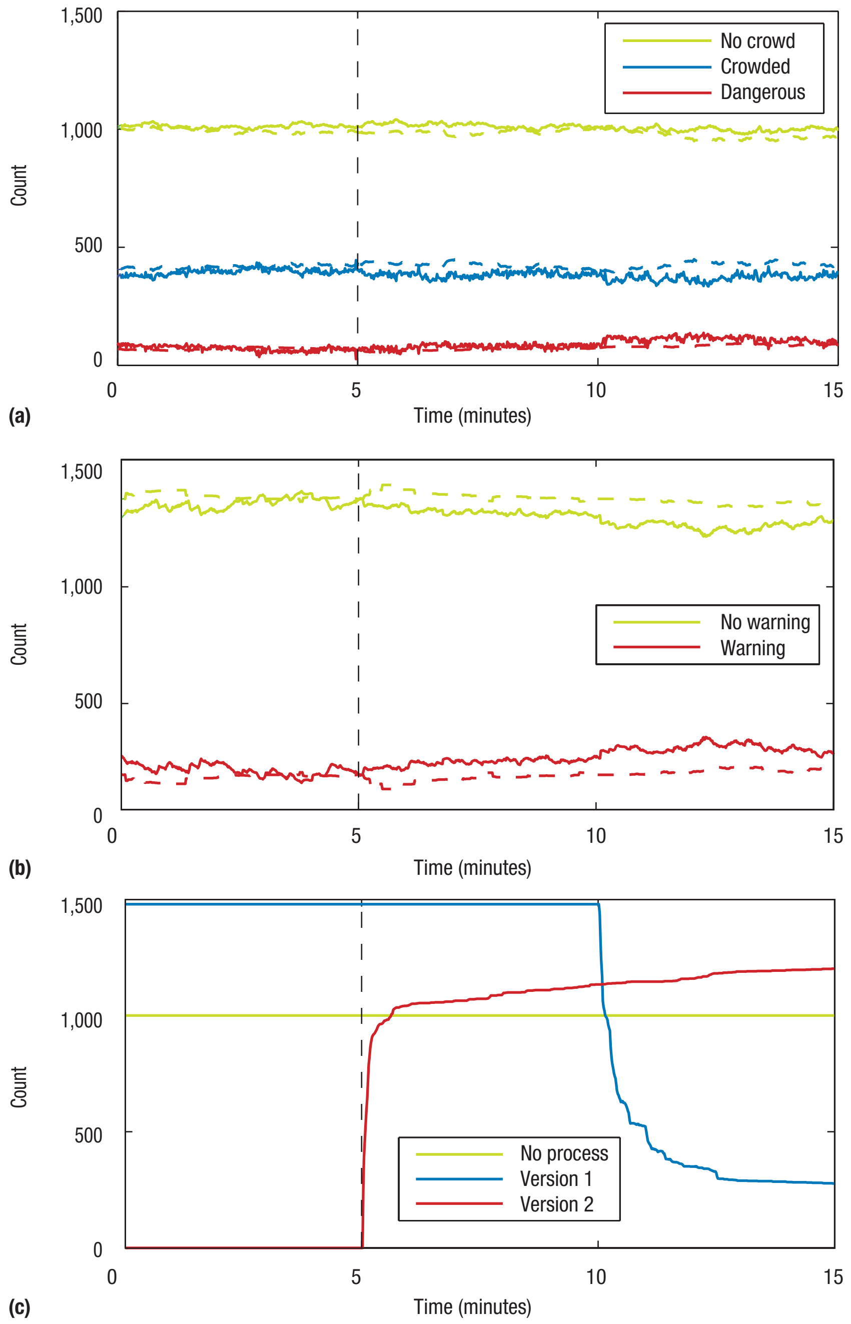

FIGURE 7. Aggregate programming enables lightweight construction of resilient IoT services, such as (a) distributed crowd estimation and (b) warnings of dangerous crowding. Despite executing on numerous mobile devices, both services produce estimates (solid lines) that track closely to the true values (dashed lines). (c) Aggregate-level manipulation of distributed services further enables disruption-free upgrades of these services while running: in this simulation, new versions of the services are injected at the 5-minute mark (vertical dashed line), but both versions are encapsulated to run concurrently until the new version is ready to take over smoothly from the old one, thereby ensuring no significant disruption in either service.

REFERENCES

1. L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Computer Networks*, vol. 54, no. 15, 2010, pp. 2787–2805.
2. D. Miorandi et al., "Internet of Things: Vision, Applications and Research Challenges," *Ad Hoc Networks*, vol. 10, no. 7, 2012, pp. 1497–1516.
3. J. Beal et al., "Organizing the Aggregate: Languages for Spatial Computing," *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, M. Mernik, ed., IGI Global, 2013, pp. 436–501.
4. M. Mamei and F. Zambonelli, "Programming Pervasive and Mobile Computing Applications: The TOTA

ABOUT THE AUTHORS

JACOB BEAL is a scientist at Raytheon BBN Technologies in Cambridge, Massachusetts. His research focuses on the engineering of robust adaptive systems, 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. Beal received a PhD in electrical engineering and computer science from MIT. He is an associate editor of *ACM Transactions on Autonomous and Adaptive Systems*, is on the steering committee of the IEEE International Conference on Self-Adapting and Self-Organizing Systems (SASO), and is a founder of the Spatial Computing Workshop series. He is a Senior Member of IEEE. Contact him at jakebeal@bbn.com.

DANILO PIANINI is a postdoctoral researcher in the Department of Computer Science and Engineering at the University of Bologna, Italy. He conducts research on pervasive computing and self-organization, with a focus on engineering, tools, and simulation. Pianini is the chief architect of the Alchemist simulator and one of the maintainers of Protelis. Contact him at danilo.pianini@unibo.it.

MIRKO VIROLI is an associate professor in the Department of Computer Science and Engineering at the University of Bologna. His research focuses on programming languages, computational models, and engineering of self-adaptive and self-organizing systems. Viroli received a PhD in computer science and engineering from the University of Bologna. He is on the editorial board of *The Knowledge Engineering Review* and is a member of IEEE and ACM. Contact him at mirko.viroli@unibo.it.

Approach," *ACM Trans. Software Eng. Methodology*, vol. 18, no. 4, 2009, pp. 1–56.
5. R. Nagpal, "Programmable Self-Assembly: Constructing Global Shape Using Biologically-Inspired Local Interactions and Origami Mathematics," PhD dissertation, Dept. of Electrical Eng. and Computer Science, MIT, 2001.
6. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, vol. 51, no. 1, 2008, pp. 107–113.
7. S. Madden et al., "Supporting Aggregate Queries over Ad-hoc Wireless Sensor Networks," *Proc. 4th IEEE Workshop Mobile Computing and Systems Applications* (WMCSA 02), 2002, pp. 49–58.
8. D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical Aggregate Programming," *Proc. 30th Ann. ACM Symp. Applied Computing* (SAC 15), 2015, pp. 1846–1853.
9. F. Damiani et al., "Code Mobility Meets Self-Organisation: A Higher-Order Calculus of Computational Fields," *Formal Techniques for Distributed Objects, Components, and Systems*, S. Graf and M. Viswanathan, eds., LNCS 9039, Springer, 2015, pp. 113–128.
10. J. Beal and M. Viroli, "Building Blocks for Aggregate Programming of Self-Organising Applications," *Proc. 2nd Workshop Fundamentals of Collective Adaptive Systems* (FoCAS 14), 2014, pp. 8–13.
11. J. Beal, M. Viroli, and F. Damiani, "Towards a Unified Model of Spatial Computing," *Proc. 7th Spatial Computing Workshop* (SCW 14), 2014; www.spatial-computing.org/_media/scw14/scw2014_p5.pdf.
12. G.K. Still, *Introduction to Crowd Science*, CRC Press, 2014.
13. B. Anzengruber et al., "Predicting Social Density in Mass Events to Prevent Crowd Disasters," *Social Informatics*, A. Jatowt et al., eds., LNCS 8238, Springer, 2013, pp. 206–215.
14. D. Pianini, S. Montagna, and M. Viroli, "Chemical-Oriented Simulation of Computational Systems with Alchemist," *J. Simulation*, vol. 7, no. 3, 2013, pp. 202–215.
15. F. Zambonelli et al., "Developing Pervasive Multi-agent Systems with Nature-Inspired Coordination," *Pervasive and Mobile Computing*, vol. 17, part B, 2015, pp. 236–252.
16. D. Pianini et al., "HPC from a Self-Organisation Perspective: The Case of Crowd Steering at the Urban Scale," *Proc. 2014 Int'l Conf. High Performance Computing & Simulation* (HPCS 14), 2014, pp. 460–467.
17. J. Fruin, *Pedestrian and Planning Design*, Metropolitan Assoc. of Urban Designers and Environmental Planners, 1971.