Lifting the Fog with Aggregate Computing
..a programming model perspective

Mirko Viroli

Alma Mater Studiorum—Università di Bologna, Italy
mirko.viroli@unibo.it

“Through the Fog” Workshop
Pisa, 15/1/2016
(Internet-of-)Things are getting a bit messy (and foggy)

A plethora of programming models for “mobile/IoT applications”

- **client side**
  - single-device program: objects + functions + concurrency...
    ..threads/actors/futures/tasks/activities
  - device-centric interactions/protocols: using APIs for MoM/SOA/ad-hoc-communications

- **server side**
  - same interactions/protocols: MoM/SOA/ad-hoc-communications
  - storage by DB: OO, relational, NoSQL
  - coordination (orchestration, mediation, rules enactment)
  - situation recognition (online/offline, mining, business intelligence, stream processing)

- **scalability in the server calls for cloudification**
  - not really orthogonal to the whole programming model
  - it often dramatically affects system design

Fog computing has likely nice benefits

..but does not seemingly simplify things
A plethora of programming models for “mobile/IoT applications”

- **client side**
  - single-device program: objects + functions + concurrency...
    - threads/actors/futures/tasks/activities
  - device-centric interactions/protocols: using APIs for MoM/SOA/ad-hoc-communications

- **server side**
  - same interactions/protocols: MoM/SOA/ad-hoc-communications
  - storage by DB: OO, relational, NoSQL
  - coordination (orchestration, mediation, rules enactment)
  - situation recognition (online/offline, mining, business intelligence, stream processing)

- scalability in the server calls for cloudification
  - not really orthogonal to the whole programming model
  - it often dramatically affects system design

---

Fog computing has likely nice benefits
..but does not seemingly simplify things
Implications

Where programming effort ends up?
- programs of clients and servers highly depend on
  - the chosen platform / API / communication technology
  - the number and type of involved devices
- IoT systems tend to be very rigid, hard and costly to debug/maintain
- design and deployments hardly tolerate changes

The technological result
- systems can’t scale with complexity of behaviour
- very few of the opportunities of large-scale IoT are taken
  - virtually any computational mechanism (sensing, actuation, processing, storage)...
  - ..could involve spontaneous, adaptive cooperation of large sets of devices!
- how many large-scale deployments of adaptive IoT systems around?
- where are the Collective Adaptive Systems?
Implications

Where programming effort ends up?
- programs of clients and servers highly depend on
  - the chosen platform / API / communication technology
  - the number and type of involved devices
- IoT systems tend to be very rigid, hard and costly to debug/maintain
- design and deployments hardly tolerate changes

The technological result
- systems can’t scale with complexity of behaviour
- very few of the opportunities of large-scale IoT are taken
  - virtually any computational mechanism (sensing, actuation, processing, storage)...
  - ..could involve spontaneous, adaptive cooperation of large sets of devices!
- how many large-scale deployments of adaptive IoT systems around?
- where are the Collective Adaptive Systems?
What to do? A programming model perspective..

What do we lack in large-scale IoT systems?
- the plain old platform-independent programming abstraction
  ⇒ fully grounding system design like objects did well... in the past
  ▶ delegating to the underlying platform virtually all deployment issues
  ▶ automagically addressing non-functional issues (resilience, self-*)

The challenge
Just directly consider the worst scenario possible..
- zillion devices unpredictably moving in the environment
- heterogeneous displacement, pervasive sensing/actuation
- abstracting away from the possible multi-layered “server system”
  (fog++/cloud++) in background
⇒ but be ready to exploit the opportunities it creates!
What to do? A programming model perspective..

What do we lack in large-scale IoT systems?
- the plain old platform-independent programming abstraction
  ⇒ fully grounding system design like objects did well... in the past
  ▶ delegating to the underlying platform virtually all deployment issues
  ▶ automagically addressing non-functional issues (resilience, self-*)

The challenge
Just directly consider the worst scenario possible..
- zillion devices unpredictably moving in the environment
- heterogeneous displacement, pervasive sensing/actuation
- abstracting away from the possible multi-layered “server system”
  (fog++/cloud++) in background
  ⇒ but be ready to exploit the opportunities it creates!
What do we lack in large-scale IoT systems?

- the plain old platform-independent programming abstraction
  ⇒ fully grounding system design like objects did well.. in the past
  ▶ delegating to the underlying platform virtually all deployment issues
  ▶ automagically addressing non-functional issues (resilience, self-*)

The challenge

Just directly consider the worst scenario possible..

- zillion devices unpredictably moving in the environment
- heterogeneous displacement, pervasive sensing/actuation
- abstracting away from the possible multi-layered “server system” (fog++/cloud++) in background
  ⇒ but be ready to exploit the opportunities it creates!

Let’s try to program *that* “computational system”!
Abstract of the talk

**Systems of interest: collective adaptive situated systems CASS**

- (possibly very large scale) collective adaptive systems
- deployed in physical space (situated), i.e., IoT-oriented
- complex (open, dynamic, in need of much self-*)

**Aggregate Computing**

- The “good” computing/programming model for CASS
- It gives nice abstractions, promoting solid engineering principles
- Simple idea, few constructs, rather tractable, somehow different

**This talk**

1. Motivation and idea of aggregate computing
2. Some semi-technicalities and overview of results
3. State of toolchain and perspectives on platforms and fog
Abstract of the talk

Systems of interest: collective adaptive situated systems CASS

- (possibly very large scale) collective adaptive systems
- deployed in physical space (situated), i.e., IoT-oriented
- complex (open, dynamic, in need of much self-*)

Aggregate Computing

- The “good” computing/programming model for CASS
- It gives nice abstractions, promoting solid engineering principles
- Simple idea, few constructs, rather tractable, somehow different

This talk

1. Motivation and idea of aggregate computing
2. Some semi-technicalities and overview of results
3. State of toolchain and perspectives on platforms and fog
Abstract of the talk

Systems of interest: collective adaptive situated systems CASS
- (possibly very large scale) collective adaptive systems
- deployed in physical space (situated), i.e., IoT-oriented
- complex (open, dynamic, in need of much self-*)

Aggregate Computing
- The “good” computing/programming model for CASS
- It gives nice abstractions, promoting solid engineering principles
- Simple idea, few constructs, rather tractable, somehow different

This talk
1. Motivation and idea of aggregate computing
2. Some semi-technicalities and overview of results
3. State of toolchain and perspectives on platforms and fog
Outline

1. Aggregate Computing
2. Field Calculus
3. Platform support
4. Field Engineering
Outline

1. Aggregate Computing
2. Field Calculus
3. Platform support
4. Field Engineering
An example opportunity for IoT-based CASS.
Gathering local context
Sensing global patterns of data
Crowd Detection
Crowd-aware Steering
Crowd dispersal
Crowd evacuation upon alerts
Broad research challenges

Computational/programming model for these services
- Programming as: “describing the problem, not hacking the solution!”
- Hiding complexity and resiliency “under-the-hood”
- How computation carries on is hidden as well, and intrinsically self-*

Grounding an effective tool-chain
- languages, compilers, simulators, scalable execution platforms

Supporting solid engineering principles
- checking/enacting functional/non-functional correctness
- supporting reuse of patterns, substitutability, compositionality

Chasing the true issue
- we should fully escape the single “device” abstraction
Broad research challenges

Computational/programming model for these services
- Programming as: “describing the problem, not hacking the solution!”
- Hiding complexity and resiliency “under-the-hood”
- How computation carries on is hidden as well, and intrinsically self-*

Grounding an effective tool-chain
- languages, compilers, simulators, scalable execution platforms

Supporting solid engineering principles
- checking/enacting functional/non-functional correctness
- supporting reuse of patterns, substitutability, compositionality

Chasing the true issue
- we should fully escape the single “device” abstraction
Broad research challenges

Computational/programming model for these services
- Programming as: “describing the problem, not hacking the solution!”
- Hiding complexity and resiliency “under-the-hood”
- How computation carries on is hidden as well, and intrinsically self-*

Grounding an effective tool-chain
- languages, compilers, simulators, scalable execution platforms

Supporting solid engineering principles
- checking/enacting functional/non-functional correctness
- supporting reuse of patterns, substitutability, compositionality

Chasing the true issue
- we should **fully** escape the single “device” abstraction
Approaches to “group interaction in space”

Survey of past approaches [Beal et.al., 2013]

- **Device abstractions** – make interaction implicit
  NetLogo, Hood, TOTA, Gro, MPI, and the SAPERE approach
- **Pattern languages** – supporting composability of spatial behaviour
  Growing Point, Origami Shape, various selforg pattern langs
- **Information movement** – gathering in space, moving elsewhere
  TinyDB and Regiment
- **Foundation** – giving linguistic means for group interactions in space
  $3\pi$, Shape Calculus, bi-graphs, KLAIM, $\sigma\tau$-linda, SCEL
- **Spatial computing** – program space-time behaviour of systems
  Proto, MGS

Our approach

- Combining the above efforts of “macro” programming
- Taking some of those ideas to the extreme consequences
Approaches to “group interaction in space”

Survey of past approaches [Beal et.al., 2013]

- **Device abstractions** – make interaction implicit
  NetLogo, Hood, TOTA, Gro, MPI, and the SAPERE approach
- **Pattern languages** – supporting composability of spatial behaviour
  Growing Point, Origami Shape, various selforg pattern langs
- **Information movement** – gathering in space, moving elsewhere
  TinyDB and Regiment
- **Foundation** – giving linguistic means for group interactions in space
  $3\pi$, Shape Calculus, bi-graphs, KLAIM, $\sigma T$-linda, SCEL
- **Spatial computing** – program space-time behaviour of systems
  Proto, MGS

Our approach

- Combining the above efforts of “macro” programming
- Taking some of those ideas to the extreme consequences
Aggregate programming at [IEEE Computer 48(9), 2015]

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, modularized, 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 permeate 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 or using peer-to-peer communications, which lower latency and increase resilience to inadequate infrastructure, 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 efficiency—of application
Manifesto of aggregate computing

**Motto:** program the aggregate, not individual devices!

1. The reference computing machine
   ⇒ an aggregate of devices as single “body”, fading to the actual space
2. The reference elaboration process
   ⇒ atomic manipulation of a collective data structure (a field)
3. The actual networked computation
   ⇒ a proximity-based self-org system hidden “under-the-hood”
Outline

1. Aggregate Computing
2. Field Calculus
3. Platform support
4. Field Engineering
Traditionally a map: \textit{Space} $\mapsto$ \textit{Values}

- possibly: evolving over time, dynamically injected, stabilising
- smoothly adapting to very heterogeneous domains
- more easily “understood” on continuous and flat spatial domains
- ranging to: booleans, reals, vectors, functions

real-valued gradient in 3D

numeric partition in 2D

boolean channel in 2D
A map: $DeviceSet \times Space \times Time \mapsto ValueSet$

- **event $E$**: a triple $\langle \delta, t, p \rangle$ – device $\delta$, “firing” at time $t$ in position $p$
- **domain $D$**: a coherent set of events (devices cannot move too fast)
- **field $\phi : D \mapsto V$**: a map from events to *field values*

Early intuition: often one will think at fields that..

- “converge” with density of events, and lose track of device identities
- eventually (in time) reach a fixpoint
- so, you can draw (and reason/design) in 2D
The “channel” example: computing a redundant route

How would you program it?

how could a program be platform-independent, unaware of global map, resilient to changes, faults,..
The “channel” example: computing a redundant route

How would you program it?

how could a program be platform-independent, unaware of global map, resilient to changes, faults,..
Functionally composing fields

- Inputs: sensor fields, Output: actuator field
- Computation is a pure function over fields (time embeds state!)

⇒ for this to be practical/expressive we need a good programming language
Field calculus [Damiani & Viroli & Beal & Pianini, FORTE2015]

Key idea

- a sort of λ-calculus with “everything is a field” philosophy!

Syntax (slightly refactored, semi-formal version of FORTE’s)

\[ e ::= x \mid v \mid e(e_1, \ldots, e_n) \mid \text{rep}(e_0)\{e\} \mid \text{nbr}\{e\} \]

\[ v ::= <\text{standard-values}> \mid \lambda \]

\[ \lambda ::= f \mid o \mid (\overline{x})\rightarrow e \]

\[ F ::= \text{def } f(\overline{x})\{e\} \]

Few explanations

- \( v \) includes numbers, booleans, strings,..
  ..tuples/vectors/maps/any-ADT (of expressions)
- \( f \) is a user-defined function
- \( o \) is a built-in functional operator (mostly pure math or a sensor)
Intuition of global-level semantics

The four main constructs at work
⇒ values, application, evolution, and interaction – in aggregate guise

- e ::= ... | v | e(e₁, ..., eₙ) | rep(e₀){e} | nbr{e}
### Intuition of field-level semantics

#### Value $v$
- A field constant in space and time, mapping any event to $v$

#### Function application $e(e_1, \ldots, e_n)$
- $e$ evaluates to a field of functions, assume it ranges to $\lambda_1, \ldots, \lambda_n$
- this naturally induces a partition of the domain $D_1, \ldots, D_n$
- now, join the fields: $\forall i, \lambda_i(e_1, \ldots, e_n)$ restricted in $D_i$

#### Repetition $\text{rep}(e_0)\{e_\lambda\}$
- the value of $e_0$ where the restricted domain “begins”
- elsewhere, unary function $e_\lambda$ is applied to previous value at each device

#### Neighbouring field construction $\text{nbr}\{e\}$
- at each event gathers most recent value of $e$ in neighbours (in restriction)
- ..what is neighbour is orthogonal (i.e., physical proximity)
The restriction trick: branching behaviour

if as a space-time branching construct

\[
\text{if}(\text{e-bool})\{\text{e-then}\}\text{else}\{\text{e-else}\}
\approx
(\text{e-bool} ? ()=>\{\text{e-then}\} : ()=>\{\text{e-else}\})(())
\]

More advanced patterns

- spread code, in different versions in different regions
- have different regions/device run different programs
Functionally composing fields

...so, is field calculus language practical/expressive?
The channel pattern

def gradient(source){  ;; reifying minimum distance from source
    rep(Infinity) {  ;; distance is infinity initially
        (distance) => source ? 0 : minHood( nbr{distance} + nbrRange )
    }
}

def distance(source, dest) {  ;; propagates minimum distance between source and dest
    snd(  ;; returning the second component of the pair
        rep(pair(Infinity, Infinity)) {  ;; computing a field of pairs (distance,value)
            (distanceValue) => source ? pair(0, gradient(dest)) :
                minHood(  ;; propagating as a gradient, using for first component of the pair
                    pair(fst(nbr{distanceValue}) + nbrRange, snd(nbr{distanceValue})))
        }
    )
}

def dilate(region, width) {  ;; a field of boolens
    gradient(region) < width
}

;; Here the “aggregate” nature of our approach gets revealed
def channel(source, dest, width) {
    dilate( gradient(source) + gradient(dest) <= distance(source,dest), width )
}
Symbols

Built-in functions exploited

- `?:` — Java-like (though, call-by-value) ternary operator
- `nbrRange` — maps each device to a neighbour field of estimated distances
- `minHood` — in each device, collapse a neighbour field into its minimum value
- `sumHood` — in each device, collapse a neighbour field into sum of values
- `*,-,*,/,>,...` — usual math, applied also pointwise to fields
- `pair,fst,snd` — construction/selection for pairs
Channel in action: note inherent self-stabilisation
On expressiveness of the field calculus

Practically, we can express:

- complex spreading / aggregation / decay functions
- spatial leader election, partitioning, consensus
- distributed spatio-temporal sensing and situation recognition
- dynamic deployment/spreading of code (via lambda)
- implicit/explicit device selection of what code execute
- “collective teams” forming based on the selected code
Outline

1. Aggregate Computing
2. Field Calculus
3. Platform support
4. Field Engineering
Key aspects of the semantics: network model

Platform abstract model

- A node state $\theta$ (value-tree) updated at asynchronous rounds
- At the end of the round, $\theta$ is made accessible to the neighbourhood
- A node state is updated “against” recently received neighbours’ trees

Nodes send their evaluation tree to neighbours at the end of each computation round.
Single-round operational semantics – pulverization

Main run-time structures

\[ \phi ::= \{ \delta \mapsto l \} \quad \text{field value: mapping nodes to local values} \]

\[ v ::= l \mid \phi \quad \text{values: local values or field values} \]

\[ \theta ::= v(\theta) \quad \text{value-tree: an ordered tree of values} \]

\[ \Theta ::= \{ \delta \mapsto \theta \} \quad \text{value-tree environment: neighbours info} \]

Big-step operational semantics judgment

\[ \delta; \Theta \vdash e \Downarrow \theta \]

Read: at device \( \delta \), with environment \( \Theta \), evaluation of \( e \) gives result \( \theta \)

\[ \Rightarrow \quad \text{Namely, computation takes input } \Theta \text{ and produces output } \theta \]

..an orthogonal “network-level” LTS completes the operational semantics
Current formalisation (under progressive shrinking..)

**Auxiliary functions:**

\[
\begin{align*}
\rho(\nu(\theta)) &= \nu \\
\pi_i(\nu(\theta_1, \ldots, \theta_n)) &= \theta_i & \text{if } 1 \leq i \leq n \\
\pi_i(\theta) &= \bullet & \text{otherwise} \\
\pi^{\ell,n}(\nu(\theta_1, \ldots, \theta_{n+2})) &= \theta_{n+2} & \text{if } \rho(\theta_{n+1}) = \ell \\
\pi^{\ell,n}(\theta) &= \bullet & \text{otherwise} \\
\end{align*}
\]

For \(\text{aux} \in \rho, \pi_i, \pi^{\ell,n}:\)

\[
\begin{align*}
\text{aux}(\delta \mapsto \theta) &= \delta \mapsto \text{aux}(\theta) & \text{if } \text{aux}(\theta) \neq \bullet \\
\text{aux}(\delta \mapsto \theta) &= \bullet & \text{if } \text{aux}(\theta) = \bullet \\
\text{aux}(\Theta, \Theta') &= \text{aux}(\Theta), \text{aux}(\Theta') \\
\end{align*}
\]

\[
\begin{align*}
\text{args}(d) &= \chi & \text{if def } d(\chi) \{ e \} \\
\text{body}(d) &= e & \text{if def } d(\chi) \{ e \} \\
\text{body}(d \Rightarrow e) &= e \\
\end{align*}
\]

**Rules for expression evaluation:**

\[
\begin{align*}
\frac{\delta; \Theta \Downarrow \ell}{\delta; \Theta \Downarrow \ell} & \quad (\text{E-LOC}) \\
\frac{\phi' = \phi |_{\text{dom}(\Theta) \cup \{ \delta \}}}{\delta; \Theta \Downarrow \phi'() \quad (\text{E-FLD})} \\
\frac{c(e_1, \ldots, e_m) \text{ not a value}}{\delta; \pi_1(\Theta) \Downarrow e_1 \, \downarrow \, \theta_1 \quad \ldots \quad \delta; \pi_m(\Theta) \Downarrow e_m \, \downarrow \, \theta_m \quad \ell = c(\rho(\theta_1), \ldots, \rho(\theta_m))}{\delta; \Theta \Downarrow c(e_1, \ldots, e_m) \Downarrow \ell(\theta_1, \ldots, \theta_n) \quad (\text{E-DATA})} \\
\frac{\delta; \pi_{n+1}(\Theta) \Downarrow e_{n+1} \, \downarrow \, \theta_{n+1} \quad \rho(\theta_{n+1}) = b}{\delta; \pi_1(\Theta) \Downarrow e_1 \, \downarrow \, \theta_1 \quad \ldots \quad \delta; \pi_n(\Theta) \Downarrow e_n \, \downarrow \, \theta_n \quad \ell = c(\rho(\theta_1), \ldots, \rho(\theta_n))}{\delta; \Theta \Downarrow e_{n+1}(e_1, \ldots, e_n) \Downarrow \ell(\theta_1, \ldots, \theta_{n+1}) \quad (\text{E-B-APP})} \\
\frac{\delta; \pi_{n+1}(\Theta) \Downarrow e_{n+1} \, \downarrow \, \theta_{n+1} \quad \rho(\theta_{n+1}) = \ell \quad \text{args}(\ell) = x_1, \ldots, x_n}{\delta; \pi_1(\Theta) \Downarrow e_1 \, \downarrow \, \theta_1 \quad \ldots \quad \delta; \pi_n(\Theta) \Downarrow e_n \, \downarrow \, \theta_n \quad \ell = c(\rho(\theta_1), \ldots, \rho(\theta_n))}{\delta; \Theta \Downarrow e_{n+1}(e_1, \ldots, e_n) \Downarrow \ell(\theta_1, \ldots, \theta_{n+1}) \quad (\text{E-B-APP})} \\
\hline
\end{align*}
\]

\[
\begin{align*}
\Theta_1 &= \pi_1(\Theta) \\
\delta; \Theta_1 \Downarrow e \, \downarrow \, \theta_1 \quad \phi = \rho(\Theta_1)[\delta \mapsto \rho(\theta_1)] \\
\delta; \Theta \Downarrow \text{nbr}(\{ e \}) \Downarrow \phi(\theta_1) \quad (\text{E-NBR}) \\
\end{align*}
\]

\[
\begin{align*}
\ell_0 &= \begin{cases}
\rho(\Theta(\delta)) & \text{if } \Theta \neq \emptyset \\
\ell & \text{otherwise}
\end{cases} \\
\delta; \pi_1(\Theta) \Downarrow e[x := \ell_0] \, \downarrow \, \theta_1 \quad \ell_1 = \rho(\theta_1) \quad (\text{E-REP})
\end{align*}
\]

\[
\delta; \Theta \Downarrow \text{rep}(\ell)(\chi) \Rightarrow e \Downarrow \ell_1(\theta_1)
\]

---

Mirko Viroli (Università di Bologna)
Core mechanisms in the operational semantics

Orthogonally..

- evaluation proceeds recursively on expression and neighbour trees
- neighbour trees may be discarded on-the-fly if not “aligned” (restriction)

Function application $e(e_1, \ldots, e_n)$

- evaluates body against a filtered set of neighbours..
- i.e., only those which evaluated $e$ to same result

Repetition $\text{rep}(e_0)\{e_\lambda\}$

- if a previous value-tree of mine is available, evaluates $e_\lambda$ on its root
- otherwise, evaluates $e_0$

Neighbouring field construction $\text{nbr}\{e\}$

- gather values from neighbour trees currently aligned
- add my current evaluation of $e$
Core mechanisms in the operational semantics

Orthogonally..
- evaluation proceeds recursively on expression and neighbour trees
- neighbour trees may be discarded on-the-fly if not “aligned” (restriction)

Function application $e(e_1, \ldots, e_n)$
- evaluates body against a filtered set of neighbours..
- i.e., only those which evaluated $e$ to same result

Repetition $\text{rep}(e_0)\{e_\lambda\}$
- if a previous value-tree of mine is available, evaluates $e_\lambda$ on its root
- otherwise, evaluates $e_0$

Neighbouring field construction $\text{nbr}\{e\}$
- gather values from neighbour trees currently aligned
- add my current evaluation of $e$
Operational semantics as blueprint for platform support

Requirements

- a notion of neighbourhood must be defined — wireless connectivity, physical proximity..
- nodes execute in asynchronous rounds, and emit a “round result”
- a node need to have recent round results of neighbours
- by construction we tolerate losses of messages
- by construction we tolerate various round frequencies

Platform details are very orthogonal to our programming model!

- the above requirements can be met by various platforms
- *programming remains mostly unaltered!*
Operational semantics as blueprint for platform support

Requirements

- A notion of neighbourhood must be defined — wireless connectivity, physical proximity.
- Nodes execute in asynchronous rounds, and emit a “round result”
- A node needs to have recent round results of neighbours
- By construction, we tolerate losses of messages
- By construction, we tolerate various round frequencies

Platform details are very orthogonal to our programming model!

- The above requirements can be met by various platforms
- Programming remains mostly unaltered!
Natural implementations

P2P
- devices see neighbours, and directly broadcast messages (ad-hoc wifi)
  ⇒ in principle possible, but interferences might be an issue

Server-mediated communication
- a single server mediates communications
- holding topology info and enacting a fully-custom topology
  ⇒ not hard to handle 10K devices firing at 1Hz
Dealing with (mobile) cloud

Cloud implementation
- we use devices only as physical containers of sensors / actuators
- the server as mediator of communications and running computations
- cloudification is easy due to our pulverization semantics
- a cloud-DB holds field maps, rounds can be executed in clusters

Advantages of the conceptual concentration
- vertical optimisation: decide what to compute in the cloud and what on device/edge
- horizontal optimisation: decide which device computation can be slowed down
  ⇒ both explicit (programmed) or implicit (dynamically activated)
Dealing with fog computing

### Explicit approach: edge devices as part of the “aggregate machine”
- edge devices are just like any other device
- the programmer takes care of using them for specific tasks
  - typically: leaders/aggregators of distributed sensing/decision making
  - they could be nodes with higher round frequency and connectivity

### Implicit approach: edge devices are part of the underlying platform
- using edge devices as sort of vertical optimisation
- when too much computation/communication resources are required, the platform starts delegating to the edges, then to cloud
Outline

1. Aggregate Computing
2. Field Calculus
3. Platform support
4. Field Engineering
How to scale with complexity?

Crowd Management

Application Code

Boiler-plate code

Field Calculus Constructs

Device Capabilities

fun-call

nbr
rep

local ops
- sensors
- actuators
- math

user-def funs
- higher-order
- code mobility
- abstraction

anonym funs
- higher-order
- code mobility
- restriction

commun. state

Mirko Viroli (Università di Bologna)
Survey of recent efforts

Attacking a multifaceted problem

- Properties (self-stabilisation, density-independence, universality)
- Tools (languages, simulators, platforms)
- Libraries (reusable components, correctness, raising abstraction)
Properties

Self-stabilisation
- **Def:** If environment and inputs stop changing, computation reaches a fixpoint
- Identified a rather large subset of the language [SASO-2015]

Density independence
- **Def:** the denotation of an expression computation converges with the space-time density of events
- Identified a (small) subset of the language [Submitted]

Universality
- **Def:** for any causal field evolution $\Phi$ over arbitrary domain $D$ (even continuous), there exists an expression whose denotation converges to $\Phi$ as the domain converges to $D$
- Field calculus is arguably universal [SCW-2014]
Self-stabilisation for computational fields

Definition of self-stabilising field expression $e$

- Given an environment: inputs (sensor fields) and network topology
  \[ \Rightarrow \] computing $e$ results in a stable unique field in finite time

Implications

- After fixing a topology, a field computation is an I/O problem
  \[ \Rightarrow \] Transient env. changes do not affect the result of computation

Self-stabilisation is undecidable, but can identify sufficient conditions
Self-stabilisation for computational fields

Definition of self-stabilising field expression $e$

- Given an environment: inputs (sensor fields) and network topology
  $\Rightarrow$ computing $e$ results in a stable unique field in finite time

Implications

- After fixing a topology, a field computation is an I/O problem
  $\Rightarrow$ Transient env. changes do not affect the result of computation

Self-stabilisation is undecidable, but can identify sufficient conditions
**Functions**

- **G**: Spreads and en-route computes information outwards a source
- **C**: Collects and en-route aggregates information inwards a destination
- **T**: Locally iterates computations until a termination

**Observations**

- The three blocks can pragmatically replace `nbr` and `rep`
- Towards a GCT-based system of libraries
Libraries (each function with a 1-5 lines body)

- **Application Code**
  - Crowd Management
    - evacuation-alert
    - crowd-warning
    - crowd-tracking
    - safe-dispersal
  - Collective Behavior
    - collective-perception
    - Collective-summary
    - management-regions
    - average
    - region-max
    - summary
    - distance-to
    - broadcast
    - partition
    - timer
    - lowpass
    - recent-true
  - Perception
    - fun-call
  - Action
    - fun-call
  - State
    - fun-call

- **Developer APIs**
  - anonymous funs
    - higher-order
    - code mobility
    - restriction
  - user-def funs
    - modularity
    - abstraction
  - commun.
  - state
  - local ops
    - sensors
    - actuators
    - math

- **Resilient Coordination Operators**
  - C
  - G
  - T
  - if

- **Field Calculus Constructs**
  - fun-call
  - nbr
  - rep

- **Device Capabilities**
Crowd estimation service, on top of APIs [Fruin, 1971]

```python
;; Density Estimation: density of neighbours within a short 3.0mt range
def densityEstimation() {
    countHood(nbrRange < 3.0) / (3.0 * 3.0 * 3.14)
}

;; More then 2.17 density and 'threshold' overcame in a 'partition' region
def dangerousDensity(partition, threshold, range) {
    average(partition, densityEstimation()) > 2.17 ;; Fruin LoS
    &&
    count(partition) > threshold ;; and, many people..
}

;; Crowd levels:
;; Level 1 (low): density greater than 1.08 in last 60 seconds
;; Level 2 (high): in a 30mt-range partition, L1 persons are > 300 with density > 2.18
;; Level 0 (none): others
def crowdTracking() {
    if (recentlyTrue(densityEstimation() > 1.08, 60) { ;; note restriction here..
        dangerousDensity(randomPartition(30), 300) ? high : low
    } else {
        none
    }
}
```
Current tool-chain for aggregate computing

- Libraries
- Platforms
  - Alchemist
  - Proto Sim
- Static analysis
- Interpreter
- Xtext parsing
- Protelis
- Proto Lang
- Properties
- Field Calculus

- Execution
- Simulation
- Language tools
- Programming Language
- Formal foundation
- Field calculus in disguised and full-blown version
- Java-like syntax and Java API integration

Alchemist simulator: [http://alchemist.apice.unibo.it/](http://alchemist.apice.unibo.it/)
- A general-purpose simulator with pluggable specification language
- XText/Eclipse integration
- Support from working with Maps, Traces, Paths, Movement models
Current/future investigations

Field calculus
- fields as processes, neighbours as ensembles, dealing with streams
- universality, relation with continuous space-time, self-stabilisation
- model checking with abstractions for large-scale systems

Language and programming
- Protelis released, and pluggable into Alchemist simulator
- Scala library support to be released soon

Platform level
- single-server general-purpose coordinator (RESTlets + RedisDB)
- cloud support (experiments with Apache Kafka & Storm)
## Conclusions

### Aggregate Computing
- A new paradigm for developing large-scale situated systems
- A bunch of results and tools emerged, many to come
- We’re always eager to find new collaborations!

### Messages for the fog people
- Evaluate our toolchain for location-aware applications
- Think at a fog support that does not impact programming
- Try to think at systems as aggregates, it is worthy!

### Acknowledgments
- Jacob Beal (BBN, USA)
- Ferruccio Damiani (UNITO)
- Danilo Pianini (UNIBO)
Aggregate programming for the Internet of Things.
IEEE Computer, 48(9) 2015.

Organizing the aggregate: Languages for spatial computing.

Towards a unified model of spatial computing.

Pedestrian Planning and Design.
Metropolitan Association of Urban Designers and Environmental Planners.

Programming pervasive and mobile computing applications: The tota approach.
ACM Transactions on Software Engineering and Methodologies, 18(4).

Practical aggregate programming with PROTELIS.
In ACM Symposium on Applied Computing (SAC 2015).
To appear.
   Code mobility meets self-organisation: a higher-order calculus of computational fields.
   In Formal Techniques for Distributed Objects, Components, and Systems, volume 9039 of LNCS, pages 113–128. Springer.

   Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields
   In IEEE Conference on Self-Adaptive and Self-Organising Systems.

   Flexible self-healing gradients