Aggregate Computing

Spatial Multiagent Systems and Aggregate Computing
New Directions for Spatial Computing

Mirko Viroli

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

PhD Course at DISI
Bologna, 17/2/2017

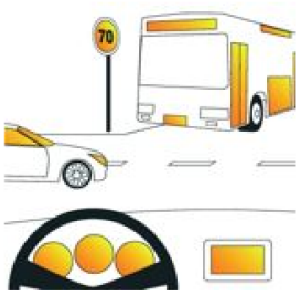
The IoT is becoming a crowded and complex place..

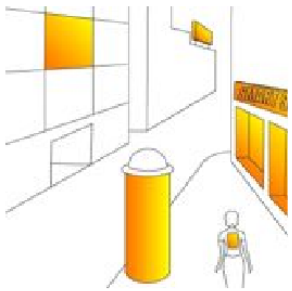
Future and emerging Internet-of-things are witnessing..

- increasing availability of wearable / mobile / embedded / flying devices
- increasing availability of heterogeneous wireless connectivity
- increasing availability of computational resources (device/edge/cloud)
- increasing production/analysis of data, everywhere, anytime

$\Rightarrow$ business / security / privacy concerns will probably be drivers, too



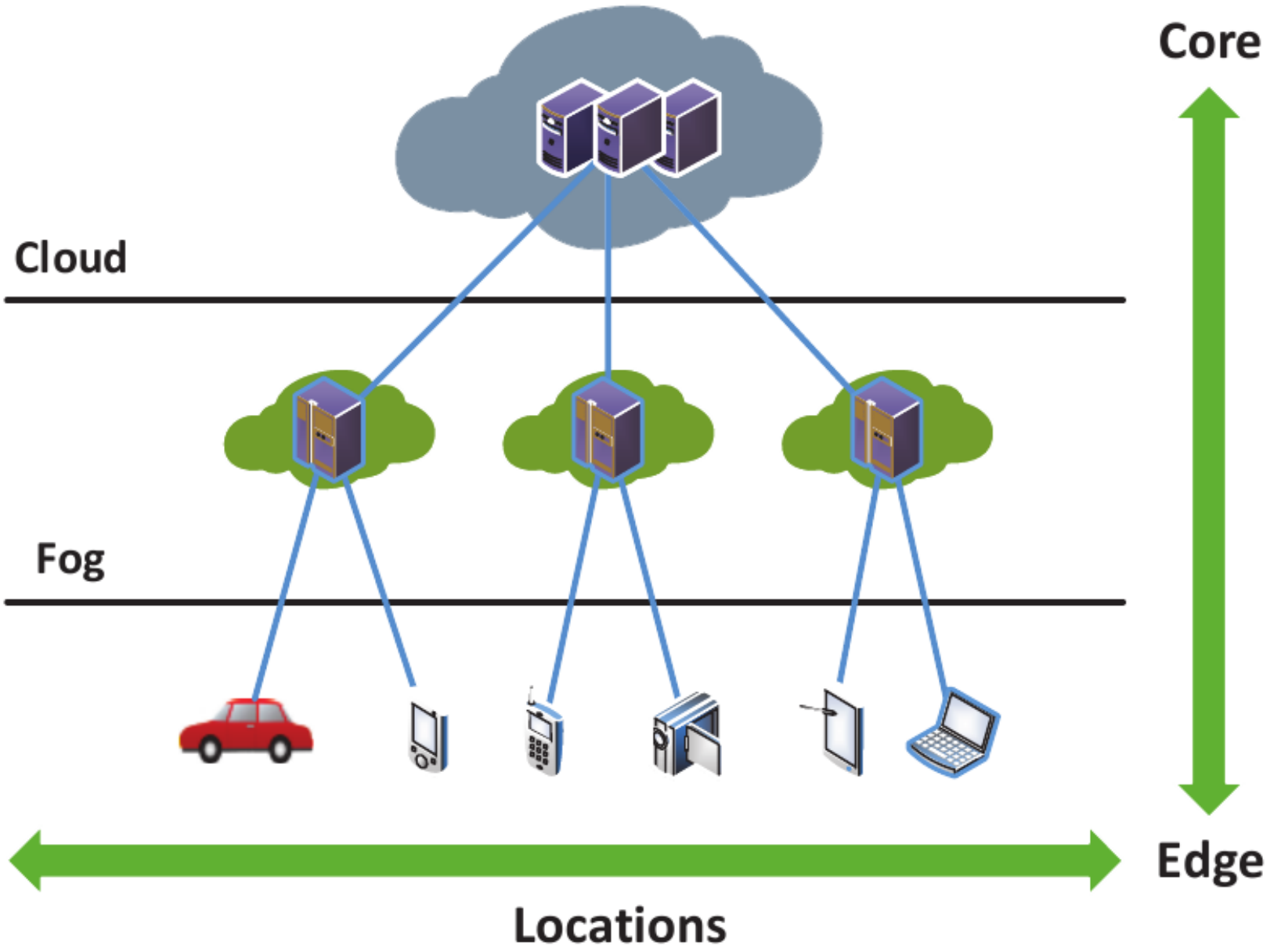



The IoT is becoming a crowded and complex place..

A plethora of programming models for "mobile/IoT applications"

- client side
  - single-device program: objects + functions + concurrency..
    ..threads/actors/futures/tasks/activities
  - local intelligence..
- client-server communication middleware/APIs
  - SOA (Http/Rest), MoM (RabbitMQ)
- server side
  - storage by DB: OO, relational, NoSQL – in memory/on-disk
  - 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

Heterogeneity of deployment contexts



Implications

Where programming effort ends up?

- programs of clients and servers highly depend on
  - the chosen platform / API / communication technology
  - the number, type, and dislocation of involved devices

$\Rightarrow$ IoT systems tend to be very rigid, hard and costly to debug/maintain

$\Rightarrow$ design and deployments hardly tolerate changes

The technological result

- systems difficultly scale with complexity of behaviour
- very few of the opportunities of large-scale IoT are currently 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, type, and dislocation of involved devices

$\Rightarrow$ IoT systems tend to be very rigid, hard and costly to debug/maintain

$\Rightarrow$ design and deployments hardly tolerate changes

The technological result

- systems difficultly scale with complexity of behaviour
- very few of the opportunities of large-scale IoT are currently 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 sytems?

- the plain old platform-independent programming abstraction..
- fully grounding system design
- tightly connected to programming abstractions
- mostly disconnected to the underlying "platform" (HW, network..)
- delegating to the underlying platform everything but the true "logic"
  - non-functional aspects (performance, resilience, self-* properties)
  - deployment issues

What's the lesson of other programming abstractions?

- procedure: a non-concurrent workflow of micro-tasks
- function: I/O tranformation of information
- object: state+behaviour, exposure of a clean interface
- actor: concurrency-enabled reactiveness, recv/send of async. messages
- agent: encapsulation of control, autonomy of behaviour, goal-orientation

What to do? A programming model perspective..

What do we lack in large-scale IoT sytems?

- the plain old platform-independent programming abstraction..
- fully grounding system design
- tightly connected to programming abstractions
- mostly disconnected to the underlying "platform" (HW, network..)
- delegating to the underlying platform everything but the true "logic"
  - non-functional aspects (performance, resilience, self-* properties)
  - deployment issues

What's the lesson of other programming abstractions?

- procedure: a non-concurrent workflow of micro-tasks
- function: I/O tranformation of information
- object: state+behaviour, exposure of a clean interface
- actor: concurrency-enabled reactiveness, recv/send of async. messages
- agent: encapsulation of control, autonomy of behaviour, goal-orientation

What to do? The challenge..

The challenge

Just directly consider the worst scenario possible..

- zillion devices unpredictably located and moving in the environment
- heterogeneous displacement, pervasive sensing/actuation
- computational behaviour is mostly contextual (i.e., spatial)
- abstracting away from the "server infrastructure"
  - whether we have fog++/cloud++ in background

  $\Rightarrow$ but be ready to exploit the opportunities it creates!

What to do? The challenge..

The challenge

Just directly consider the worst scenario possible..

- zillion devices unpredictably located and moving in the environment
- heterogeneous displacement, pervasive sensing/actuation
- computational behaviour is mostly contextual (i.e., spatial)
- abstracting away from the "server infrastructure"
  - whether we have fog++/cloud++ in background

  $\Rightarrow$ but be ready to exploit the opportunities it creates!

Let's try to program *that* "computational system"!

What to do? The challenge..

The challenge

Just directly consider the worst scenario possible..

- zillion devices unpredictably located and moving in the environment
- heterogeneous displacement, pervasive sensing/actuation
- computational behaviour is mostly contextual (i.e., spatial)
- abstracting away from the "server infrastructure"
  - whether we have fog++/cloud++ in background

  $\Rightarrow$ but be ready to exploit the opportunities it creates!

Let's try to program *that* "computational system"!

What is the right programming abstraction in this case?

Towards "Aggregate Computing"

Systems of interest: collective adaptive situated systems CASS

- (possibly very large scale) collective adaptive systems
- deployed in physical space (situated, spatial), 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
- it is supported by a toolchain for design and programming
- simple idea, few constructs, rather tractable
- somehow "*very different*"

Towards "Aggregate Computing"

Systems of interest: collective adaptive situated systems CASS

- (possibly very large scale) collective adaptive systems
- deployed in physical space (situated, spatial), 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
- it is supported by a toolchain for design and programming
- simple idea, few constructs, rather tractable
- somehow "*very different*"

On this seminar

Outline

1. Background, motivation and idea of aggregate computing
2. The computational model: field calculus
3. The toolchain: Scafi
   - mini-tutorial of the language
   - platform support
4. Engineering of resilient systems
   - self-stabilisation
   - building blocks
   - eventual consistency
5. Conclusions

Outline

1. Aggregate Computing
2. Field Calculus
   - Computing with fields
   - Syntax
   - Practical expressiveness
   - Local semantics and platform support
3. Toolchain
   - Scafi: core and simulation
   - Scafi: distributed platform
4. Field Engineering and resilience
   - Self-stabilisation
   - Eventual consistency
   - Controlling dynamics

Outline

An example opportunity for IoT-based CASS..

Gathering local context

Sensing global patterns of data

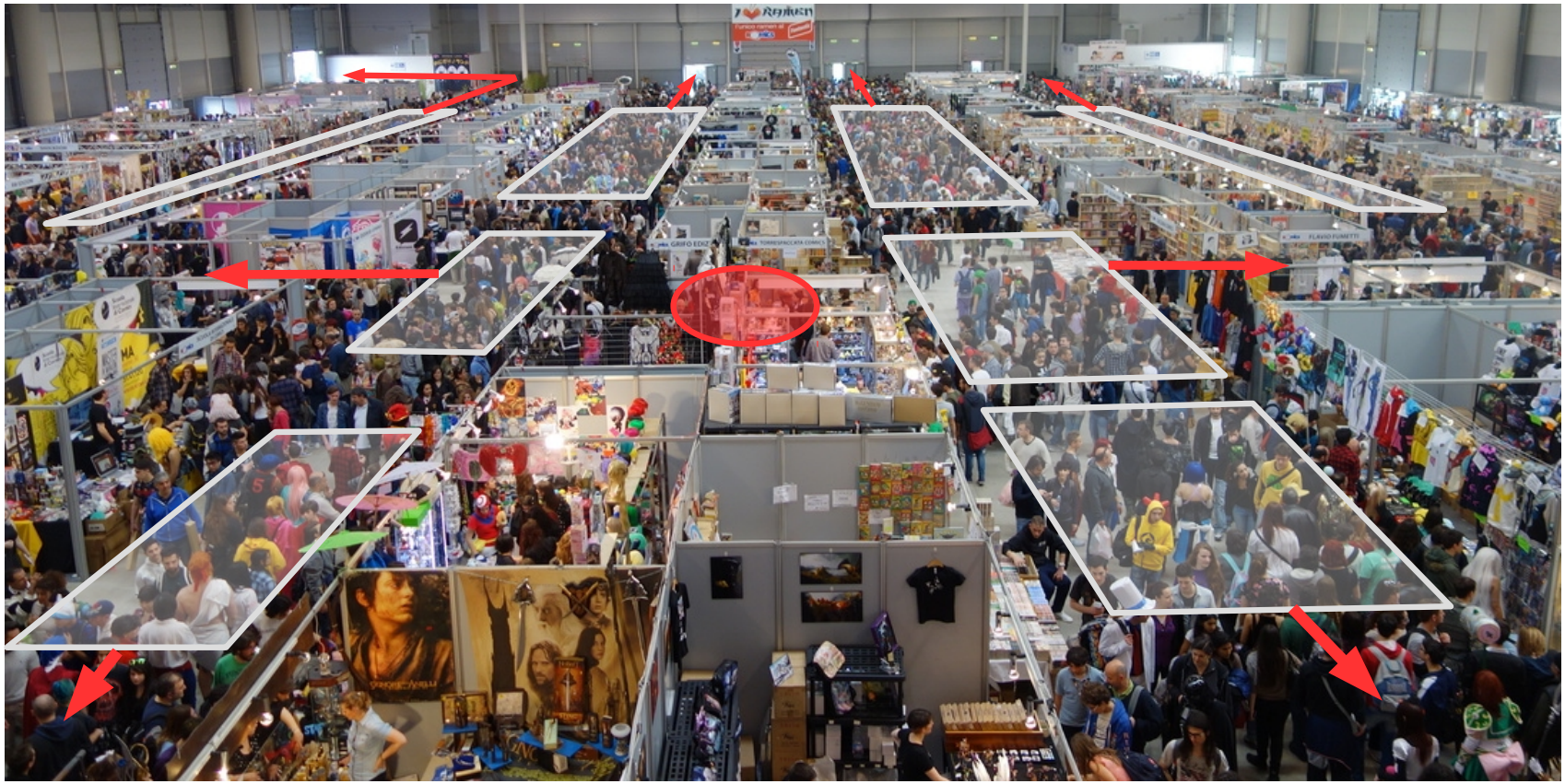
Crowd Detection

Crowd Anticipation

Crowd-aware Steering

Crowd dispersal

Crowd evacuation upon alerts



Let's follow standard "software engineering" process

Requirements and Analysis

- The customer does not mention "servers" or "connectivity"
- Different services to be implemented
- All in need of robustness at different levels
- Several common problems, to be "factored out"

Architectural design

- Depict strategies and abstractions in a platform-independent way
- Using concepts very near to the problem domain
- Identify common patterns

Detailed design and other stages

- Choose technologies, write APIs and component interfaces
- Implementation, Testing, Deployment

Let's follow standard "software engineering" process

Requirements and Analysis

- The customer does not mention "servers" or "connectivity"
- Different services to be implemented
- All in need of robustness at different levels
- Several common problems, to be "factored out"

Architectural design

- Depict strategies and abstractions in a platform-independent way
- Using concepts very near to the problem domain
- Identify common patterns

Detailed design and other stages

- Choose technologies, write APIs and component interfaces
- Implementation, Testing, Deployment

Let's follow standard "software engineering" process

Requirements and Analysis

- The customer does not mention "servers" or "connectivity"
- Different services to be implemented
- All in need of robustness at different levels
- Several common problems, to be "factored out"

Architectural design

- Depict strategies and abstractions in a platform-independent way
- Using concepts very near to the problem domain
- Identify common patterns

Detailed design and other stages

- Choose technologies, write APIs and component interfaces
- Implementation, Testing, Deployment

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
  3π, 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
  3π, Shape Calculus, bi-graphs, KLAIM, $\sigma\tau$-linda, SCEL
- *Spatial computing* – program space-time behaviour of systems
  Proto, MGS

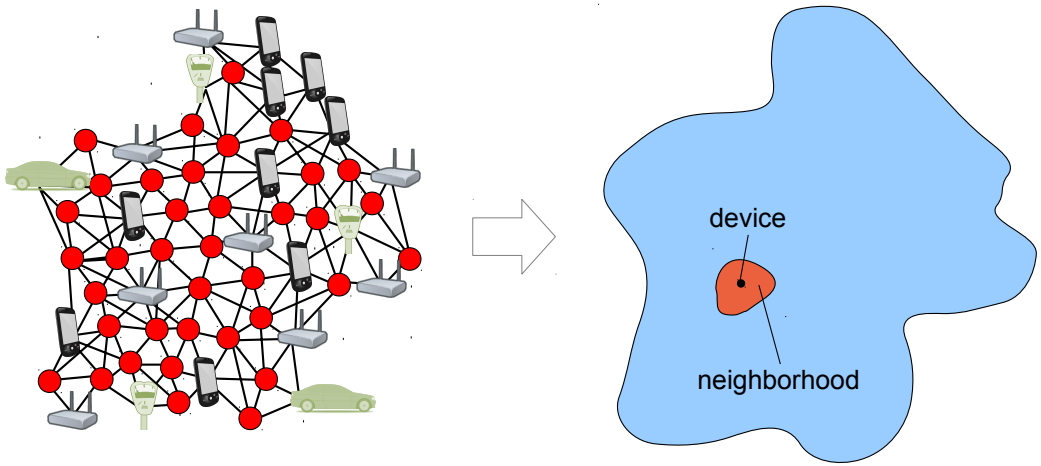
Our approach

- Combining the above efforts of "macro" programming
- Taking some of those ideas to the extreme consequences

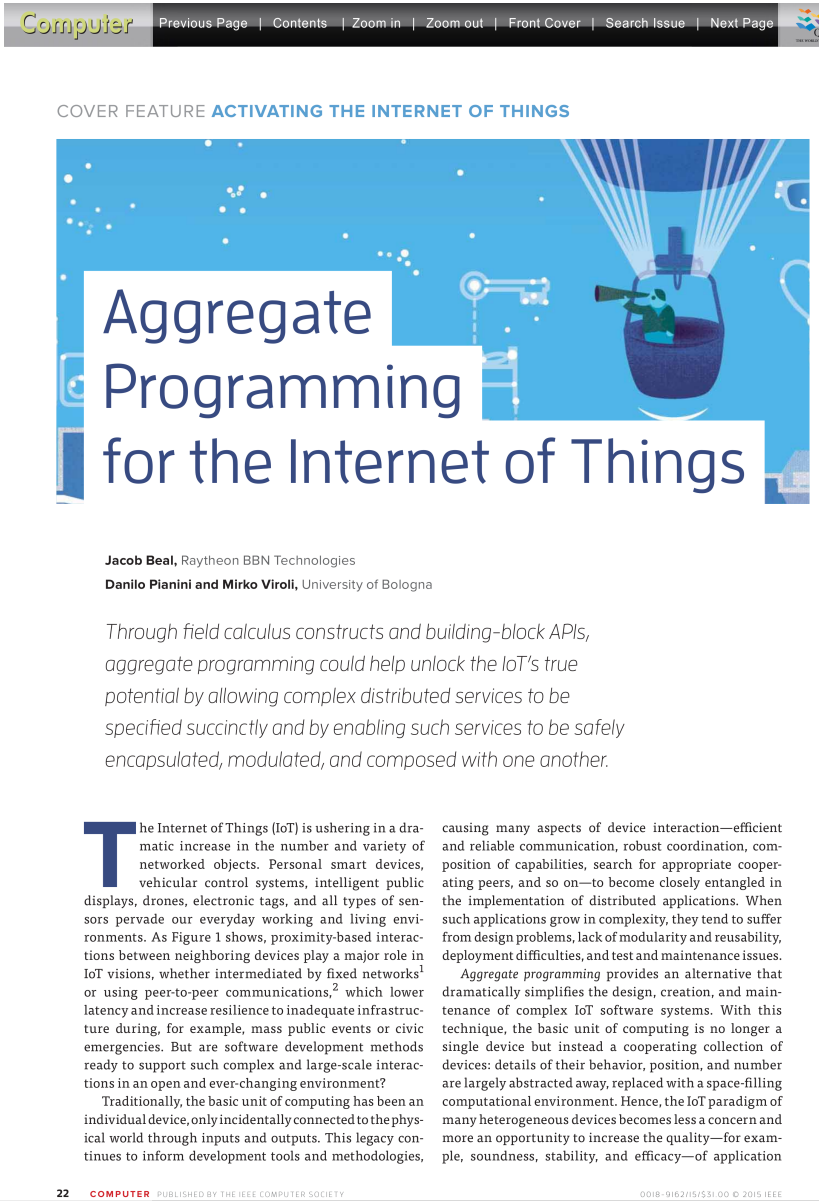
Manifesto of aggregate computing

Motto: program the aggregate, not individual devices!

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



Aggregate programming at IEEE Computer, Sep. 2015



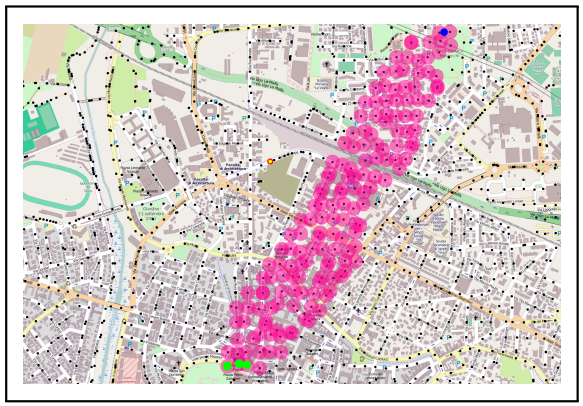
Outline

Outline

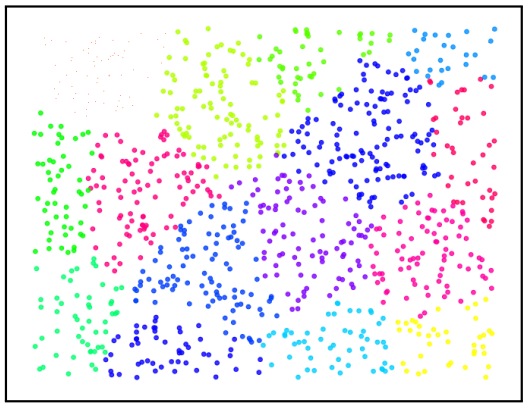
Computational Fields [Mamei et.al., 2009, Beal et.al., 2013]

Traditionally a map: *Space* $\mapsto$ *Values*

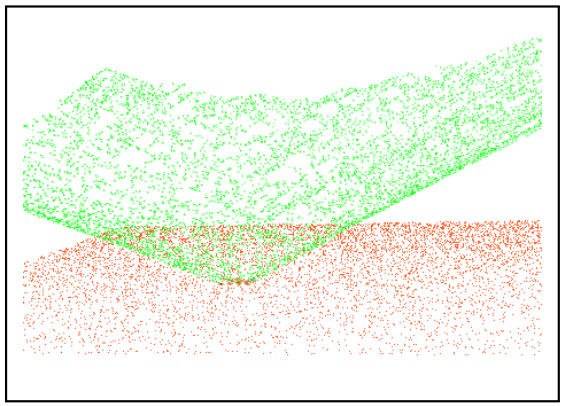
- possibly: evolving over time, dynamically injected, stabilising
- smoothly adapting to very heterogeneous domains
- more easily "understood" on continuous and flat spatial domains
- ranging to: booleans, reals, vectors, functions



boolean channel in 2D



numeric partition in 2D

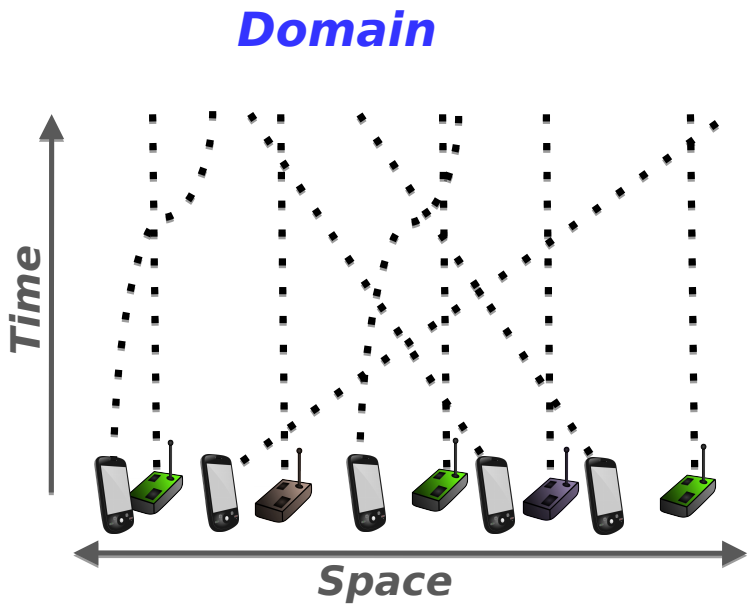


real-valued gradient in 3D

(Computational) Fields revisited

A field as a space-time structure: $\phi : D \mapsto V$

- *event* E: a triple $\langle \delta, t, p \rangle$ – device δ, "firing" at time t in position p
- *events domain* D: a coherent set of events (devices cannot move too fast)
- *field values* V: any data value

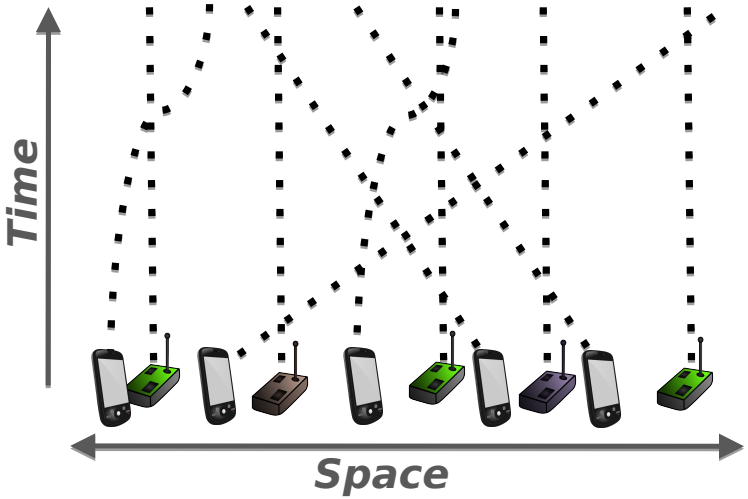


(Computational) Fields revisited

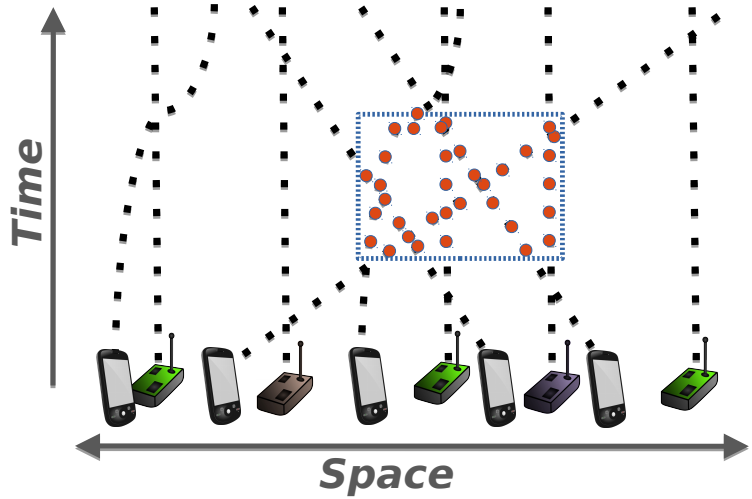
A field as a space-time structure: $\phi : D \mapsto V$

- *event* E: a triple $\langle \delta, t, p \rangle$ – device δ, "firing" at time t in position p
- *events domain* D: a coherent set of events (devices cannot move too fast)
- *field values* V: any data value

Domain



Field

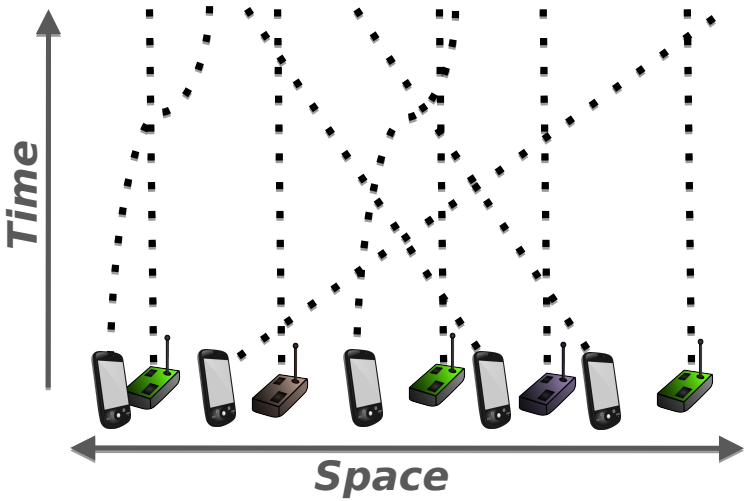


(Computational) Fields revisited

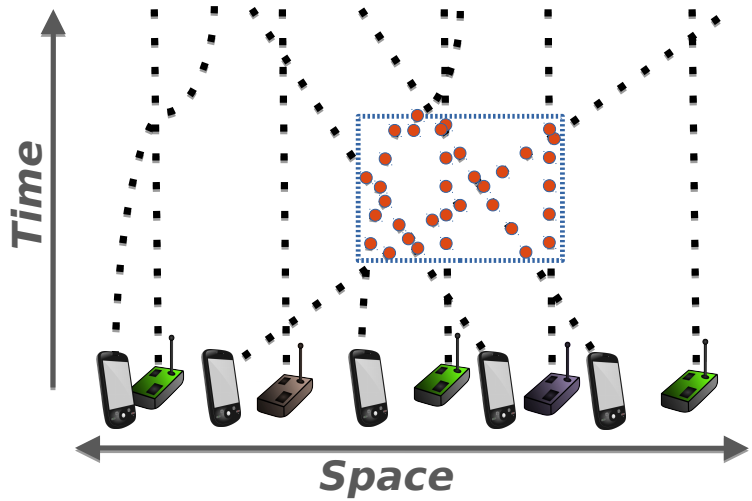
A field as a space-time structure: $\phi : D \mapsto V$

- *event* E: a triple $\langle \delta, t, p \rangle$ – device δ, "firing" at time t in position p
- *events domain* D: a coherent set of events (devices cannot move too fast)
- *field values* V: any data value

Domain



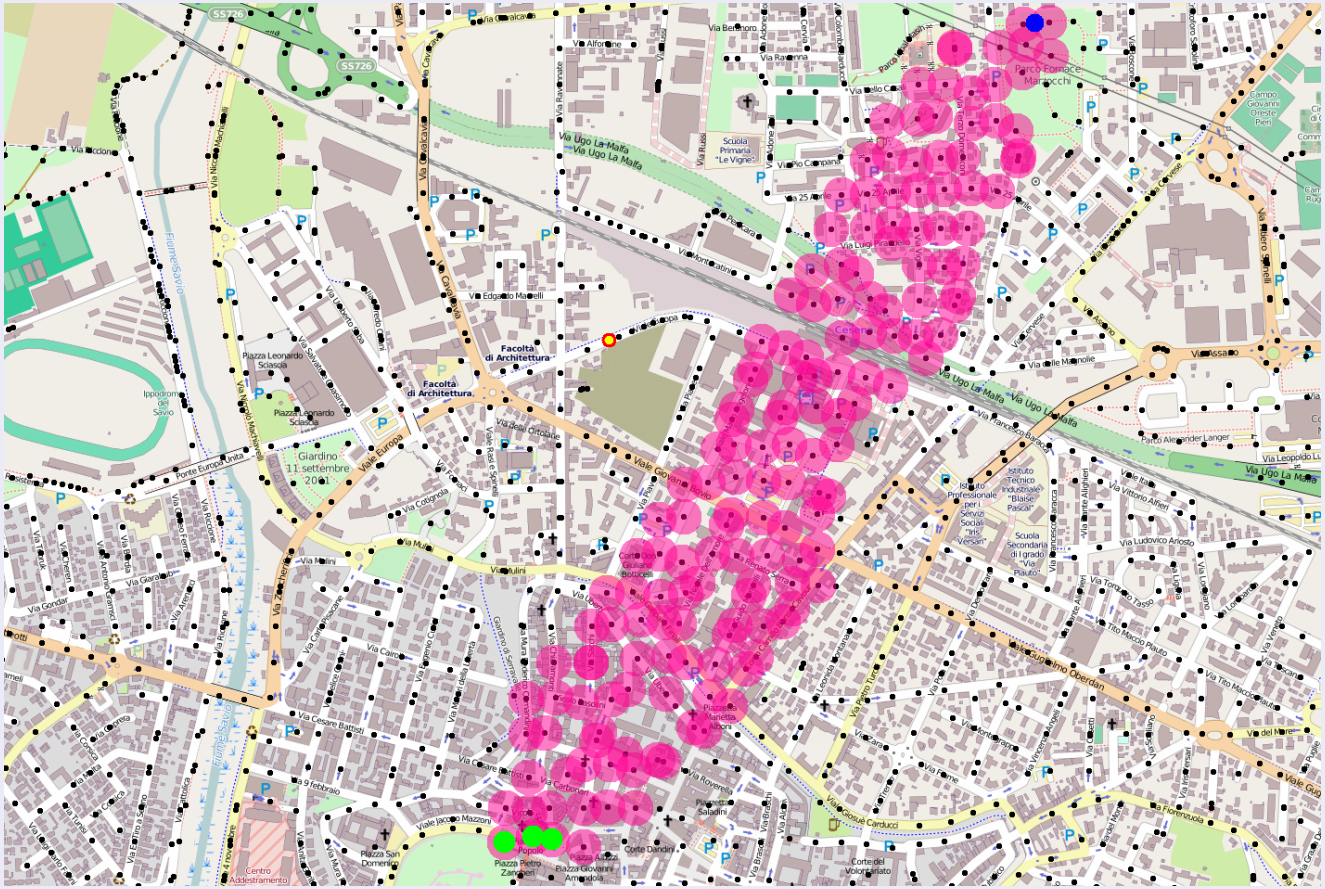
Field



will later show only snaphots of fields in 2D space..

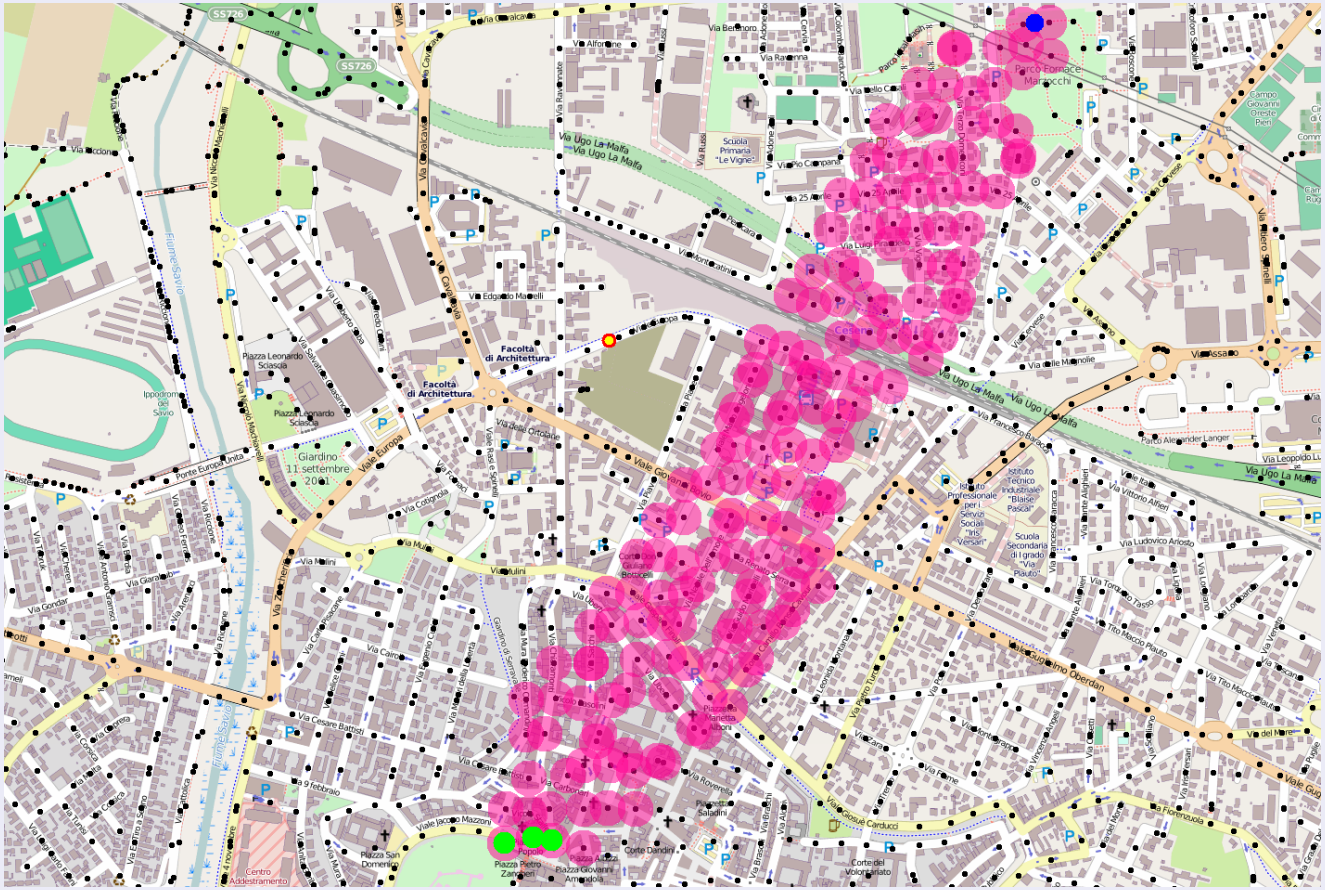
The "channel" example: computing a route

Atemporal visualisation of the "channel" boolean field



The "channel" example: computing a route

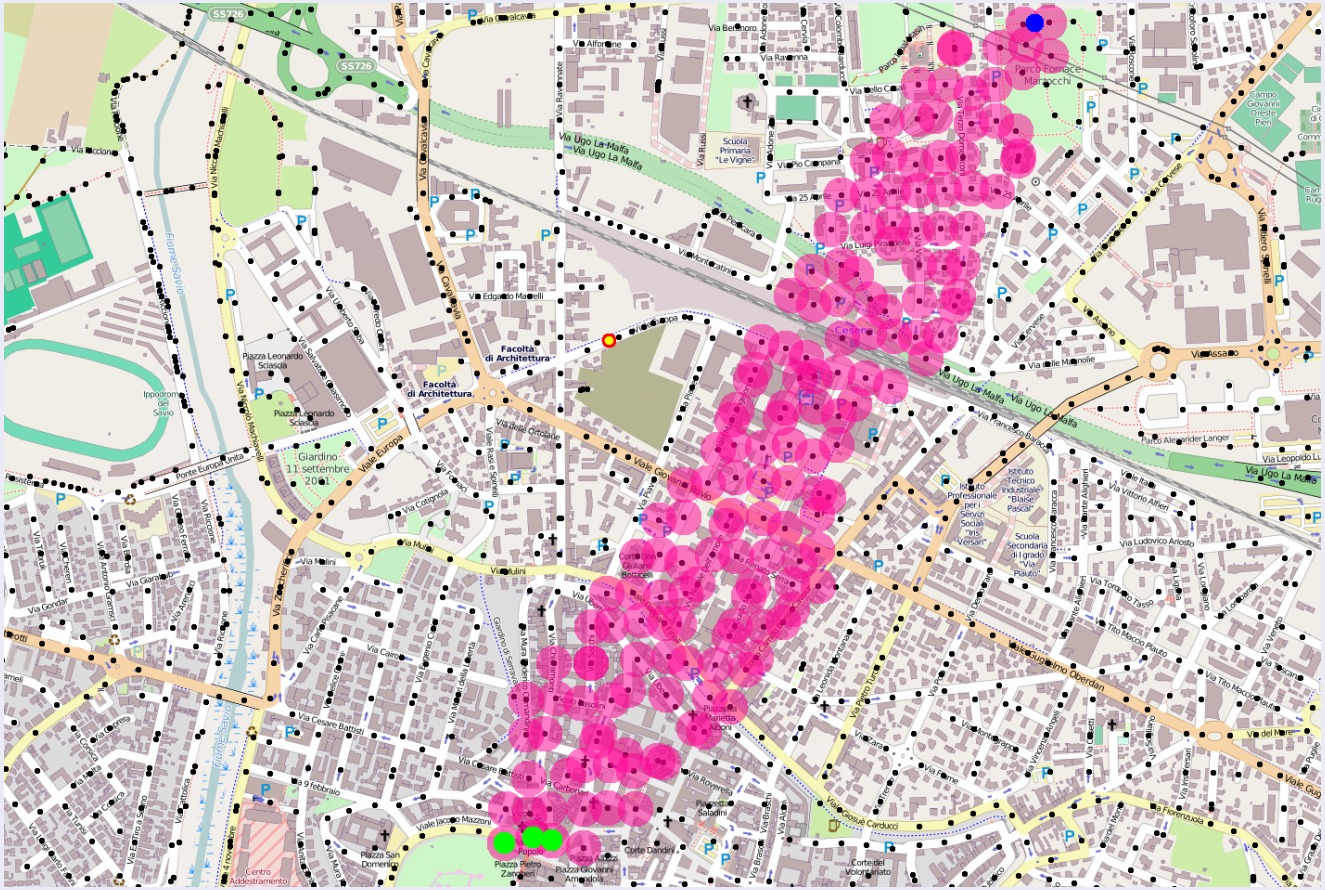
Atemporal visualisation of the "channel" boolean field



the figure shows a stabilised time-snapshot of the field,
in time, there's a transient for the formation of the field

The "channel" example: computing a route

How would you program it?



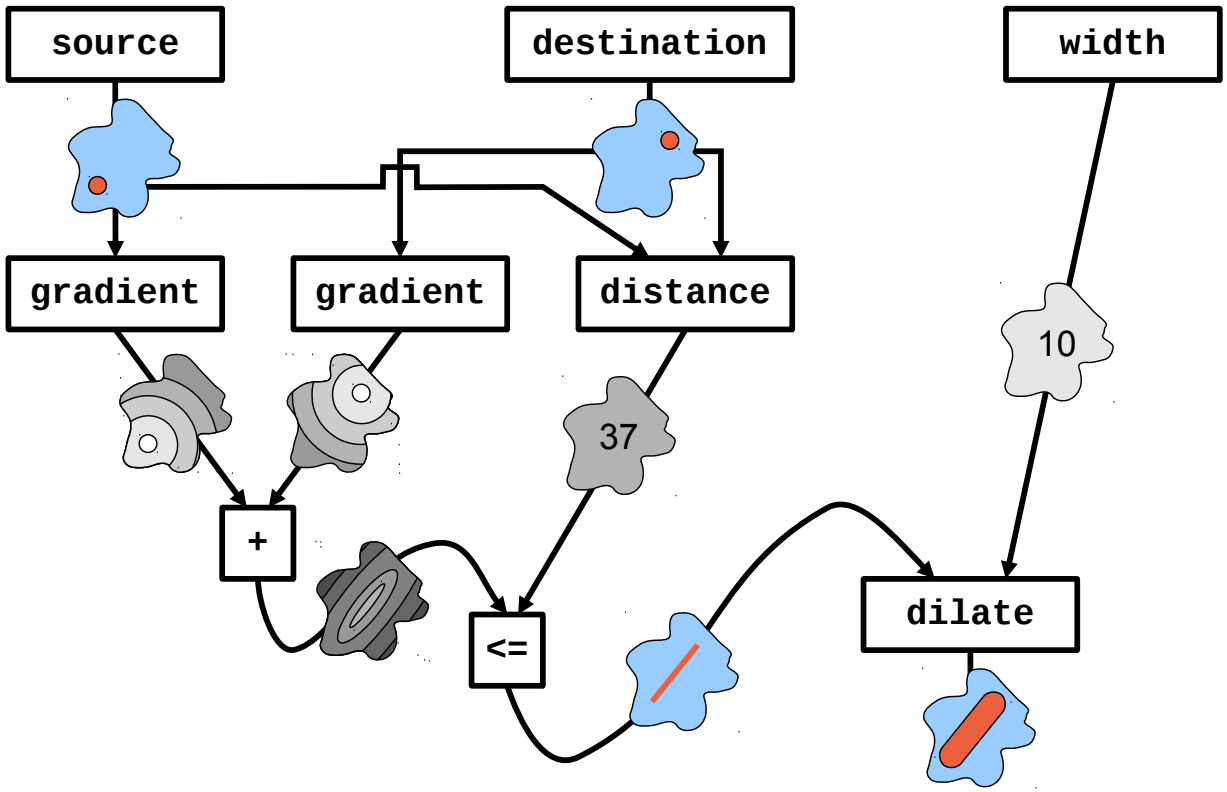
how could the channel program be platform-independent, unaware of global map, resilient to changes, faults,..

Aggregate programming as a functional approach

Functionally composing fields

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

$\Rightarrow$ for this to be practical/expressive we need a good programming language

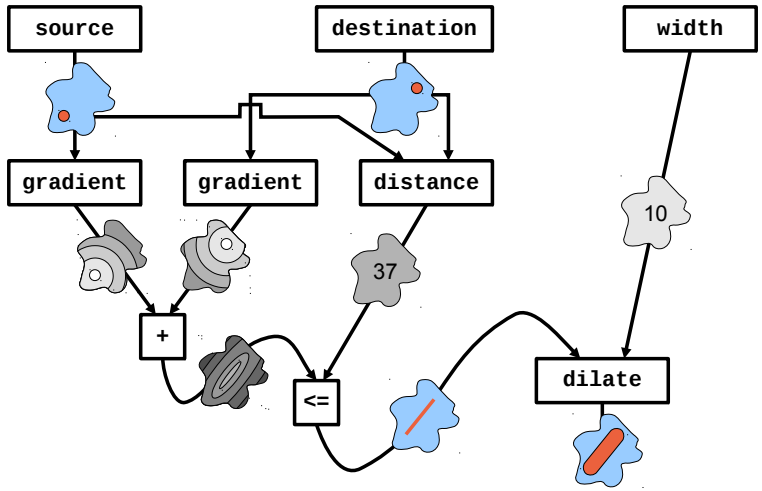


Preview / requirement

How we want that computation to be expressed?

- `source`, `dest` and `width` as inputs
- `gradient`, `distance` and `dilate` as reusable functions

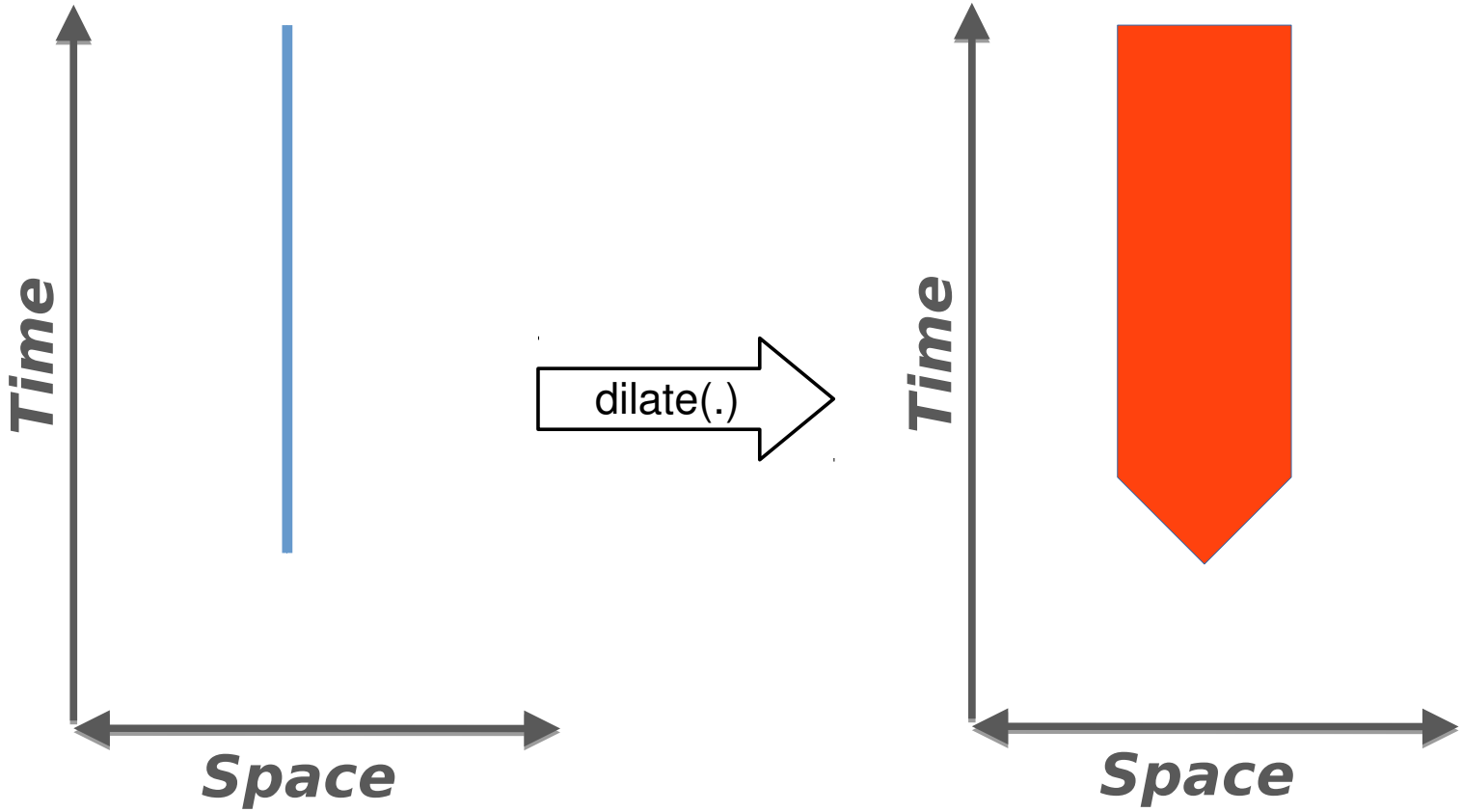
$\Rightarrow$ note we are reusing and composing global-level, aggregate specs




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

Computing abstraction: computing evolving fields

dilate as a function over evolving fields



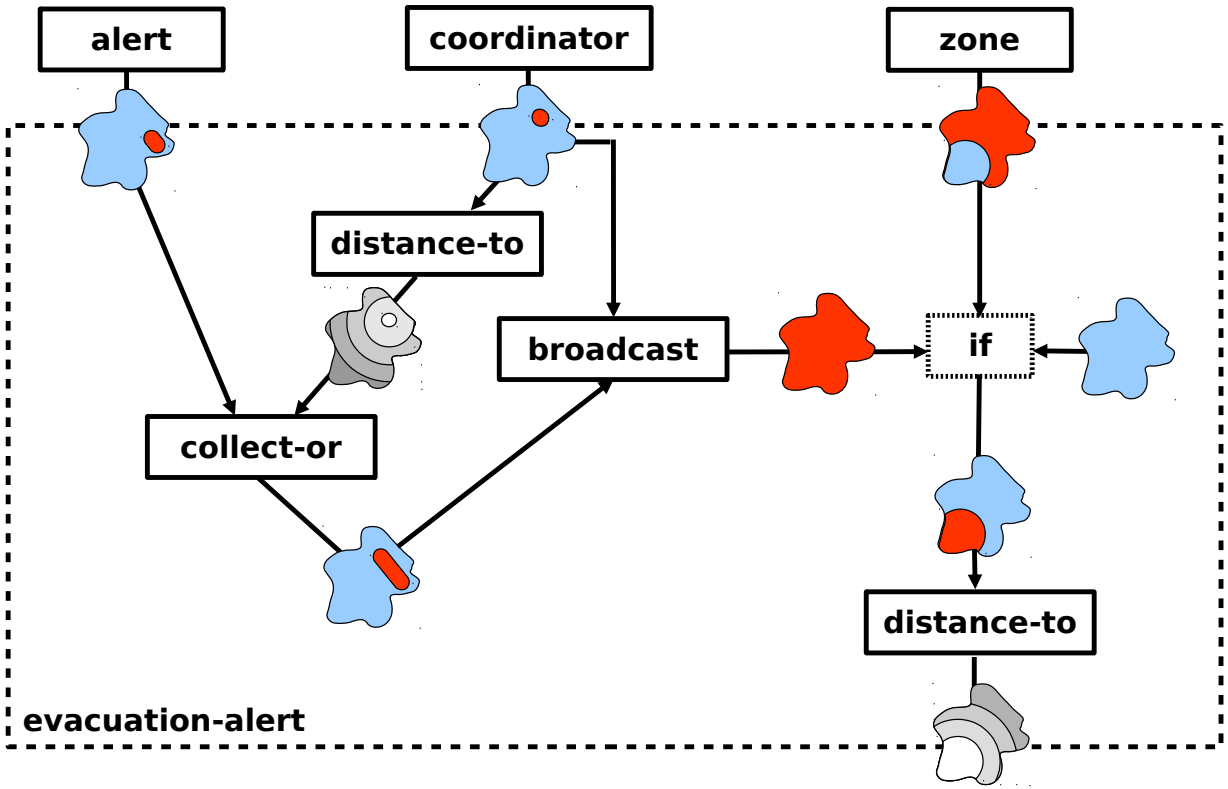
Note, finite velocity of information propagation

Crowd evacuation as a field computation

Computing by purely functional composition of space-time fields

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

$\Rightarrow$ for this to be practical/expressive we need a good programming language



Outline

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)

$$\texttt{e} ::= \texttt{x} \mid \texttt{v} \mid \texttt{e}(\texttt{e}_1, \ldots, \texttt{e}_n) \mid \texttt{rep}(\texttt{e}_0)\{\texttt{e}\} \mid \texttt{nbr}\{\texttt{e}\} \quad \text{(expr)}$$

$$\texttt{v} ::= < \text{standard-values} > \mid \lambda \quad \text{(value)}$$

$$\lambda ::= \texttt{f} \mid \texttt{o} \mid (\overline{\texttt{x}})\texttt{=>e} \quad \text{(functional value)}$$

$$\texttt{F} ::= \texttt{def f}(\overline{\texttt{x}})\ \{\texttt{e}\} \quad \text{(function definition)}$$

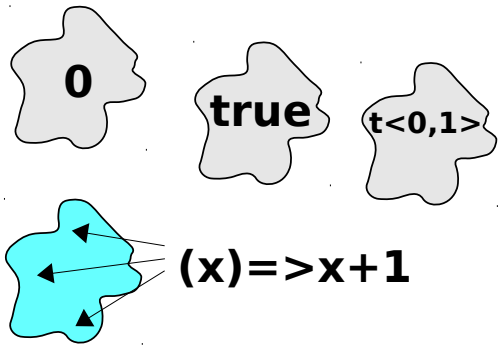
Few explanations

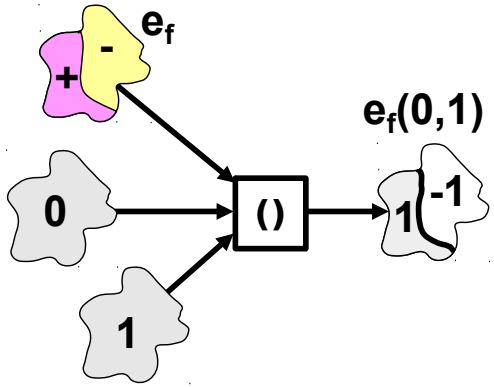
- v includes numbers, booleans, strings,..
  ..tuples/vectors/maps/any-ADT (of expressions)
- f is a user-defined function (the key aggregate computing abstraction)
- o is a built-in local operator (pure math, local sensors,..)

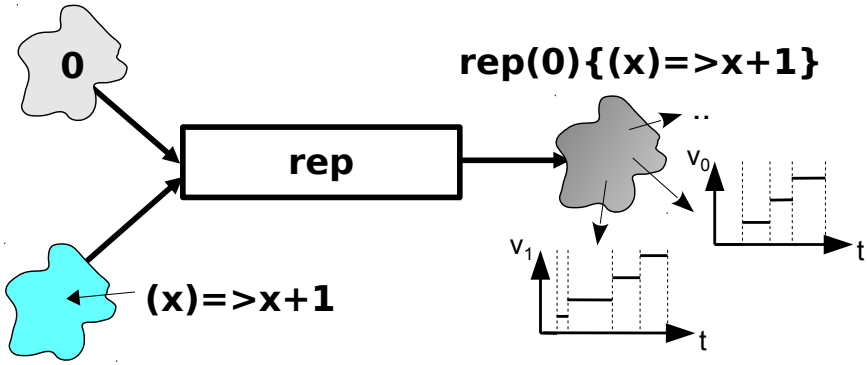
Intuition of global-level (denotational) semantics

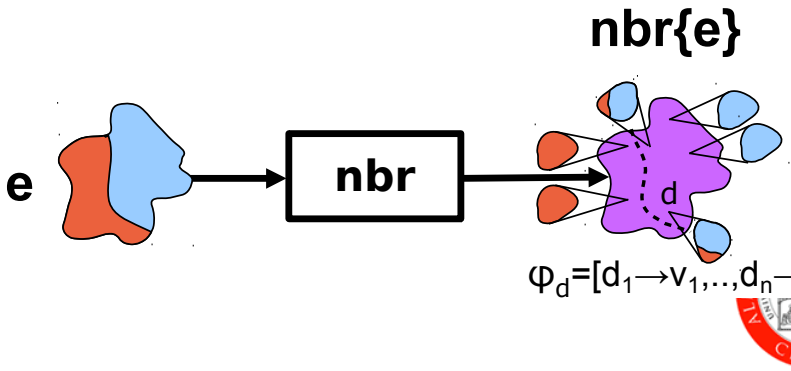
The four main constructs at work
⇒ values, application, evolution, and interaction – in aggregate guise

- $e ::= \ldots \mid v \mid e(e_1, \ldots, e_n) \mid \texttt{rep}(e_0)\{e\} \mid \texttt{nbr}\{e\}$









A mini-tutorial

```
1: 1
2: 2 + 3
3: pair(10,20)
4: random()
5: sense(1)
6: sense(1) ? 10 : 20
7: mid()
8: minHood(nbrRange)
```

```
9: rep(0){ (x) => x + 1 }
10: rep(random()){ (x) => x }
11: rep(0){ (x) => x + rep(random()){ (y) => y } }
```

```
12: maxHood( nbr{ sense(1) } )
13: sumHood( nbr{ 1 } )
```

```
14: rep(0){ (x) => max( sense(1), maxHood( nbr{ x } ) ) }
15: rep(Infinity) { (d) => sense(1) ? 0 : minHood( nbr{d} + 1 ) }
16: rep(Infinity) { (d) => sense(1) ? 0 : minHood( nbr{d} + nbrRange ) }
```

Intuition of global-level semantics

Value $\mathtt{v}$

- A field constant in space and time, mapping any event to $\mathtt{v}$

Function application $\mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$

- $\mathtt{e}$ 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: $\forall i$, $\lambda_i(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ restricted in D_i

Repetition $\mathtt{rep}(\mathtt{e}_0)\{\mathtt{e}_\lambda\}$

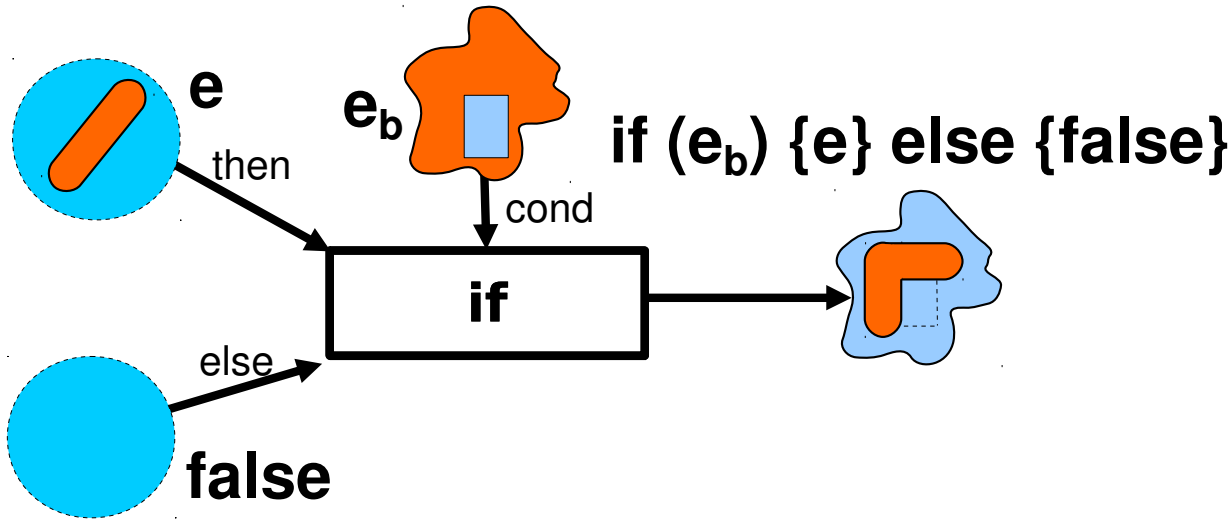
- the value of $\mathtt{e}_0$ where the restricted domain "begins"
- elsewhere, unary function $\mathtt{e}_\lambda$ is applied to previous value at each device

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

- at each event gathers most recent value of $\mathtt{e}$ in neighbours (in restriction)
- ..what is neighbour is orthogonal (i.e., physical proximity)

Outline

The restriction trick: branching behaviour



if as a space-time branching construct

```
if(e-bool){e-then}else{e-else}
≈
(e-bool ? ()=>{e-then} : ()=>{e-else})()
```

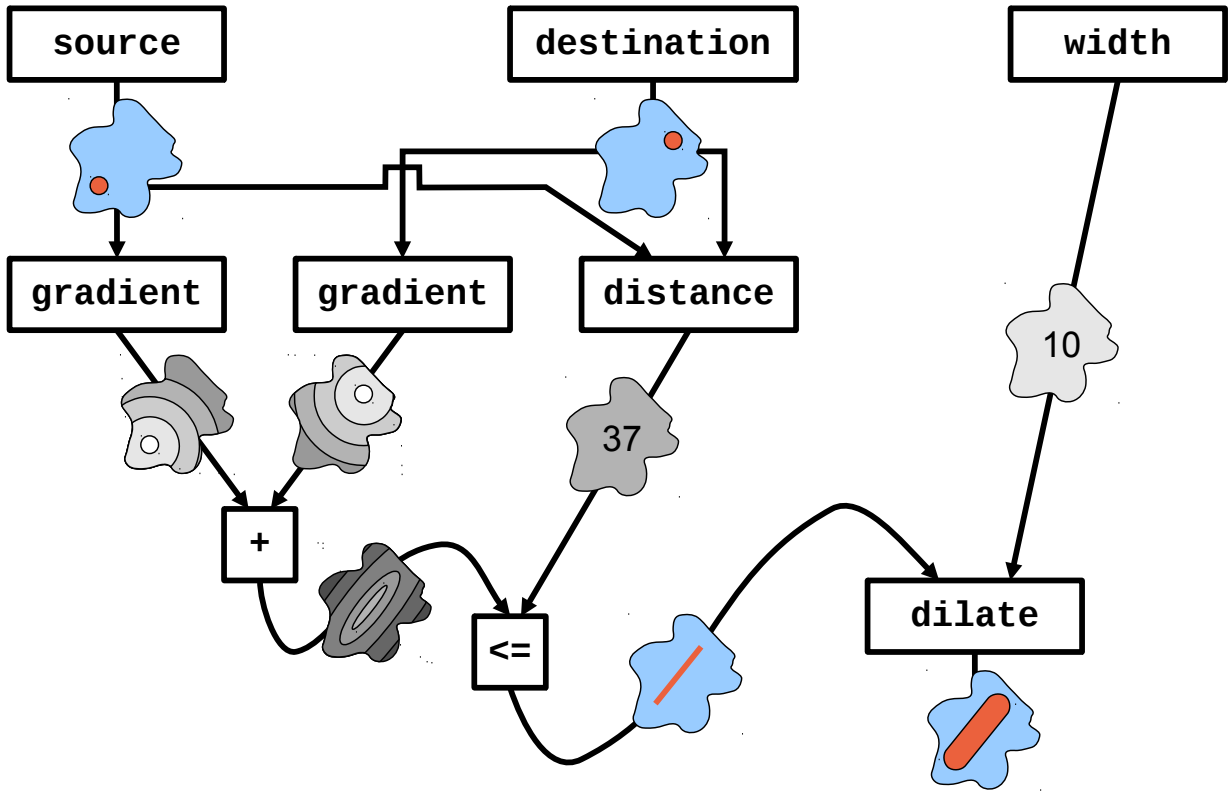
More advanced patterns

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

Aggregate programming as a functional approach

Functionally composing fields

- ...so, is field calculus language practical/expressive?



A preview: 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 booleans
  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

Builtin functions exploited

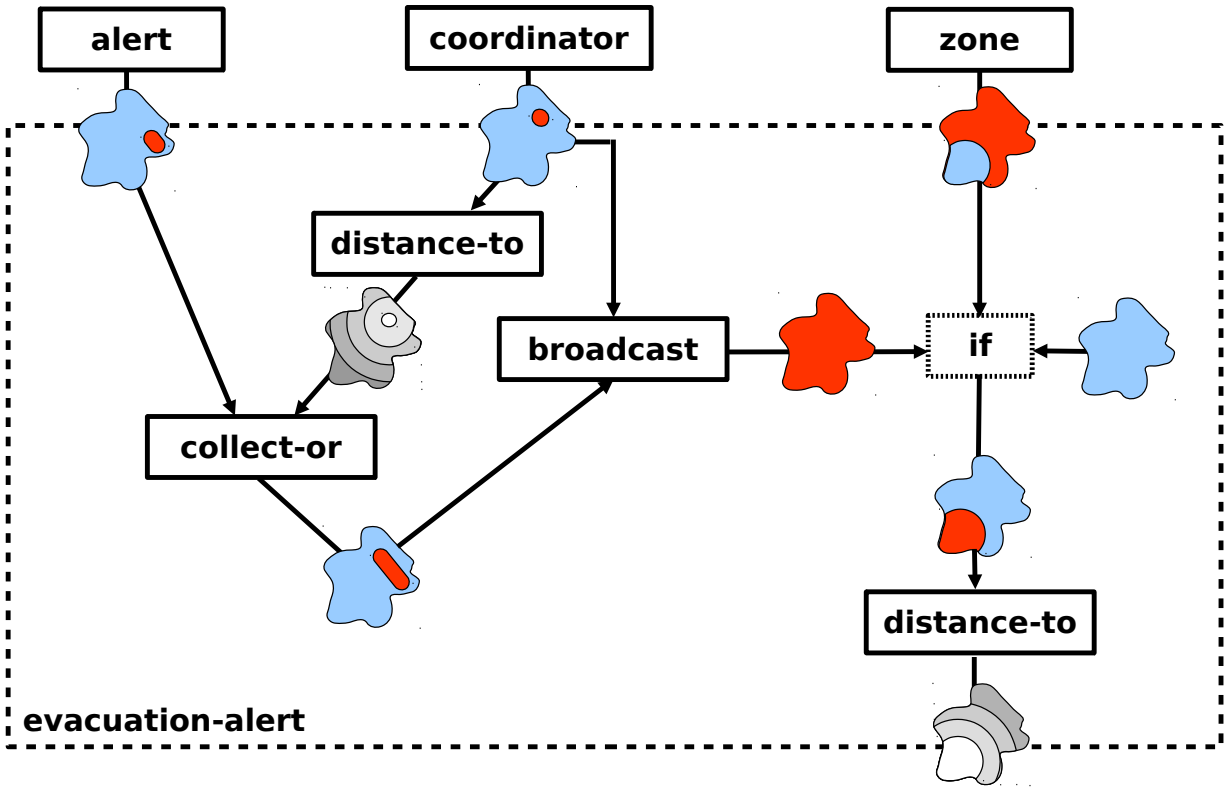
- ?: — Java-like (though, call-by-value) ternary operator
- nbrRange — maps each device to a neighbour field of estimated distances
- minHood — in each device, collapse a neighbour field into its minimum value
- sumHood — in each device, collapse a neighbour field into sum of values
- *,-,*,/,>,... —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

Crowd evacuation as a field computation

Computing by purely functional composition of space-time fields

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

$\Rightarrow$ for this to be practical/expressive we need a good programming language



A preview: Evacuation example

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

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

def collect-or (potential, value){;; Collects 'value' by descending 'potential', by 'or'
  rep(value){
     (v) => anyHood( nbr{find-parent potential()} = uid ? nbr{v} : false )
            or value
} }

def evacuation-alert (zone, coordinator, alert){
  distance-to(
     if(zone){false} else {
         broadcast(coordinator,collect-or(distance-to(coordinator),alert))})
}
```

On expressiveness of the field calculus

Practically, we can express:

- complex spreading / aggregation / decay functions [IEEE Computer 48(9), 2015]
- spatial leader election, partitioning, consensus [SCP2015]
- distributed spatio-temporal sensing [COORD2016]
- splitting in parallel independent subprocesses [COORD2016]
- dynamic deployment/spreading of code (via lambda) [Damiani & Viroli & Beal & Pianini, FORTE2015]
- implicit/explicit device selection of what code execute [PRIMA2015]
- "collective teams" forming based on the selected code [PRIMA2015]

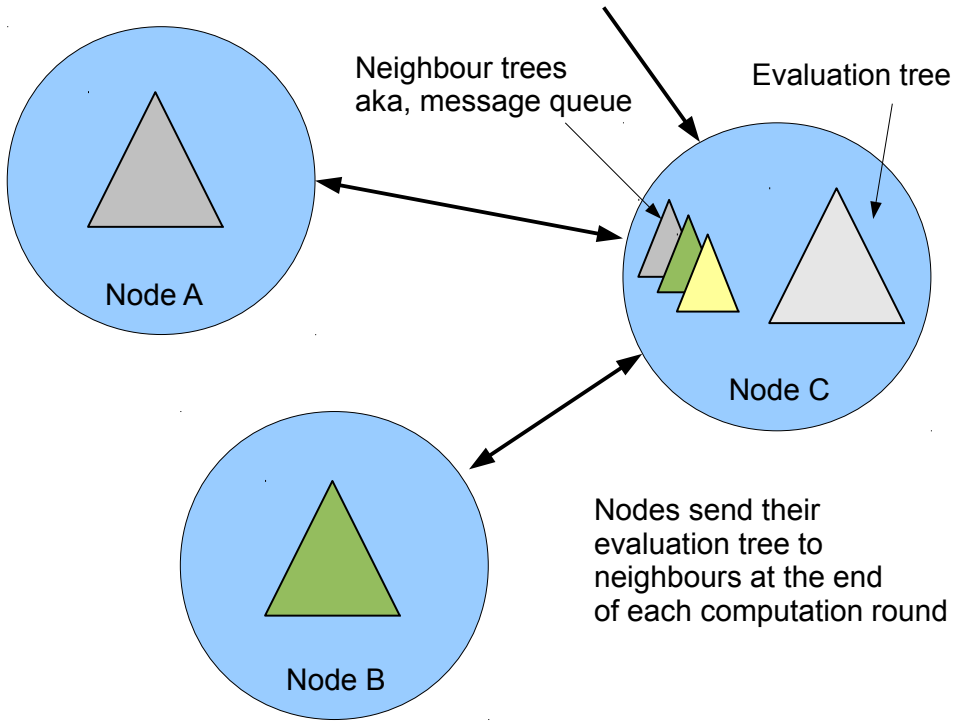
$\Rightarrow$ ..and critically, we can smoothly compose/nest the above features..

Outline

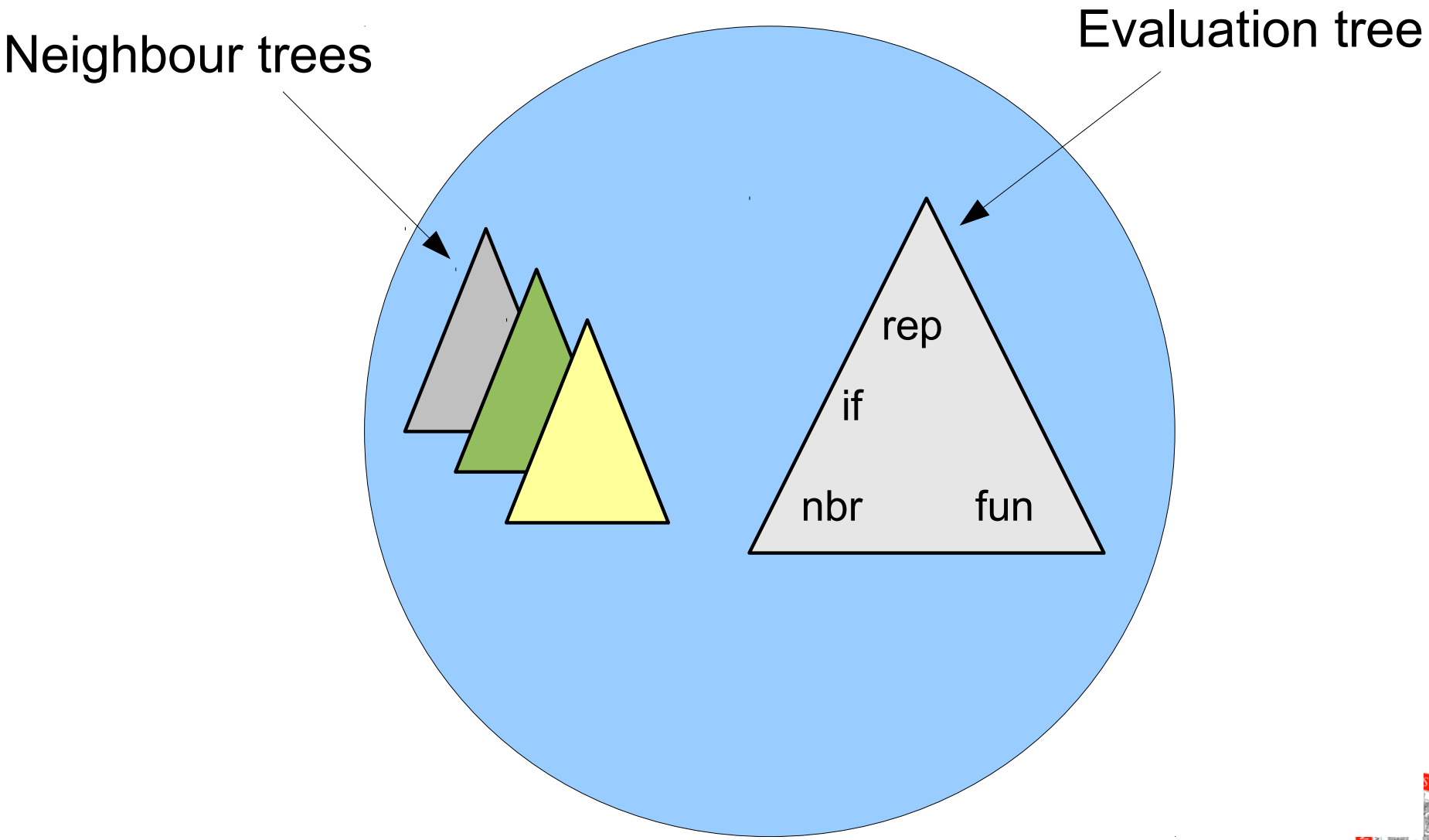
Key aspects of the semantics: network model

Platform abstract model

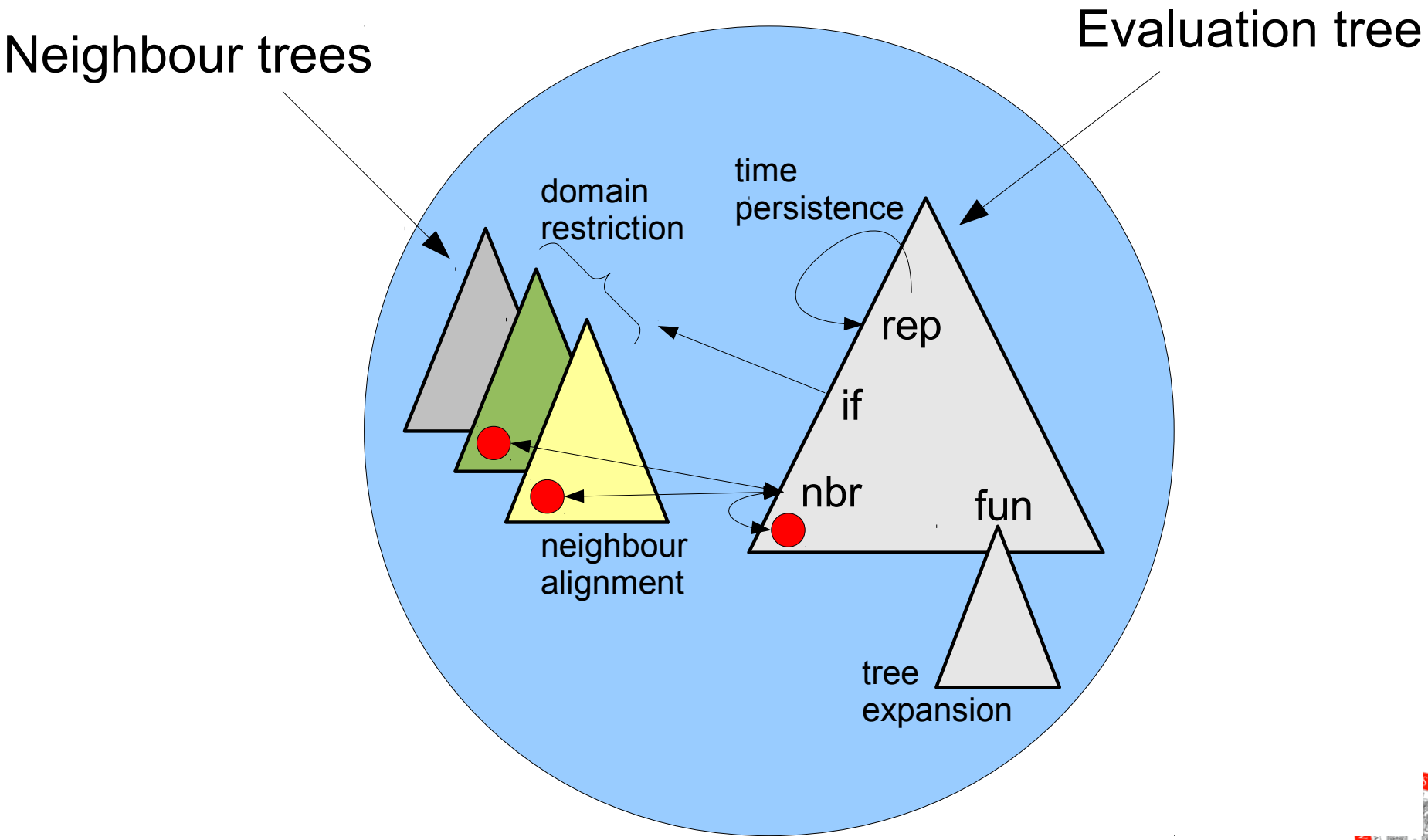
- A node state θ (value-tree) updated at asynchronous rounds
- At the end of the round, θ is made accessible to the neighbourhood
- A node state is updated "against" recently received neighbours' trees



Tree evaluation: pictorial semantics



Tree evaluation: pictorial semantics



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 $\texttt{rep}(e_0)\{e_\lambda\}$

- if a previous value-tree of mine is available, evaluates e_λ on its root
- otherwise, evaluates e_0

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

- 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 $\texttt{rep}(e_0)\{e_\lambda\}$

- if a previous value-tree of mine is available, evaluates e_λ on its root
- otherwise, evaluates e_0

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

- 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 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!*

Natural implementations

P2P

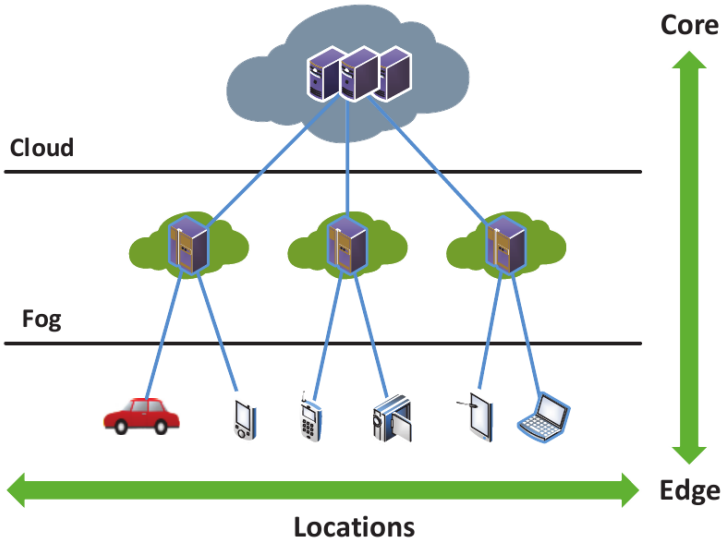
- devices see neighbours, and directly broadcast messages (ad-hoc wifi)

$\Rightarrow$ 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

$\Rightarrow$ not hard to handle 10K devices firing at 1Hz



Outline

Toolchain

Motivation

- synthesise advanced algoritms
- empirically evaluate functional and non-functional properties
- prototype distributed systems

$\Rightarrow$ ..all this requires a set of tools to close the gap between the formal model of field calculus and applications

Stack

- programming language
- core interpreter + checker
- libraries
- simulator
- distributed execution platforms

On the difficulty of maintaining a tool

Research tools

- sooner or later most of you will be needed to create one
- some groups might already have significant expertise
- critical question: is the tool for yourself or for the community, really?

Well-engineered and accessible tools: issues

- when is tool development really "over"?
- do not get stuck in spending your time as developer
- learning "continuous-integration tools" might save some of your life
  - `git`, `gradle`
  - it is also useful professionally, later
- getting your tool be happily used by the community is an exceptional case
  - it will be felt as more exotic than you think it is
  - documentation won't be enough

General state of our Toolchain

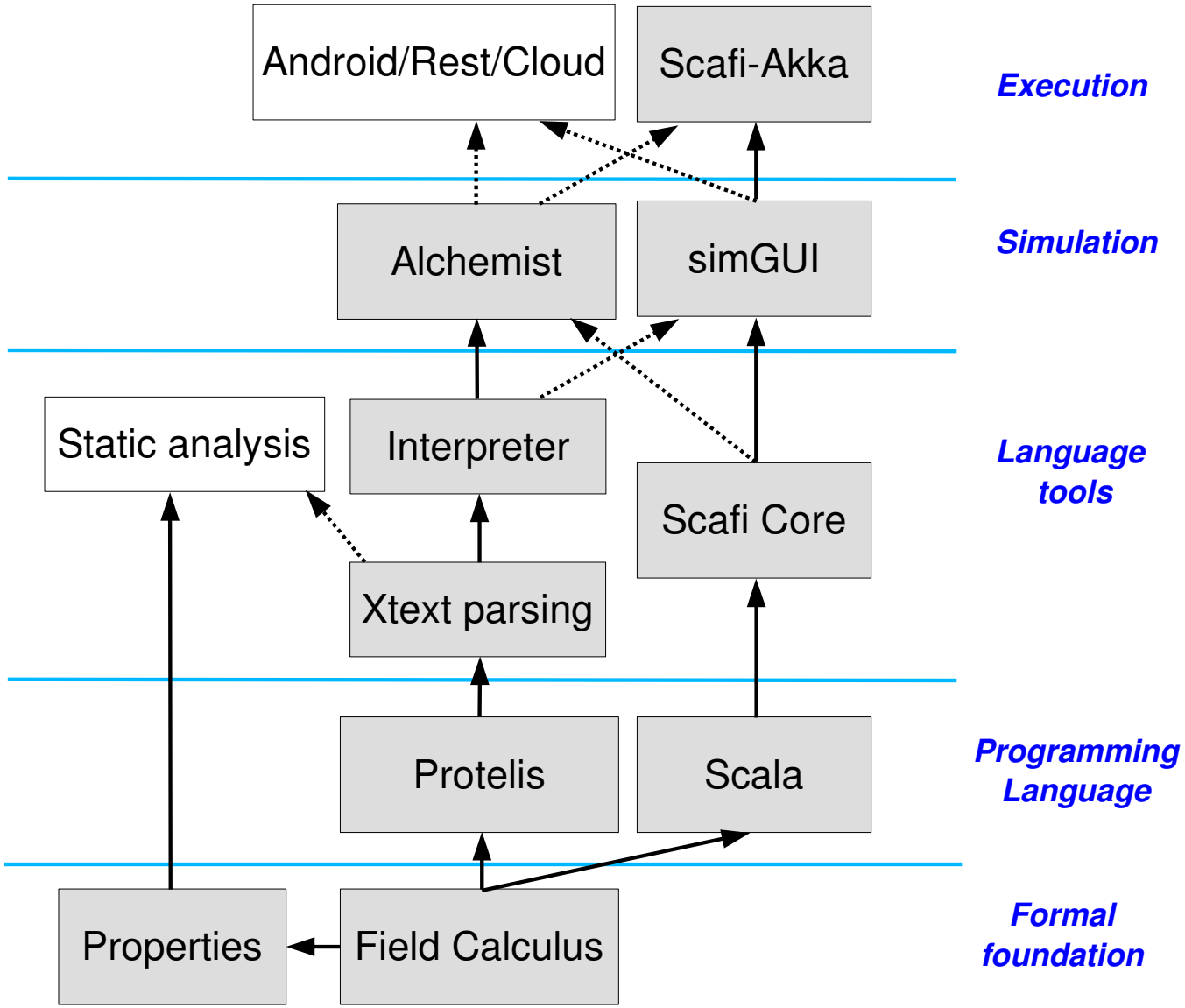
Protelis-based: rather consolidated, hard to use

- an evolution of an existing language/simulator: Proto
- a DSL interpred by the JVM, and Java-integrated
- Alchemist as simulator

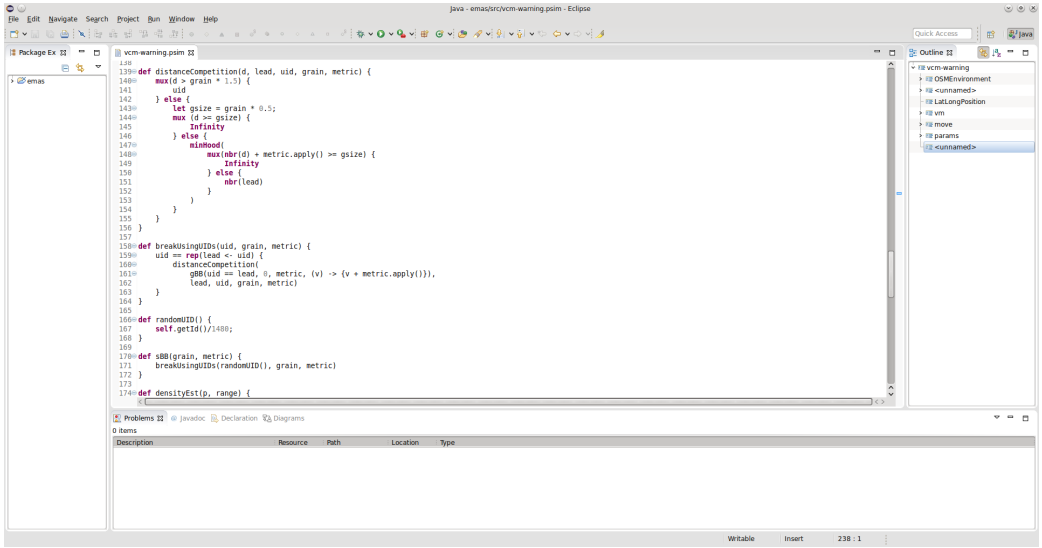
Scafi-based (Scala fields): growing, simpler for the Scala developer

- a stack fully-integrated with the Scala programming language and the JVM
- language as a Scala internal-DSL, i.e., as API
- prototype simulator for demos
- Akka-based platform support

Current tool-chain for aggregate computing



Protelis + Alchemist [SAC-2015]



Protelis language: `http://protelis.org/`

- Field calculus in disguised and full-blown version
- Java-like syntax and Java API integration

Alchemist simulator: `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

Outline

The Scala Programming Language

Scala as general-purpose, multi-paradigm JVM language

- Java integration
- Smooth integration of OOP and FP
- Expressive type system (with type inference)
- Advanced features for library development

Scala in the real world

- Increasing adoption in the industry
- Used by: Twitter, Amazon, LinkedIn, Netflix, Coursera, ...
- Killer apps: Akka, Apache Spark, Play...
- It is becoming the de-facto standard for building distributed systems

Scala for Library Development, Scala for field calculus

Features

- Typical FP features: first-class functions, lambdas, closures, HOFs, laziness, ...
- Traits and mix-in composition
- Advanced typing: self-types, abstract types, ...
- Generic programming support
- Implicit system
- Syntactic sugar, useful to support DSL

Scala and field calculus

- Both are functional
- Both have values, structures, lambdas, function definitions
- Scala can smoothly add advanced type system features

$\Rightarrow$ Scala seems the perfect framework for fields!

The ScaFi core language/API

```
1  trait Constructs {
2    // the unique idenifier of the local device
3    def mid(): ID
4
5    // applies fun to the previous result, or to init at first call
6    def rep[A](init: A)(fun: (A) => A): A
7
8    // evaluation of expr at the currently-considered neighbour
9    def nbr[A](expr: => A): A
10
11   // accumulates available evaluations of expr, with acc/init
       monoid
12   def foldhood[A](init: => A)(acc: (A,A)=>A)(expr: => A): A
13
14   // splits computation: th where cond is true, el everywhere/time
       else
15   def branch[A](cond: => Boolean)(th: => A)(el: => A): A
16
17   // perception of local sensor
18   def sense[A](name: LSNS): A
19
20   // perception of neighbourhood sensor
21   def nbrvar[A](name: NSNS): A
22   ...
```

How-to to run Scafi simulations

Requirements on installed frameworks:

- Java 8 installed
- IntelliJ with Scala support (if want to write new code)
- sbt build tool (`www.scala-sbt.org/`)
- git (`https://git-scm.com/`)

Running the simulator: long way

- Cloning the repository: `https://bitbucket.org/scafiteam/scafi`
- Checkout the `develop` branch
- Run `sbt`, compile and test
- Import existing sources (as SBT) in IntelliJ

The short way

- get the demos jar here:

  `https://bitbucket.org/scafiteam/scafi/downloads/scafi-demos-assembly-0.1.0.jar`

How-to to run your own Scafi simulations

Writing and testing specifications

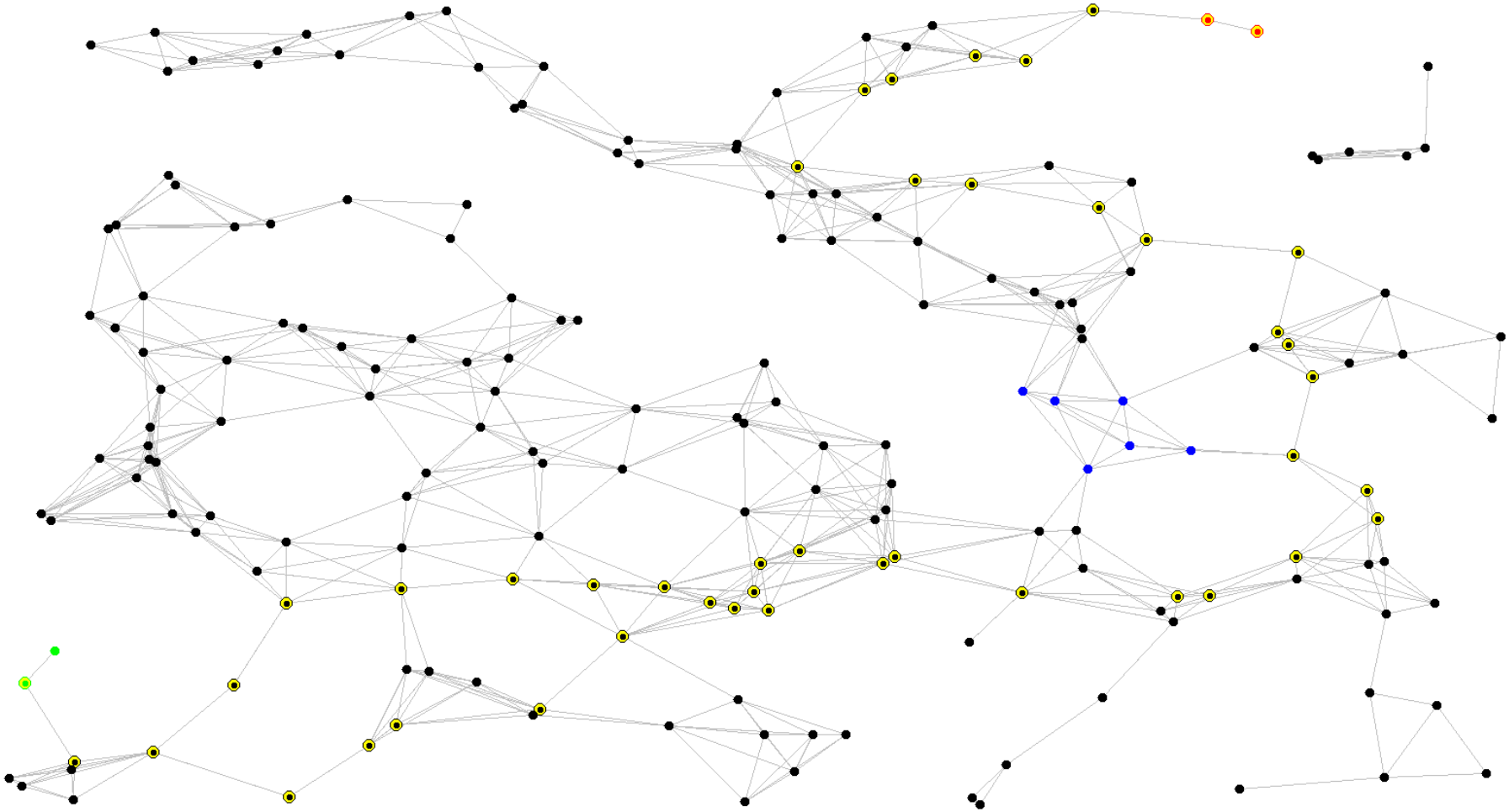
- Add, e.g. in the demos module in sims package, the following launcher class

```
1  package sims
2
3  import it.unibo.scafi.simulation.gui._
4  import it.unibo.scafi.simulation.gui.model._
5
6  object MyDemo extends Launcher {
7    // Configuring simulation
8    Settings.Sim_ProgramClass = "sims.MyDemo" // starting class, via Reflection
9    Settings.ShowConfigPanel = false          // show a configuration panel
10   Settings.Sim_NbrRadius = 0.15             // neighbourhood radius
11   Settings.Sim_NumNodes = 100               // number of nodes
12   launch()
13 }
14
15 class MyDemo extends AggregateProgram {
16   override def main() = rep(0)(_+1)         // the aggregate program
17 }
```

The Channel case in ScaFi – using power-block G

```
1  trait SensorDefinitions { self: AggregateProgram =>
2    def sense1 = sense[Boolean]("sens1")
3    def sense2 = sense[Boolean]("sens2")
4    def sense3 = sense[Boolean]("sens3")
5    def nbrRange = nbrvar[Double]("nbrRange") * 100
6  }
7
8  trait BlockG { self: AggregateProgram with SensorDefinitions =>
9
10   def G[V: OrderingFoldable](source: Boolean)(field: V)(acc: V => V)(metric: => Double = nbrRange): V =
11     rep((Double.MaxValue, field)) { case (d,v) =>
12       mux(source) { (0.0, field) } {
13         minHoodPlus { (nbr{d} + metric, acc(nbr{v})) }
14       }
15     }._2
16
17   def distanceTo(source: Boolean): Double =
18     G(source)(0.0)(_ + nbrRange)()
19
20   def broadcast[V: OrderingFoldable](source: Boolean, field: V): V =
21     G(source)(field)(v=>v)()
22
23   def distanceBetween(source: Boolean, target: Boolean): Double =
24     broadcast(source, distanceTo(target))
25 }
26
27 class Channel extends AggregateProgram  with SensorDefinitions with BlockG {
28
29   def channel(source: Boolean, target: Boolean, width: Double): Boolean =
30     distanceTo(source) + distanceTo(target) <= distanceBetween(source, target) + width
31
32   override def main() = branch(sense3){false}{channel(sense1, sense2, 1)}
33 }
```

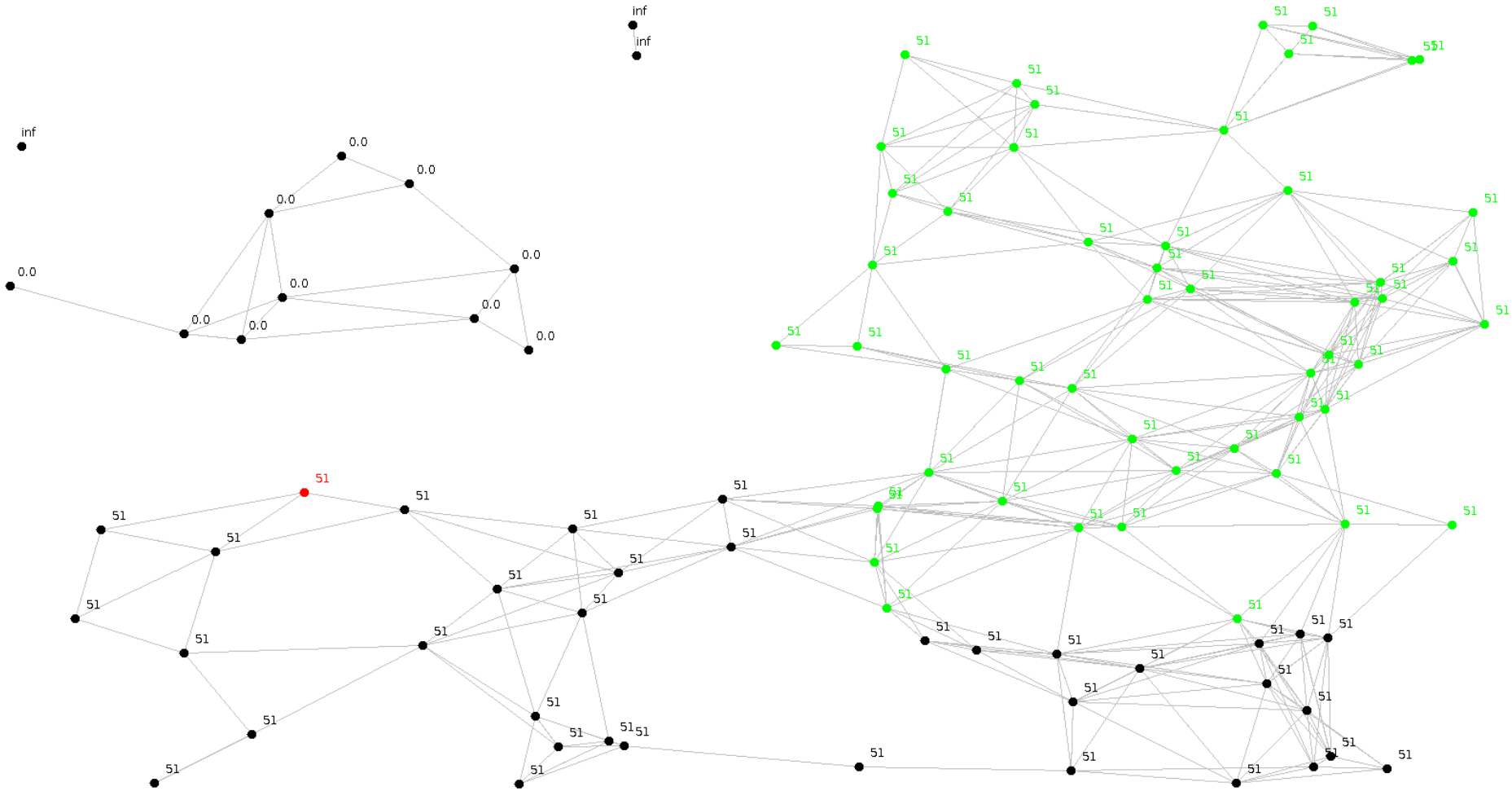
Simulation snapshot



Distributed Collection in ScaFi – using power-block C

```
1  trait BlockC { self: AggregateProgram with SensorDefinitions =>
2
3    def smaller[V: OrderingFoldable](a: V, b: V):Boolean =
4      implicitly[OrderingFoldable[V]].compare(a,b)<0
5
6    def findParent[V:OrderingFoldable](potential: V): ID = {
7      mux(smaller(minHood { nbr(potential) }, potential)) {
8        minHood { nbr { (potential, mid()) } }._2
9      } {
10       Int.MaxValue
11     }
12   }
13
14   def C[V: OrderingFoldable](potential: V, acc: (V, V) => V, local: V, Null: V): V = {
15     rep(local) { v =>
16       acc(local, foldhood(Null)(acc) {
17         mux(nbr(findParent(potential)) == mid()) {
18           nbr(v)
19         } {
20           nbr(Null)
21         }
22       })
23     }
24   }
25 }
26
27 class Collection extends AggregateProgram with SensorDefinitions with BlockC with BlockG {
28
29   def summarize(sink: Boolean, acc:(Double,Double)=>Double, local:Double, Null:Double): Double =
30     broadcast(sink, C(distanceTo(sink), acc, local, Null))
31
32   override def main() = summarize(sense1, _+_, if (sense2) 1.0 else 0.0, 0.0)
33 }
```

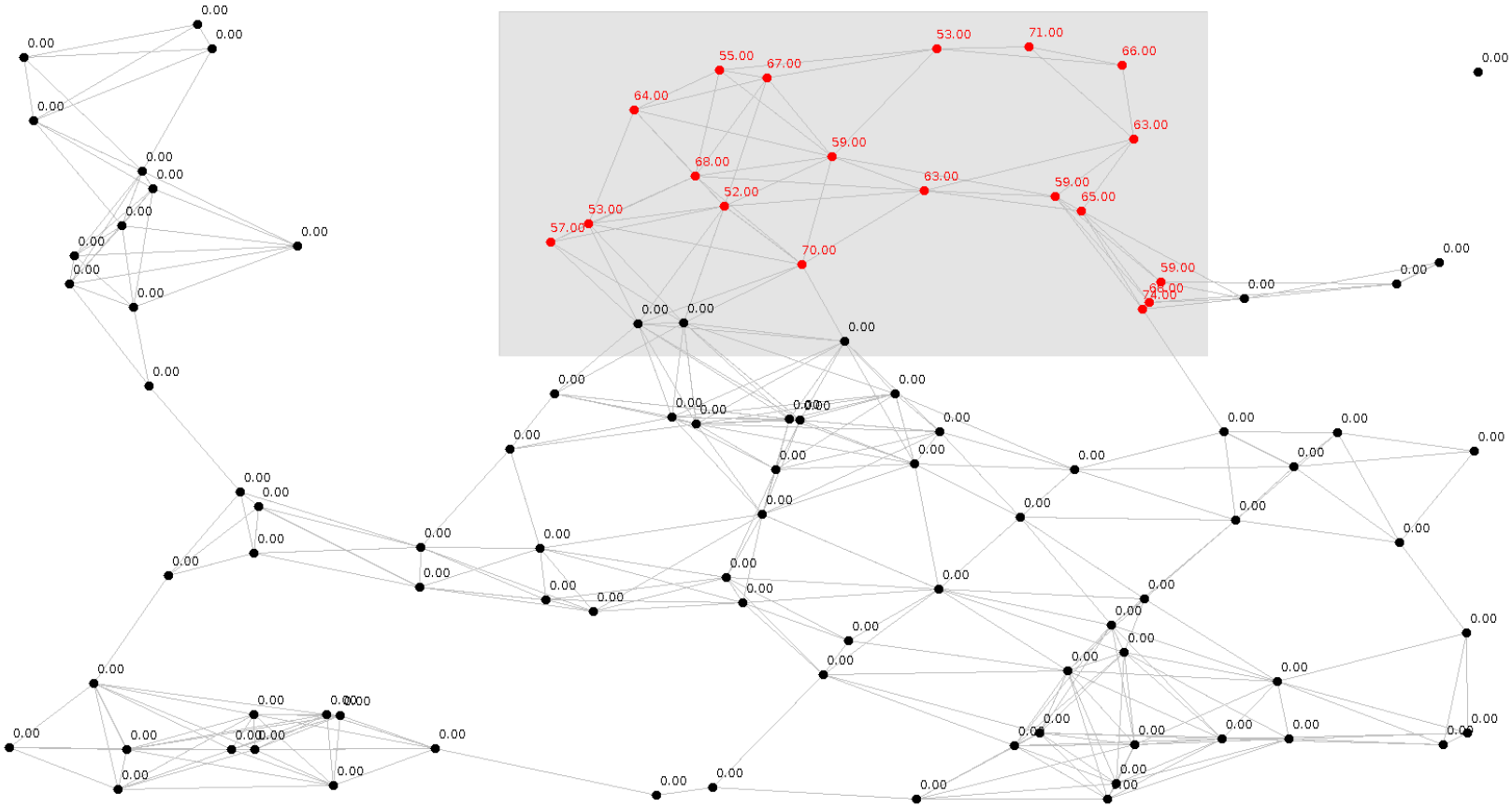
Simulation snapshot



The Timer case in ScaFi – using power-block T

```
1  trait BlockT { self: AggregateProgram with SensorDefinitions =>
2
3    def implicitMin[V: Numeric](a:V, b:V): V = implicitly[Numeric[V]].min(a,b)
4
5    def implicitMax[V: Numeric](a:V, b:V): V = implicitly[Numeric[V]].max(a,b)
6
7    def T[V: Numeric](initial: V)(floor: V)(decay: V => V): V = {
8      rep(initial) { v => implicitMin(initial, implicitMax(floor, decay(v))) }
9    }
10
11   def linearFlow(time: Double): Double =
12     T(time)(0.0)(v => v-1)
13
14   def timer(time: Double): Boolean =
15     linearFlow(time) == 0.0
16 }
17
18 class SimpleTimer extends AggregateProgram with SensorDefinitions with BlockT {
19
20   override def main() = branch(sense1){linearFlow(100)}{0}
21 }
```

Simulation snapshot



Outline

Akka platform for scala

Goal: easy configuration and execution of aggregate systems

- Setup of an aggregate application
- Setup of individual nodes/subsystems

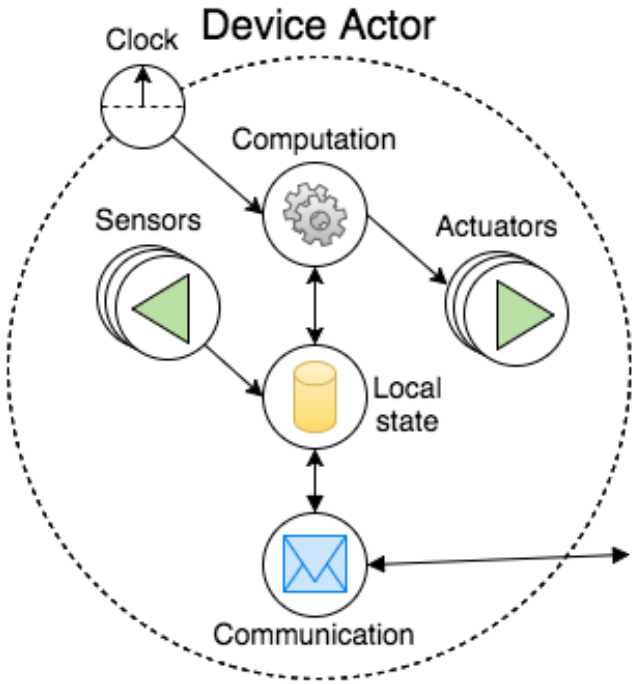
Support for multiple architectural styles and system configurations

- Fully peer-to-peer
- Server-based (mediating communications)

Implemented on top of the Akka actor framework

- An actor is a (re)active entity that
  - represents an independent locus of control,
  - encapsulates a state and behaviour,
  - has a global UID that allows for location transparency, and
  - interacts with other actors via asynchronous message passing (each actor has a mailbox for buffering incoming messages).
- scafi defines actors for different system components: devices, sensors, schedulers, servers (topology managers).
- scafi exposes an object-oriented facade API but also allows for interaction with the underlying actors.

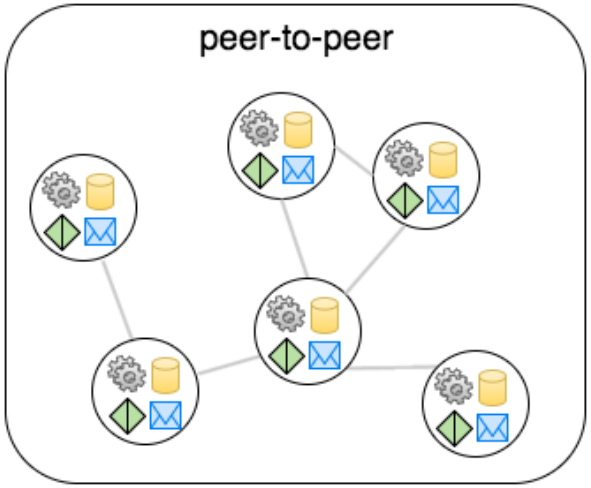
(Fully operational) Device actors

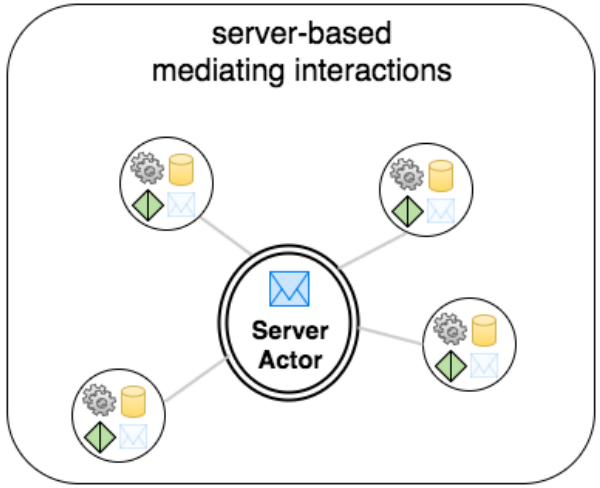


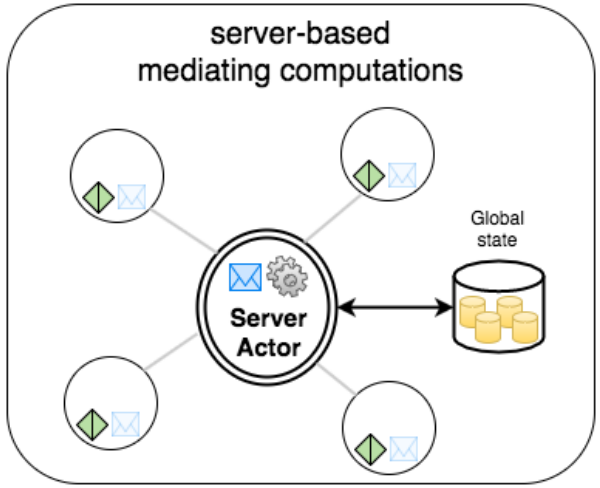
Responsibilities (each assigned to an actor)

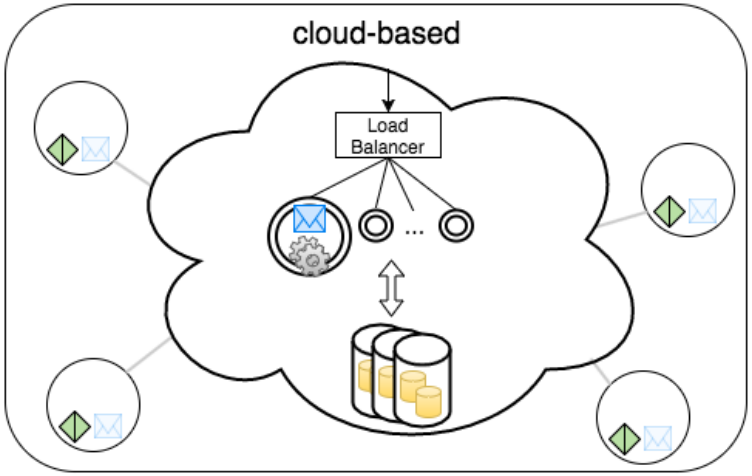
(i) Sensing/actuating
(ii) Aggregate program execution
(iii) Broadcasting/receiving the exports of computation to/from neighbours
(iv) Storing state (last computation, exports from neighbours)

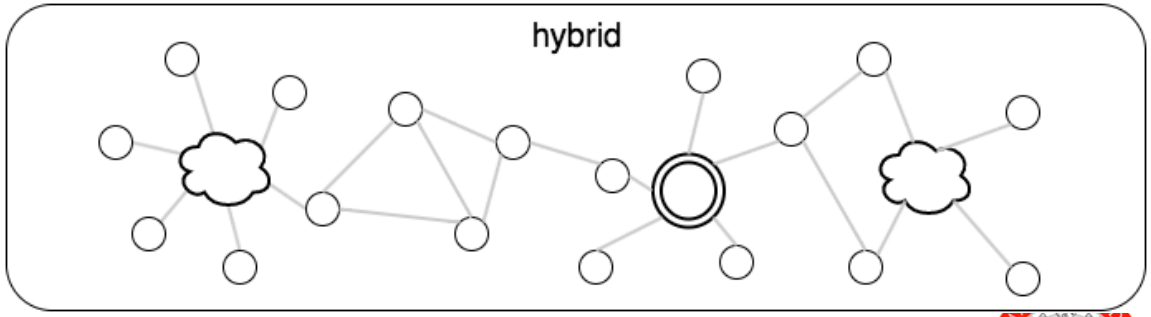
Execution strategies and architectures [UBICOMPW2016]





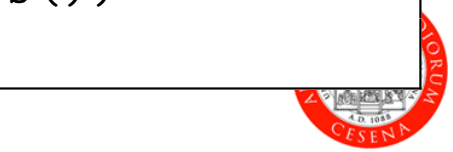






Setup of an (actor-based) aggregate system

```
1  // STEP 1: CHOOSE INCARNATION
2  import scafi.incarnations.{ BasicActorP2P => Platform }
3  import Platform.{AggregateProgram,Settings,PlatformConfig}
4
5  // STEP 2: DEFINE AGGREGATE PROGRAM SCHEMA
6  class Program extends AggregateProgram with CrowdAPI {
7    // Specify a "dangerous density" aggregate computation
8    override def main(): Any = dangerousDensity()
9  }
10
11 // STEP 3: PLATFORM SETUP
12 val settings = Settings()
13 val platform = PlatformConfig.setupPlatform(settings)
14
15 // STEP 4: NODE SETUP
16 val sys = platform.newAggregateApplication()
17 val dm = sys.newDevice(id = Utils.newId(),
18                        program = Program,
19                        neighbours = Utils.discoverNbrs())
20 val devActor = dm.actorRef // get underlying actor
```



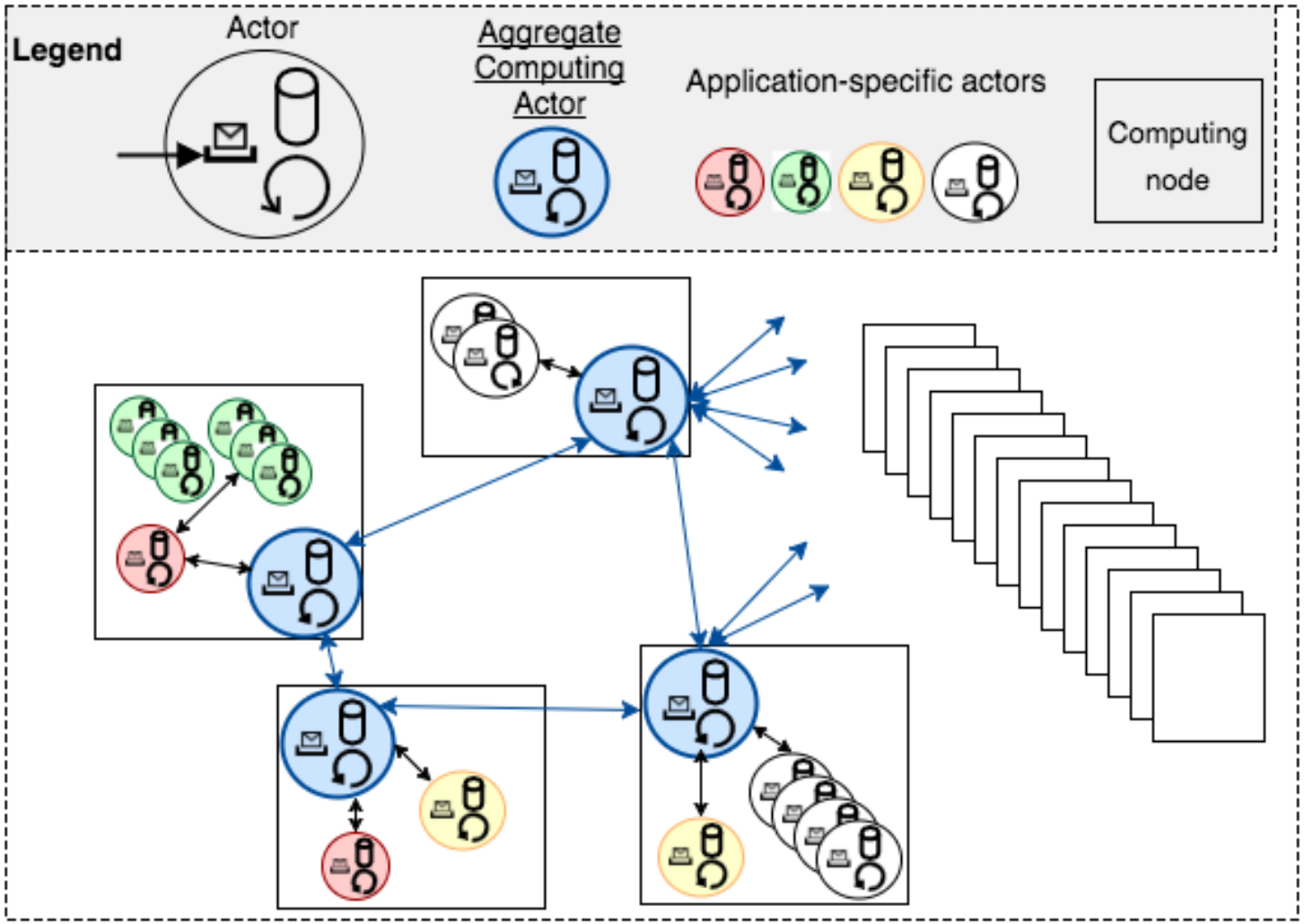
Embedded actors

On actors and distribution..

- The Actor model is generally considered appropriate for modelling & development of distributed systems.
- It comes naturally to adopt it in the context of distributed situated systems and those found in the IoT sc1enario.
- An *embedded* actor can work as the bridging abstraction that adds support for situatedness on top of plain actors.
- In other words, an embedded actor can be the *software interface to a sensor, an actuator, a processor, or a device* (the logical boundary comprising possibly many of the previous elements)

How can actors be used to implement complex decentralised behaviours?

Integrating Aggregate Computing with the Actor Model I



Integrating Aggregate Computing with the Actor Model II

Key idea

- An aggregate computing system can be seen, in a larger system, as a specific part that provides advanced capabilities such as robust coordination, resilient adaptation, and collective computation.
- In particular, aggregate computing actors can integrate with application-specific actors.
- This integration can be achieved in multiple ways:
  - Application actors as sensors for device actors
  - Application actors as managers of the aggregate platform (e.g., fine-tune neighbourhood)
  - Application actors as observers of device actors
  - Device actors as actuators for application logic

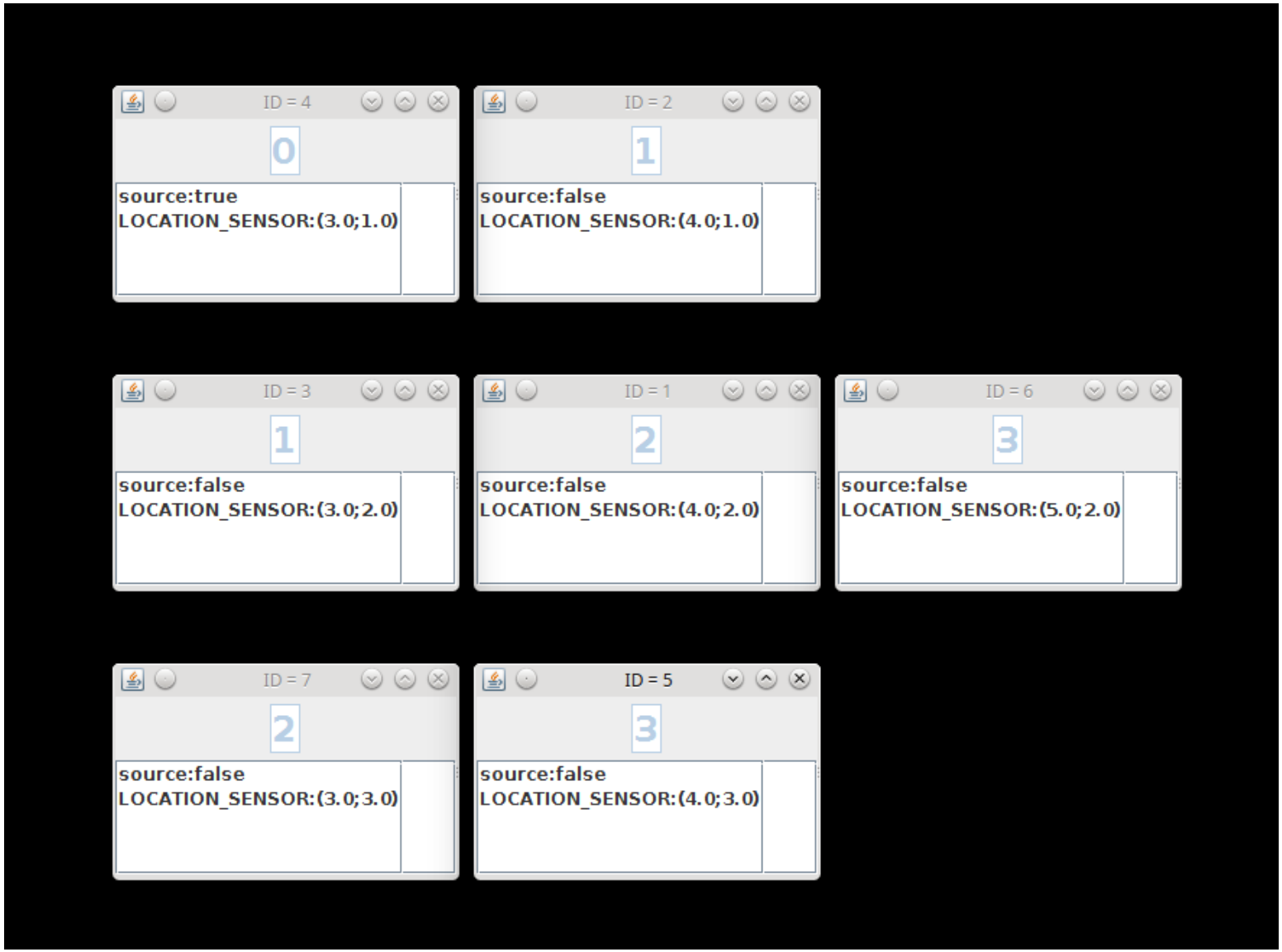
Example run in Scafi

```
Demo3_ServerMain -h 127.0.0.1
                 -p 9000
                 --sched-global rr
Demo4_MainProgram --program "demos.Demo3_AggregateProgram"
                  -P 9000 -p 9001
                  --sched-global rr -e 1;3 --gui
Demo4_MainProgram --program "demos.Demo3_AggregateProgram"
                  -P 9000 -p 9002
                  --sched-global rr -e 2;4 --gui
```

Explanation

- a central server at local host, computing a gradient, coordinating devices
- two programs, each running 2 actors (i.e., 2 devices) each
- the devices directly show themselves as GUI frames

A distributed GUI



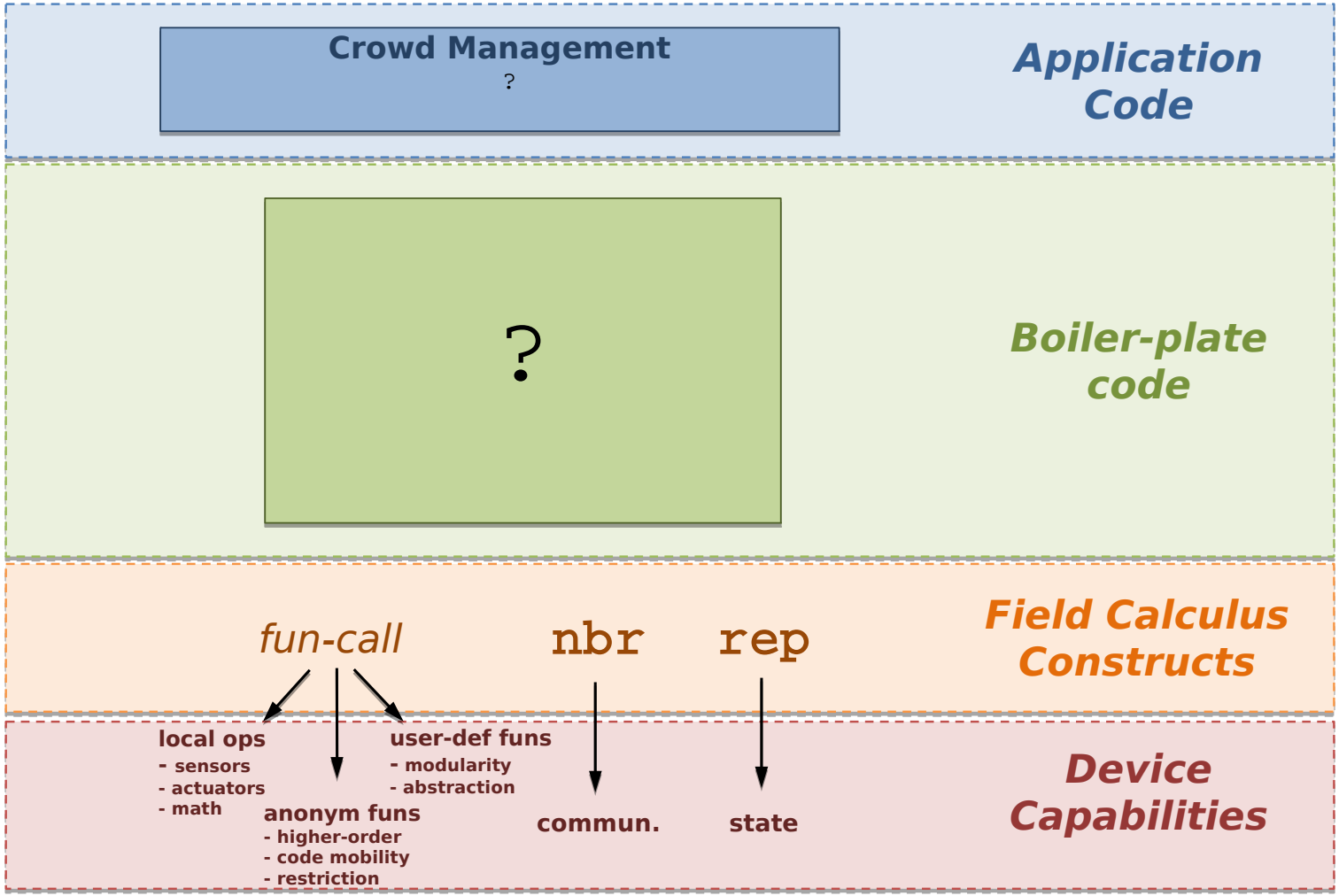
Outline

Field calculus, is it expressive?

Practically, we can express:

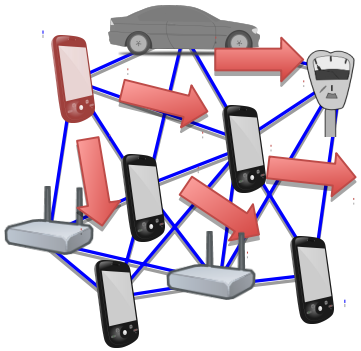
- complex spreading / aggregation / decay functions [IEEE Computer 48(9), 2015]
- spatial leader election, partitioning, consensus [SCP2015]
- distributed spatio-temporal sensing [COORD2016]
- splitting in parallel independent subprocesses [COORD2016]
- dynamic deployment/spreading of code (via lambda) [Damiani & Viroli & Beal & Pianini, FORTE2015]
- implicit/explicit device selection of what code execute [PRIMA2015]
- "collective teams" forming based on the selected code [PRIMA2015]

How to engineer complex apps out of basic constructs?

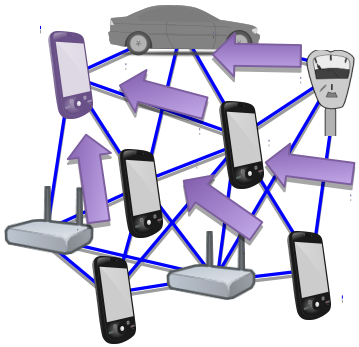


GCT as a combinator set

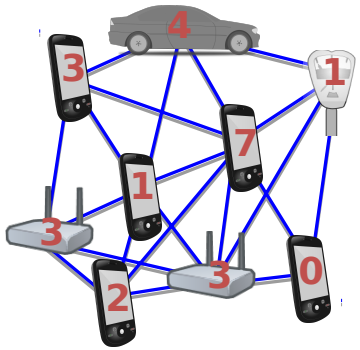
G: spreading



C: collecting



T: time decay

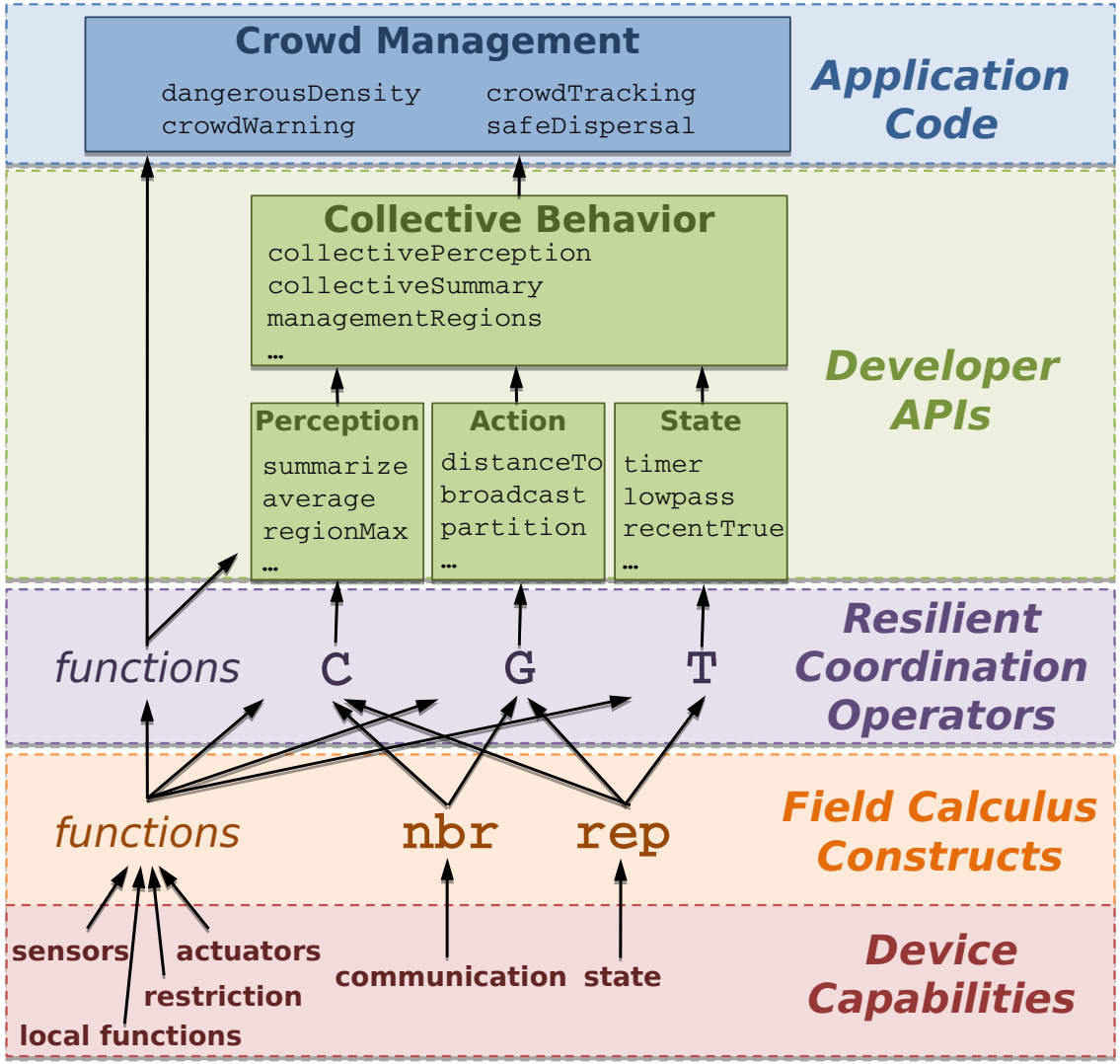


Functions

- `G`: Spreads and en-route computes information outwards a source
- `C`: Collects and en-route aggregates information inwards a destination
- `T`: Locally iterates computations (with termination)

```
;; Obtaining various functions on top of solely G,C and T
def distance (source) { G(source,0,()=>nbrRange(),(x)=>x+nbrRange()) }
def broadcast (source, value) { G(source,value,()=>nbrRange(),(x)=>x) }
def sum (source, field) { broadcast(source,C(source,distance(source),+,0)) }
def timer (timeout) { T(timeout,0,(t)=>t-dt()) == 0}
```

Building blocks supersede **rep** and **nbr** in designing APIs



Crowd estimation service, on top of APIs [Fruin, 1971]

```
;; 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
  }
}
```

Resilience with aggregate computing

Vocabolary definition

The ability to adapt well in the face of adversity

In aggregate computing

The ability of a functional component to properly work in the face of changes to inputs and environment

Three notions of resilience considered so far

1. resilience to occasional changes: self-stabilisation
2. resilience to device distribution: eventual consistency
3. resilience to ongoing changes: dynamics control

Other notions for future work

- resilience to change of computational infrastructure, to varying requirements on computational load, to malicious attacks..

Resilience with aggregate computing

Vocabolary definition

The ability to adapt well in the face of adversity

In aggregate computing

The ability of a functional component to properly work in the face of changes to inputs and environment

Three notions of resilience considered so far

1. resilience to occasional changes: self-stabilisation
2. resilience to device distribution: eventual consistency
3. resilience to ongoing changes: dynamics control

Other notions for future work

- resilience to change of computational infrastructure, to varying requirements on computational load, to malicious attacks..

Resilience with aggregate computing

Vocabolary definition

The ability to adapt well in the face of adversity

In aggregate computing

The ability of a functional component to properly work in the face of changes to inputs and environment

Three notions of resilience considered so far

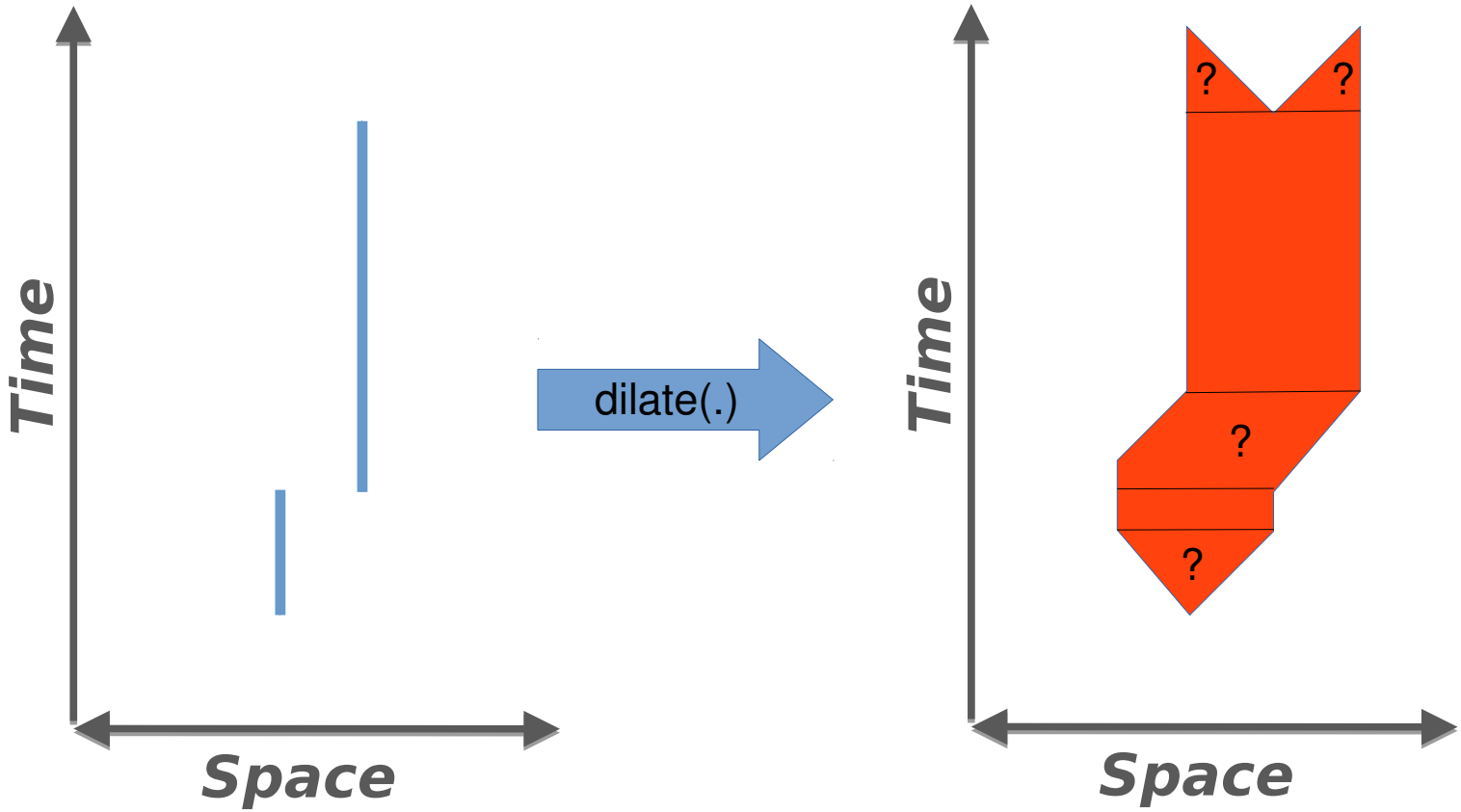
1. resilience to occasional changes: self-stabilisation
2. resilience to device distribution: eventual consistency
3. resilience to ongoing changes: dynamics control

Other notions for future work

- resilience to change of computational infrastructure, to varying requirements on computational load, to malicious attacks..

Outline

Seeking a notion of self-stabilisation



Self-stabilisation for computational fields [SASO 2015]

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 space problem

$\Rightarrow$ occasional env. changes do not affect the result of computation

$\Rightarrow$ we can somehow get rid of time and evolution in design

Self-stabilisation is undecidable, but can identify sufficient conditions

Self-stabilisation for computational fields [SASO 2015]

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 space problem

$\Rightarrow$ occasional env. changes do not affect the result of computation

$\Rightarrow$ we can somehow get rid of time and evolution in design

Self-stabilisation is undecidable, but can identify sufficient conditions

The fragment of self-stabilising field expressions

Syntax (note composability, and 3 constraint patterns on `rep`)

$$
\begin{array}{rcll}
\mathtt{s} & ::= & \ell \mid \mathtt{x} \mid (\mathtt{s}\ \overline{\mathtt{s}}) \mid (\mathtt{nbr}\ \mathtt{s}) \mid (\mathtt{if}\ \mathtt{s}\ \mathtt{s}\ \mathtt{s}) & \\
 & \mid & (\mathtt{rep}\ \mathtt{x}\ \mathtt{w}\ (\pi^{\mathsf{MB}}\ \mathtt{x}\ \overline{\mathtt{s}})) & \mathtt{x} \notin \mathbf{FV}(\overline{\mathtt{s}}) \\
 & \mid & (\mathtt{rep}\ \mathtt{x}\ \mathtt{w}\ (\pi^{\mathsf{F}}\ \mathtt{s}^{\mathsf{A}}\ (\mathtt{nbr}\ (\mathtt{s}\ \mathtt{x}))\ \overline{\mathtt{s}})) & \mathtt{x} \notin \mathbf{FV}(\mathtt{s}, \overline{\mathtt{s}}, \mathtt{s}^{\mathsf{A}}) \\
 & \mid & (\mathtt{rep}\ \mathtt{x}\ \mathtt{w}\ (\pi\ (\pi'\ (\mathtt{nbr}\ (\pi''\ \mathtt{x}\ \overline{\mathtt{s}}''))\ \overline{\mathtt{s}}')\ \overline{\mathtt{s}})) & \pi' \circ \pi = \pi^{\mathsf{MD}}, \pi'' \circ \pi' = \pi^{\mathsf{MBP}}, \mathtt{x} \notin \mathbf{FV}(\overline{\mathtt{s}}, \overline{\mathtt{s}}', \overline{\mathtt{s}}'')
\end{array}
$$

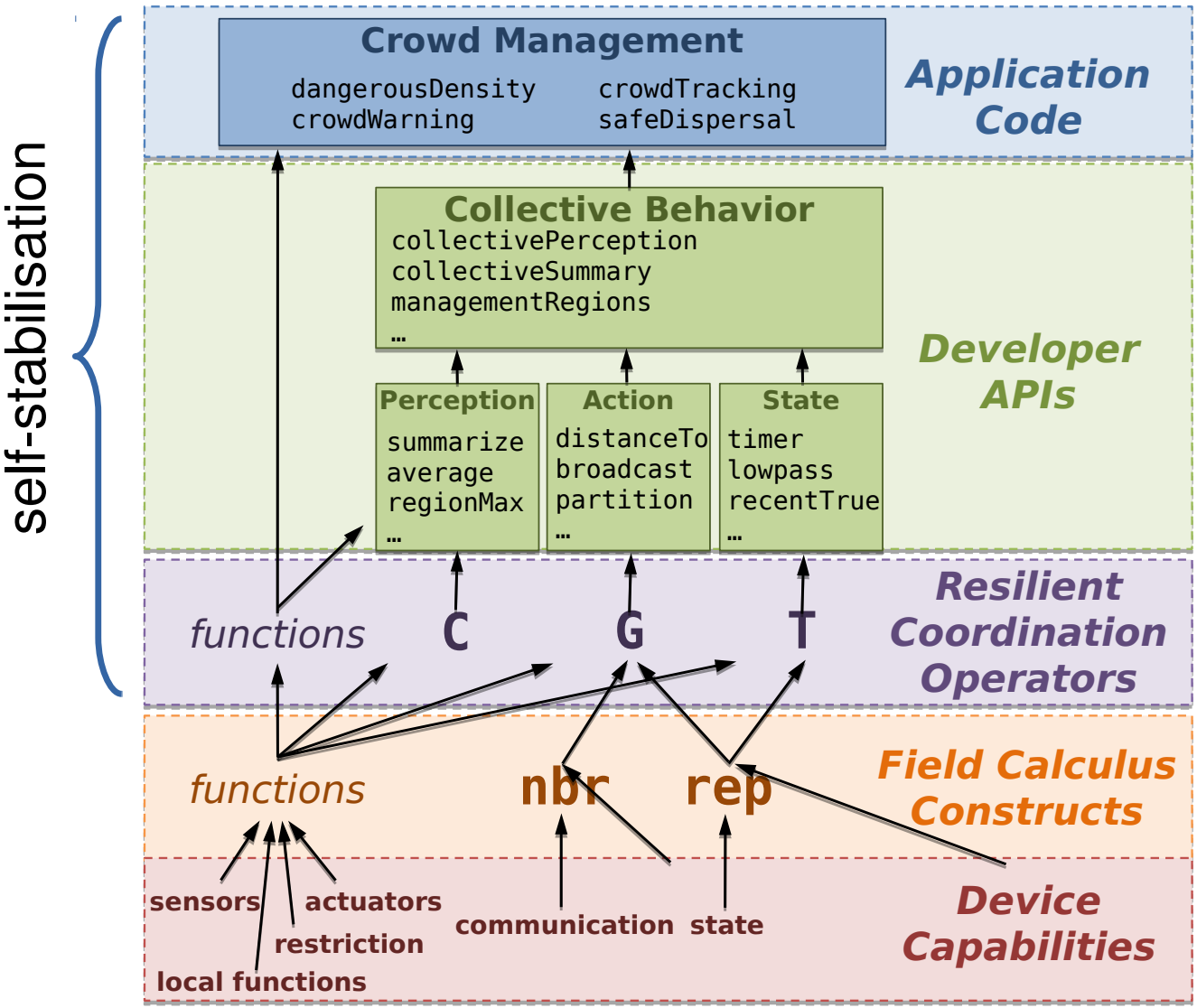
Key technical elements (see the paper for details):

- Function π^{B}: bounded
- Function π^{P}: progressive in first argument
- Function π^{M}: monotonic non-decreasing in first argument
- Function π^{D}: double bounded, with a LB due to first argument
- Function π^{F}: filter applying a projection based on first argument
- Expression s^{A}: a field modelling an acyclic topological relation

Main lemma

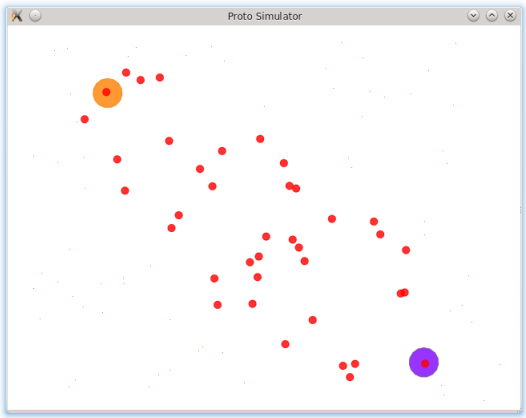
- G, C and T are self-stabilising, hence the same holds "on top"

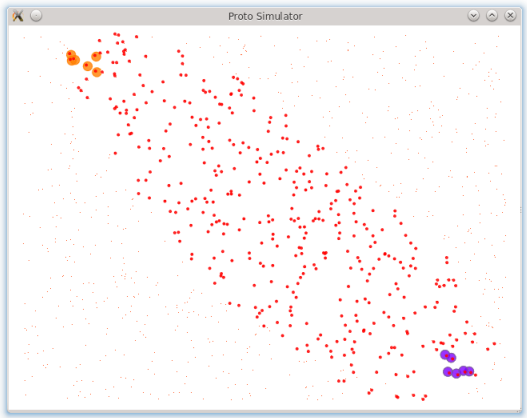
Self-stabilisation stack

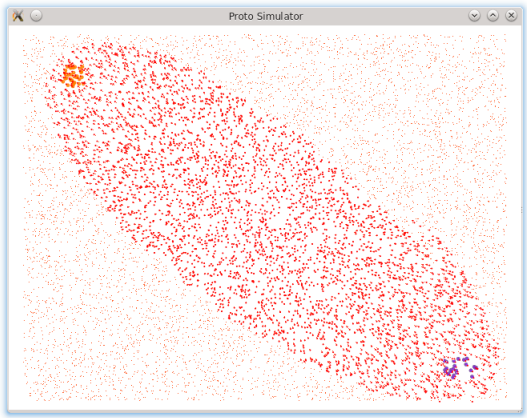


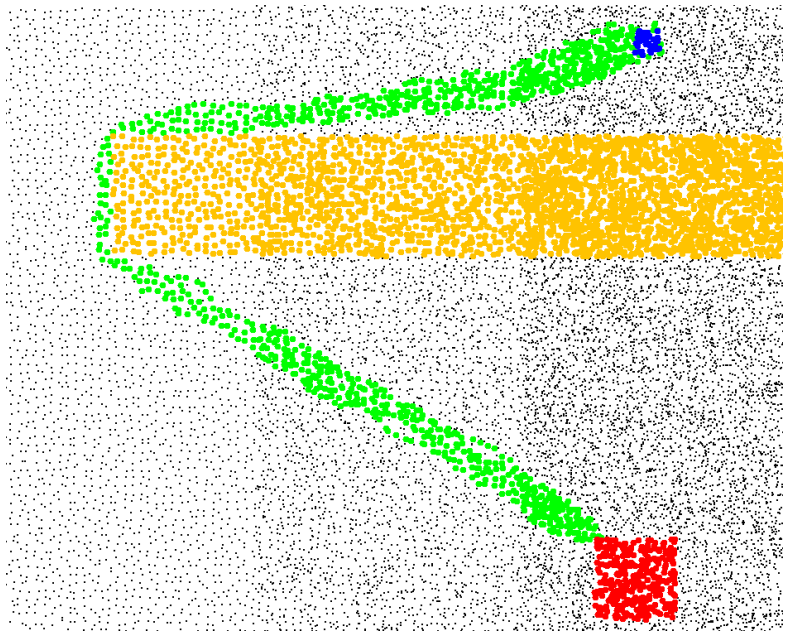
Outline

Eventual consistency

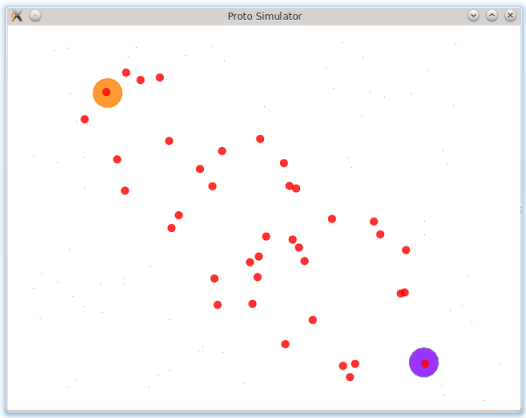


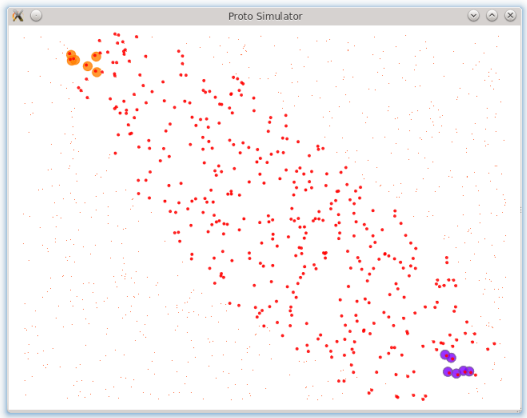


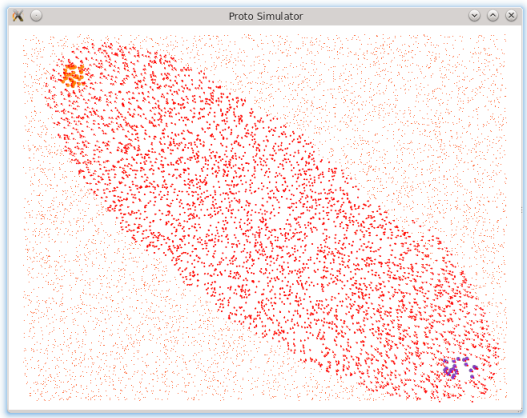


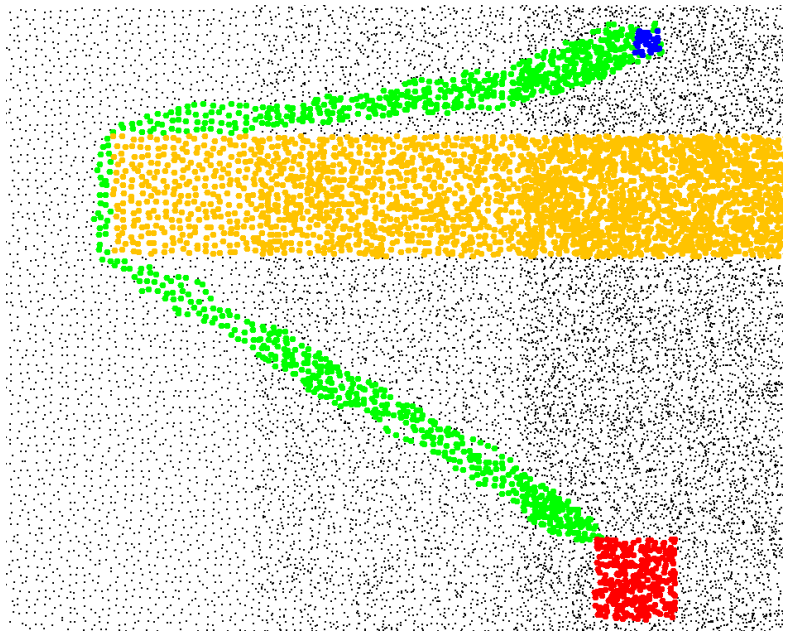


Eventual consistency









Eventual consistency [SASO 2016]

On network dependence

- How is self-stabilised state sensitive to small changes in dev. location?

Eventual consistency $\approx$ self-stabilisation + approximability

- Let M be a representative of a field domain D over a continuous space-time
- The computation self-stabilises and the final result (M_i, F_i) is the discrete approximation of a continuous model (M, F), as precise as the density of devices increases

$$\lim_{i \to \infty} |(M \cup M_i) - (M \cap M_i)| = 0 \qquad \lim_{i \to \infty} \int_{M \cap M_i} |F - F_i| = 0$$

$\Rightarrow$ Consequence 1: good behaviour on heterogeneous networks

$\Rightarrow$ Consequence 2: intrinsic scaling to high densities

$\Rightarrow$ Consequence 3: good predictability ($\phi_o \approx \Phi_e(\phi_i)$)

Eventual consistency [SASO 2016]

On network dependence

- How is self-stabilised state sensitive to small changes in dev. location?

Eventual consistency $\approx$ self-stabilisation + approximability

- Let M be a representative of a field domain D over a continuous space-time
- The computation self-stabilises and the final result (M_i, F_i) is the discrete approximation of a continuous model (M, F), as precise as the density of devices increases

$$\lim_{i \to \infty} |(M \cup M_i) - (M \cap M_i)| = 0 \qquad \lim_{i \to \infty} \int_{M \cap M_i} |F - F_i| = 0$$

$\Rightarrow$ Consequence 1: good behaviour on heterogeneous networks

$\Rightarrow$ Consequence 2: intrinsic scaling to high densities

$\Rightarrow$ Consequence 3: good predictability ($\phi_o \approx \Phi_e(\phi_i)$)

GPI calculus

$$e ::= x \mid v \mid e_\lambda(e_1, \ldots, e_n) \mid \texttt{GPI}(e_{\texttt{src}}, e_{\texttt{init}}, e_{\texttt{metric}}, e_{\texttt{integrand}})$$

Meaning of GPI

- Gradient Path Integral – essentially a variation of G
- From `source`, `initial` value gets propagated
- Direction of propagation given by minimal paths according to `metric`
- During propagation we do path integral of `integrand` field

$\Rightarrow$ Result of GPI satisfies approximability

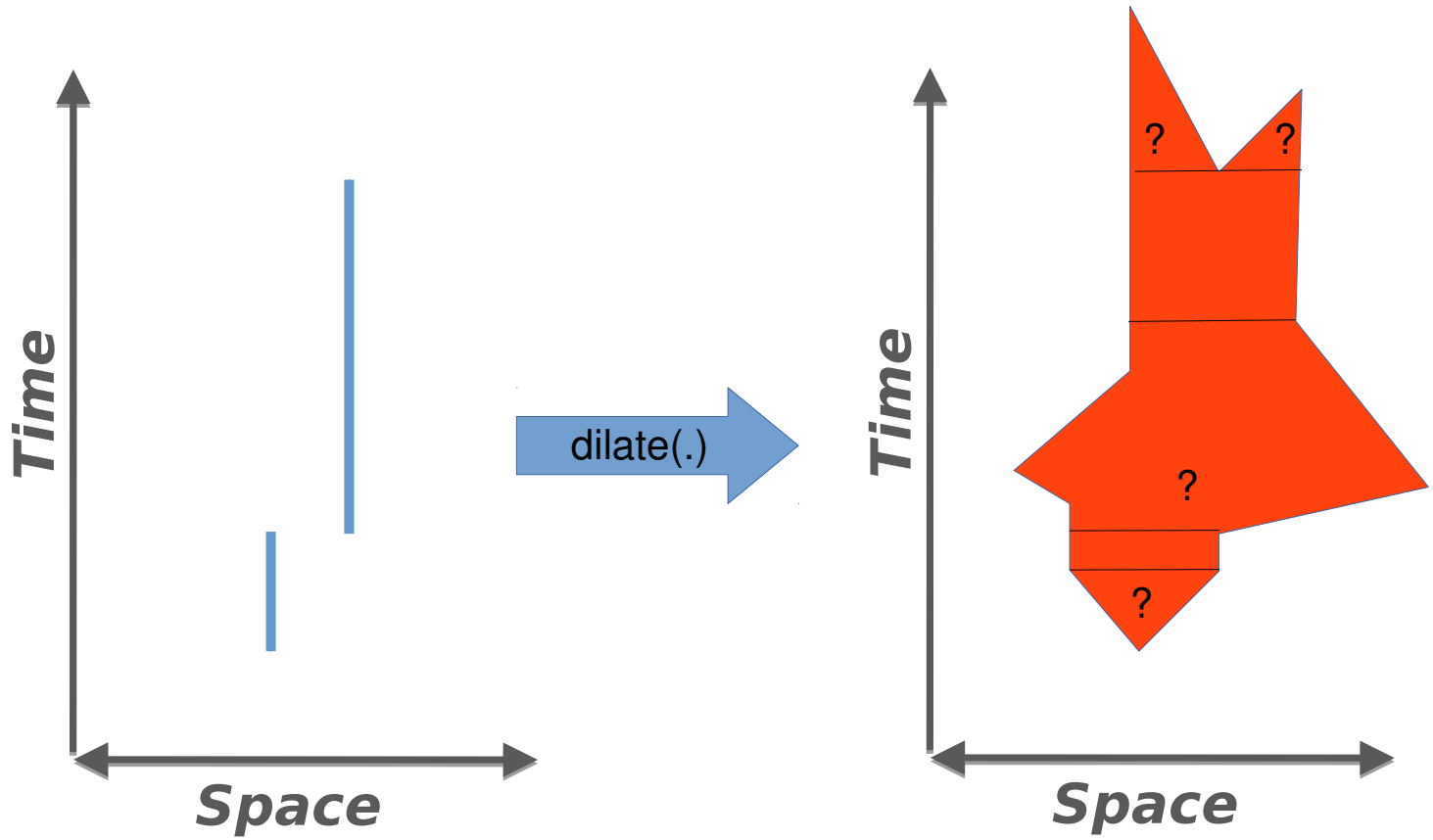
```
;; Obtaining various functions on top of GPI
def distance (source) { GPI(source,0,1,1) }
def broadcast (source, value) { GPI(source,value,1,0) }
```

Additional caveat

"Boundary" values due to non-continuous functions need to be confined

Outline

A bad dynamics for dilate..



Controlling dynamics

A very hard problem in general

- How is self-stabilisation reached?

$\Rightarrow$ sometimes we need this to happen quickly

$\Rightarrow$ sometimes we need this to happen smoothly

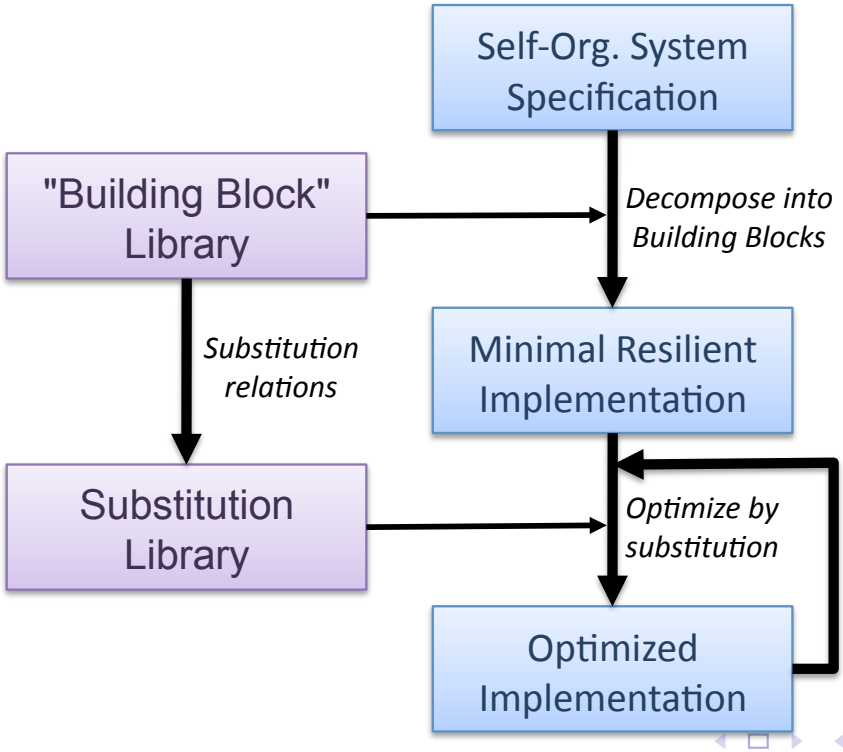
State of research

- no formal understanding of the entire problem
- A – proposed substituted GCT for ad-hoc situations
- B – proposed meta-techniques to improve self-stabilisation

A methodological workflow [SASO-2015]

Key idea:

- GCT-set is conceived to be very general
- Specialised and optimised replacements of G, C and T can be designed
  - one just needs to check they are I/O-equivalent
- Selectively using them can improve overall performance



Specialised alternative implementations of GCT

Alternate G: Flex gradient [Beal, SAC2009]

- Prioritises calculation of direction of slope wrt correct distances

$\Rightarrow$ better choice for steering applications

Alternate C: multi-path calculation

- Divides data and re-collects along multiple paths, not just the shortest

$\Rightarrow$ slower but less fragile, works better in dynamic scenarios

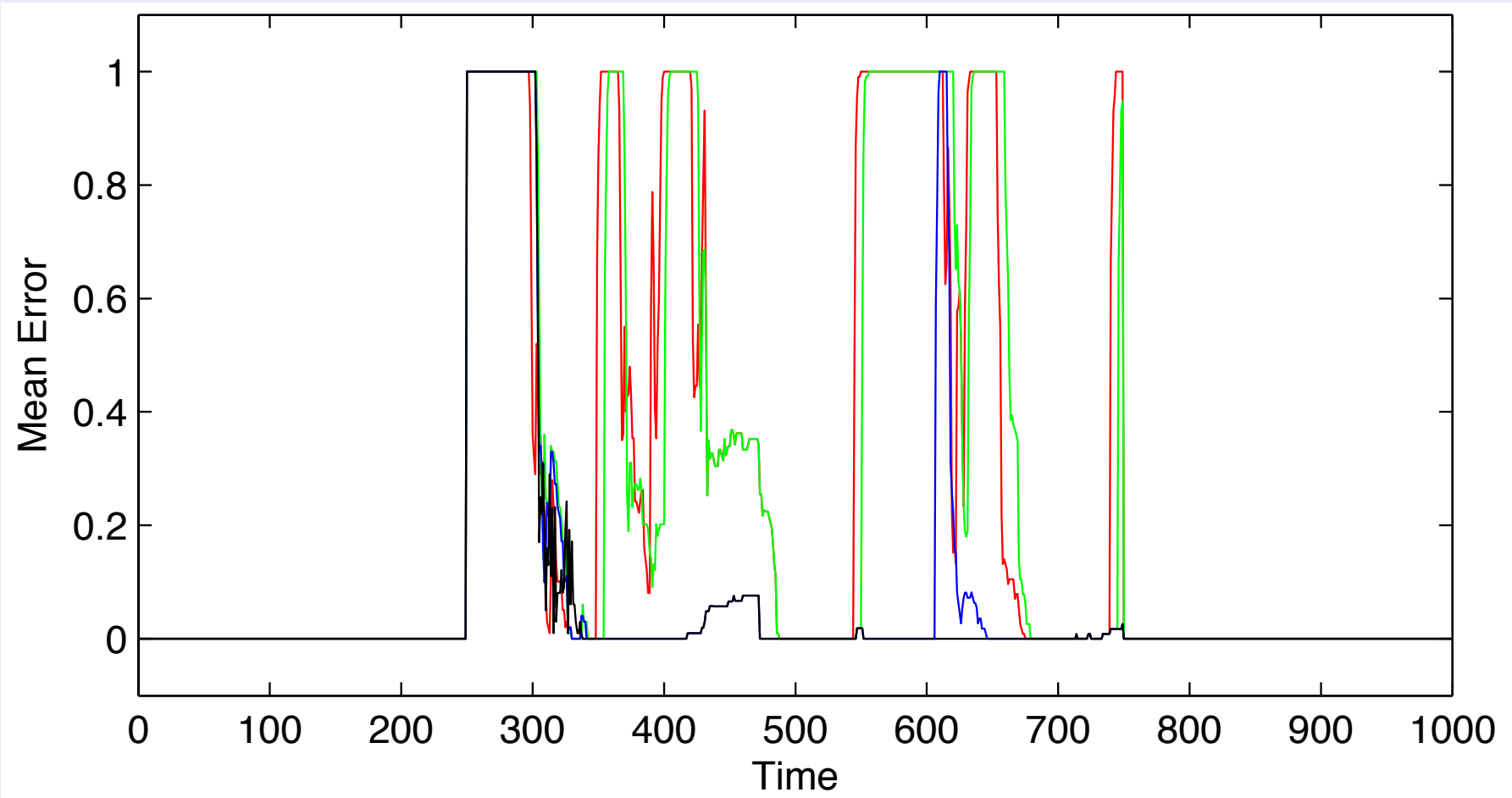
Alternate T: exponential filter

- When tracking data, it can help to apply a low-pass filter

$\Rightarrow$ allows one to trade off smoothness and response rate

Substituting GCT in evacuation scenario

Global error in reacting to an alert, with different GCT sets



GCT (red), GCT' (green), GC'T' (blue), G'C'T' (black)

Substitution meta-technique [COORD2016]

Problem

- some algorithms reach self-stabilisation in uneffective way
- other algorithms even do not self-stabilise in certain specific cases

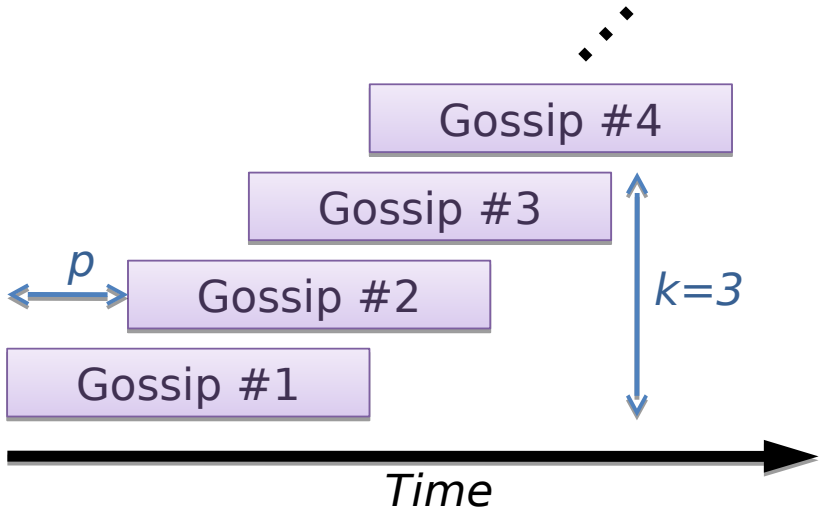
```
def gossip(f,x) { ;; Gossips value of x, using f as binary accumulator
  rep(v <- x) { ;; Declare state variable v, initialized to current value of x
    ;; Every round, merge x with neighbors' values of v
    f(x, hood(f, x, nbr(v)));
} }

;; Example run: gossip minimum of devices' unique id
gossip((a,b) -> min(a,b), id())
```

Facts about gossip

- optimisation on **rep**, **nbr** and **hood** are possible, essense is the same
- cannot repair to increase of the minimum value sensed

Time-replicated gossip (TR-Gossip)



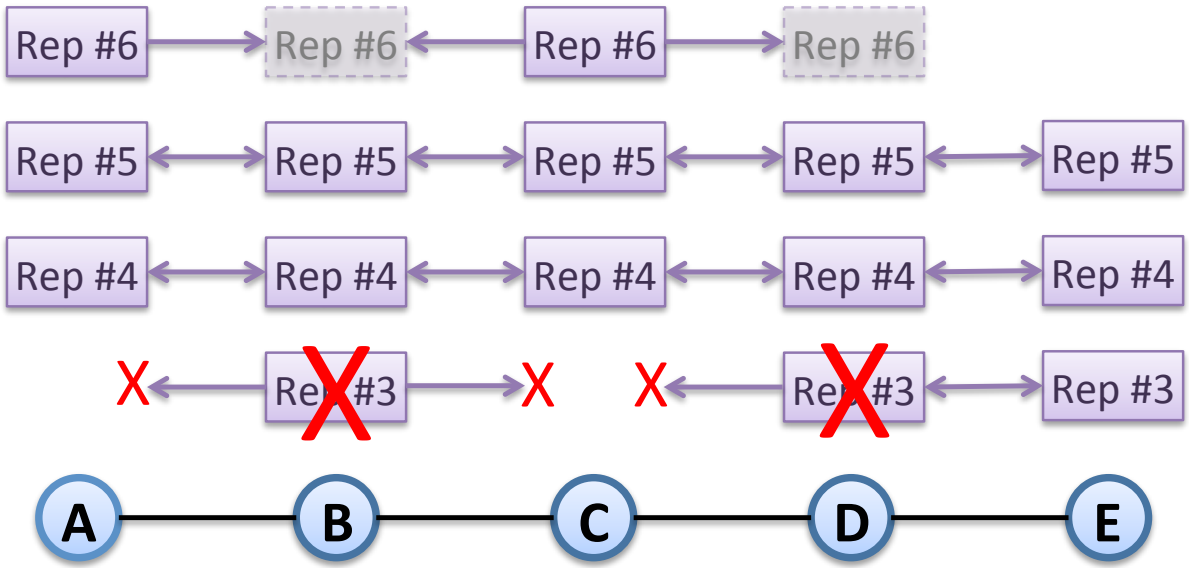
Idea: run k isolated instances of gossip, once each p time units

- a new instance appearing makes the k^{th} oldest disappear
  - at some point: $(G_o, G_{o+1}, \ldots, G_{o+k-1}) \rightarrow (G_{o+1}, \ldots, G_{o+k-1}, G_{o+k})$
- of replicate values, the oldest value is the one to use
- duration of replicas is key
  - too long $\Rightarrow$ higher delays, too short $\Rightarrow$ no convergence
- as duration is settled, can adjust adaptation/cost tweaking k and p

Field calculus implementation

```
def timeReplicated(process, default, p, k) {
  rep([]) { ;; list of [replicateID, value] pairs
    (procs) =>
    keepTail(k,
      alignedMap(
          mergeAfter(procs,[sharedTimer(p,procs), default]),
          (replicateID, value) => { process() }))}}
}
```

Visualisation of replicates creation/disposal



Scenario ($k = 3$)

- Devices A and C have independently started replicate #6
- Instances of #6 are spreading/merging new instances to other devices
- The arrival of replicate #6 deletes replicate #3 and blocks its spread

Replicated gossip at work on `min`

actual replicated gossip

Basic simulation scenario (`min`, `avg`, `and`): we consider `min` here

- 100 devices, unit disc proximity (15 neighbours on average)
- devices move by Lévi walks (full walk in ≈ 100 rounds)
- each condition tested with 40 simulations of 300 rounds each
- values: s1:[2L,4R,D1]; s2:[6L,3R,D50], s3:[2L,4R,D1]

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!

Acknowledgments

- Jacob Beal (BBN, USA)
- Ferruccio Damiani, Giorgio Audrito (UNITO)
- Danilo Pianini, Roberto Casadei (UNIBO)

References I

[IEEE Computer 48(9), 2015] Beal, J., Pianini, D, and Viroli, M. (2015).
Aggregate programming for the Internet of Things.
IEEE Computer, 48(9) 2015.

[Beal et.al., 2013] Beal, J., Dulman, S., Usbeck, K., Viroli, M., and Correll, N. (2013).
Organizing the aggregate: Languages for spatial computing.
In Mernik, M., editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global.
A longer version available at: `http://arxiv.org/abs/1202.5509`.

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

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

[Mamei et.al., 2009] Mamei, M. and Zambonelli, F. (2009).
Programming pervasive and mobile computing applications: The tota approach.
ACM Transactions on Software Engineering and Methodologies, 18(4).

[SAC-2015] Pianini, D., Beal, J., and Viroli, M. (2015).
Practical aggregate programming with PROTELIS.
In *ACM Symposium on Applied Computing (SAC 2015)*.
To appear.

References II

[Damiani & Viroli & Beal & Pianini, FORTE2015] Damiani, F., Viroli, M., Pianini, D., and Beal, J. (2015).
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.

[SASO-2015] Viroli, M., Beal, J., Damiani, F. and Pianini, D. (2015).
Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields
In *IEEE Conference on Self-Adaptive and Self-Organising Systems*.

[Beal, SAC2009] Beal, j. (2009).
Flexible self-healing gradients
In *ACM Symposium on Applied Computing*, pp. 1197–1201.

[UBICOMPW2016] Viroli, M., Casadei, R., and Pianini D. (2016).
On execution platforms for large-scale aggregate computing.
In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp '16, 1321–1326, New York, NY, USA, 2016. ACM.

[SCP2015] Damiani, F., Viroli, M., and Beal, J. (2015).
A type-sound calculus of computational fields
Science of Computer Porgramming (117)

[COORD2016] Pianini, D., Beal, J., and Viroli, M. (2016).
Improving Gossip Dynamics Through Overlapping Replicates
In *COORDINATION 2016, LNCS 9686*

References III

[PRIMA2015] Pianini, D., Beal, J., and Viroli, M. (2016).
Multi-agent Systems Meet Aggregate Programming: Towards a Notion of Aggregate Plan
In *PRIMA 2015, LNCS 9387*