Space-Fluid and Time-Fluid Programming

Chapter 6
Space-fluid and Time-fluid Programming

Danilo Pianini[1], Roberto Casadei[1], Stefano Mariani[2], Gianluca Aguzzi[1], Mirko Viroli[1], and Franco Zambonelli[2]

Abstract This chapter addresses coordination challenges in emerging application scenarios, such as cyber-physical systems (CPSs), the Internet of Things (IoT), and edge computing, through a Fluidware-based approach to field-based coordination, showing how data structures distributed in space and time (*fields*) can be extended towards *fluidity*. In time, we challenge the traditional assumption of fixed clocks regulating local activities and propose "causality fields" for scheduling in coordination with field-based methods. This innovative approach enables "time-fluid" coordination, balancing system reactivity and resource usage. We formalize the scheduling framework in the field calculus, provide a reference implementation, and assess its effectiveness through simulations across diverse case studies. In space, we explore managing spatially varying signals in coordinated systems, employing decentralized and situated computing for collaborative adaptive sampling. Our algorithm dynamically partitions space into regions that adapt, creating a "fluid" virtualized space responsive to pressure from the underlying phenomenon under observation. Proven self-stabilizing and locally optimal, our adaptive sampling algorithm enables spatially adaptive computations with a tunable trade-off between accuracy and efficiency.

Key words: reactive field-based programming, causality, spatial sampling, coordination, self-stabilisation

Università degli Studi di Bologna, Cesena, Italy
e-mail: \{danilo.pianini,roby.casadei,mirko.viroli\}@unibo.it · Università di Modena e Reggio Emilia, Reggio Emilia, Italy
e-mail: \{stefano.mariani,franco.zambonelli\}@unimore.it

6.1 Introduction

luidity is the property of fluids, namely substances that can flow easily and have the tendency of assuming the shape of their container. Viscosity is its antonym, which somewhat involves a resistance to change. Therefore, fluidity is a matter of *adaptiveness* in abstract space, and viscosity is where adaptation is difficult or hindered (e.g., by the need for manual intervention, lack of context awareness, or rigid architecture). What can it mean for computations to be fluid? In this chapter, we reply with a notion of computations that should move, with little resistance, where they are most needed and where they can be most effective and efficient for the application and constraints/preferences at hand. We thus cover two main kinds of computational fluidity that are investigated in the FluidWare project: *time fluidity* [37], namely the ability to adapt computation across time (e.g., to make it denser or sparser in specific time intervals, according to application needs); and *space fluidity* [9], namely the ability to adapt computation across space (e.g., by suitably distributing it across systems situated in environments where phenomena to track or react to are also spatially situated).

More specifically, regarding time, approaches to self-organisation [23, 11, 28] like field-based coordination [34] often assume that device computations are regulated by a periodic clock, configured on a per-application basis and dictating a basic frequency of activity. In this chapter, we challenge this assumption, and propose a *reactive* approach where scheduling follows the paths of change propagation. In particular, we enable *programming* of the scheduling in terms of *causality fields*, each providing a distributed notion of a computational "cause" (why and when a field computation has to be locally computed) and how it should change across time and space. Starting from low-level platform triggers, such causality fields can be organised into multiple layers, up to defining high-level, collectively-computed time abstractions, to be used at the application level. This reinterpretation of the traditional view of time in terms of articulated causality relations allows us to express what we call "time-fluid" coordination, where scheduling can be finely tuned so as to select the triggers to react to, generally allowing to adaptively balance application performance (cf. system reactivity) and efficiency (cf. costs and resource usage).

Regarding space, we focus on a recurrent problem in pervasive computing systems, namely managing (i.e. estimating, predicting, or controlling) environmental signals that vary in space. Nodes can locally sense, process, and act upon signals, and coordinate with neighbours to implement distributed collective strategies for the estimation of a spatial phenomenon through collaborative adaptive sampling of environmental data. Our proposal leverages the idea of dynamically partitioning space into competing regions that grow or shrink to provide accurate global-level sampling. Indeed, such regional partition defines a sort of virtualised space that is "fluid", as its shape adapts in response to pressure forces exerted by the underlying phenomenon. We provide an adaptive sampling algorithm in the field-based coordination framework, also proving it is self-stabilising and locally optimal, and showing it provides a means for fostering a tuneable trade-off between accuracy and efficiency.

The remainder of the chapter is organised as follows. Section 6.2 discusses previous research on adaptive computations across space and time; Section 6.3 and Section 6.4 present our field-based approach to time- and space-fluidity, respectively; and Section 6.5 provides concluding remarks.

6.2 State of the art

Time fluidity relates computation adaptation to time, whereas space fluidity relates computation adaptation to space. In this section, we report on relevant previous research about time and space in distributed and pervasive computing systems.

6.2.1 Time and synchronisation in pervasive computing systems

Many algorithms exist for processing data in a distributed way, that is by a set of distributed processes [14]. However, when such data dynamically vary with time, processing must be periodically recomputed, which introduces the problem of identifying the appropriate frequency for computations. Most solutions typically adopt a predefined frequency [14, 7], but this strategy does not take into account that such frequency should be properly tuned to match the dynamics of the data being processed. Also, synchronising processes to let them organise processing in successive rounds is far from easy in asynchronous systems lacking a shared physical clock [12, 10]. Our proposal starts from similar problems, but is specifically conceived for distributed field-based data processing, and accounts for the strict relations between the spatial and temporal dimensions that exist in situated computations.

Specific timing problems also arise in the area of wireless sensor networks. There, acquiring a globally shared notion of time (as accurate as possible) is of fundamental importance [29] to capture accurate snapshots of the distributed phenomena under observation. However, global synchronisation also serves energy saving purposes. In fact, when not monitoring or not communicating, the nodes of the network should go to sleep to avoid energy waste, but this implies that to exchange monitoring information with each other they must periodically wake-up in a synchronised way. Several proposals exist for adaptive synchronisation in wireless sensor networks [1, 15, 13], dynamically changing the sampling frequency (and hence frequency of communication rounds) so as to adapt to the dynamics of the observed phenomena. Besides sensor networks, the issue of adaptive sampling has recently landed in the broader area of IoT systems and applications [32], again with the primary goal of optimising energy consumption of devices while not losing relevant phenomena under observation. However, in these contexts, such optimisations typically take place in a centralised (cloud) [31] or semi-decentralised (fog) way [17], which again disregards spatial issues and the strict space-time relations of phenomena.

6.2.2 Space-aware computing and adaptive spatial sampling

Switching to considering the spatial dimension, we are interested in approaches that can adapt data processing to some spatial measure, for instance the density or extent of the phenomenon under observation—e.g. air pollution requiring denser sampling in regions with more variance, and more sparse sampling in those with less variance. A few research works pursue similar objectives to the one we aim at with our notion of space fluidity.

Topology Adaptive Spatial Clustering [35] is a distributed algorithm that partitions a Wireless Sensor Network (WSN) into a set of disjoint sampling clusters, with no prior knowledge of cluster number or size (like ours), by encoding geographical distance, connectivity, and deployment density information in a single measure upon which leader election (for cluster heads) happens. The goal is to group together nodes in proximity and within regions of similar deployment density, to improve efficiency of data aggregation and compression. Besides the focus on efficiency, that is only half the story in our approach (the other being accuracy, hence the trade-off), two are the fundamental differences our approach has with respect to this: first, in our case adaptation is domain dependent, in the sense that it depends on some property (e.g. variance) of the phenomenon under observation, not on infrastructural properties; second, we do also consider over-sampling when variation is high, whereas reference [35] only considers under-sampling where measures are redundant.

Another distributed approach to adaptive spatial sampling [19] is based on the assumption that neighbouring sampling nodes usually have similar readings (i.e. high spatial correlation), hence can be grouped in a cluster to improve energy consumption. The proposed algorithm uses such spatial correlation, two application-specific threshold parameters (error tolerance and correlation range), and residual energy to elect cluster heads, with the goal of minimising the number of clusters, and the variance of their size. However, most of the calculations still rely on infrastructural properties rather than measures about the phenomenon of interest, and the focus is, once again, on energy saving, thus, authors do not consider over-sampling but solely under-sampling.

In [18], an approach is proposed that controls adaptation based on the information observed by sensors, rather than on network, energy, or other infrastructural aspects. However, they consider a special case where the clusters must correspond to pre-determined "sources of interest", such as physical objects/devices in the environment. Moreover, clusters are formed with the secondary objective of being balanced, whereas we allow them to be of different sizes and shapes (and even encourage to be so depending on the spatial distribution of the phenomenon of interest).

SILENCE [16] is a distributed, space-time adaptive sampling algorithm based on (space-time) correlation of measured values. Its main goal is efficiency: indeed, SILENCE strives to reduce communication and processing overhead, by minimising sampling redundancy and reducing the scheduling speed of sensor devices. However, it focusses on the case where spatial correlation of sampled values is high.

Finally, the *ASample* algorithm [30] considers fully-distributed domain-driven adaptation, also accounting the opportunity to over-sample, too. In particular, ASample builds a Voronoi tessellation of the area where the WSN is deployed, in a fully distributed way by considering only neighbourhood data. Such a tessellation promotes a desired application-level sampling accuracy: while a given Voronoi region is within the accuracy bound, it keeps expanding; on the contrary, whenever the accuracy constraint is violated, a virtual centroid of a novel Voronoi region is spawned, with a value that is obtained through interpolation of the neighbouring regions. This is a key aspect to consider, as it introduces synthetic data, which is something we avoid as we only increase sampling granularity if there are actual devices available in the target area. However, ASample assumes that the smaller the area covered by the Voronoi region, the less representative the samples drawn are, hence the smaller the impact on the global sampling accuracy. In our targeted scenarios, the opposite could be true, too: smaller regions represent sharp variance of measurements across space, and more accurately represent the irregularities of the underlying phenomenon. Indeed, in our proposal, we are fine that potentially irregular clusters reflect the irregular spatial distribution of the observed phenomenon.

6.3 Fluidity in time

This section presents a model for time-fluid field-based coordination, emphasising the integration of a *causality field* within the field calculus framework [4]. The central concept involves utilising field-based coordination to govern the dynamics of computations in application-level fields.

Our proposed approach is broadly applicable to any field-based coordination framework, with a specific focus here on the general field calculus framework for clarity. The execution of field calculus programs yields *computational fields*, mapping *events* (space-time points representing computation rounds) to computational values. Inter-event communication forms a directed acyclic graph known as an *event structure* [21]–see Figure 6.1 for a graphical example, modelling the interactions in a system of computing and interacting devices.

Locally, events are scheduled by the middleware platform, serving as a low-level, asynchronous, distributed "clock". This local event scheduling provides the foundation for defining higher-level, time-fluid concepts. As a practical example, in a landslide monitoring application, events correspond to sensor readings assessing terrain movement, and the computational field represents the probability of a landslide occurring in the next T minutes.

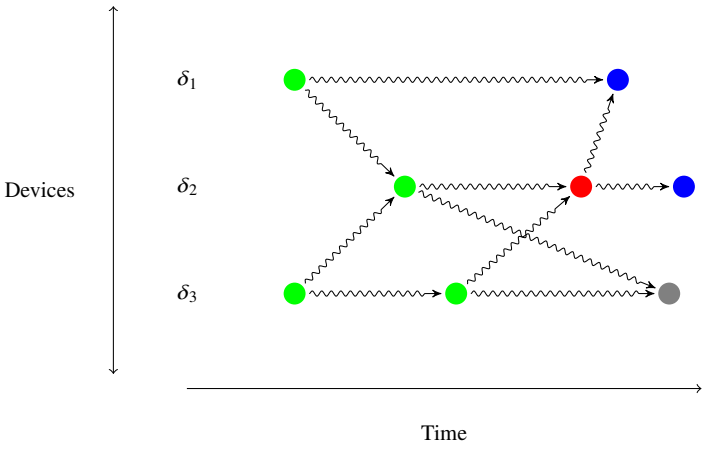


Fig. 6.1: Example of an event structure. Each node is an event (i.e., a computation round happening at a given device), and the curly arrows denote the causal relationship between events resulting from communication, which defines a partial order on the events. Given a reference event (red), through the $\rightsquigarrow$ relationship it is possible to define its "causal past" event cone (green), "causal future" event cone (blue), and concurrent events (gray). Conventionally, we assume that the events horizontally aligned to the device labels δ_i are rounds performed by those devices. Along the arrows, information can be transferred, modelling state persistence in a given device or communication between different devices: in this example, δ_1 and δ_3 both communicate bidirectionally with δ_2. Moreover, since devices might be situated in space, and the event structure captures a distributed, platform-level notion of time, the events can denote space-time locations and, together with the corresponding computed values, a space-time computational field (modelling, e.g., a temperature field, a warning field, a field of suggestions for crowd dispersal, etc.).

6.3.1 A Time-Fluid model

For a field calculus program **P**, each round involves processing valid messages from neighbours, $M \in \mathcal{M}$, and contextual information $S \in \mathcal{S}$ obtained via sensors. In our reference landslide monitoring scenario, S could include terrain movement measurements, and M messages from neighbouring sensors. The executing platform must decide whether to launch the next **P** evaluation round at a given event, providing valid values for $\mathcal{M}$ and $\mathcal{S}$. Multiple programs can be executed concurrently.

To support causality-driven coordination, we introduce a *(local event) trigger* associated with each event, representing the cause for the event. Triggers involve changes at the application level ($\mathcal{M}$) or physical level ($\mathcal{S}$), e.g., "new message arrived" or "sensor provides a new value". $\mathcal{T}$ denotes the set of possible local event triggers.

Additionally, we introduce a *guard policy*, denoted as G, acting as a scheduler for a "parent"field computation. A guard policy, expressed in the same language as **P**, is viewed locally as $f_G\colon (\mathcal{S}, \mathcal{M}) \to \mathcal{V}$. This allows multiple guard policies to be associated with a field computation, forming a *tree structure*. Platform triggers schedule low-level guard policies, forming the leafs, and higher-level policies use their outcomes to schedule subsequent evaluations.

Considering our landslide monitoring example, triggers may be sensor-related events, low-level guard policies could compute landslide probability based on sensor readings, and higher-level policies may visualise a landslide frontier. Feedback from a guard policy's evaluation can regulate its next evaluation, aiding in self-regulation. Refer to Figure 6.2 for a graphical representation.

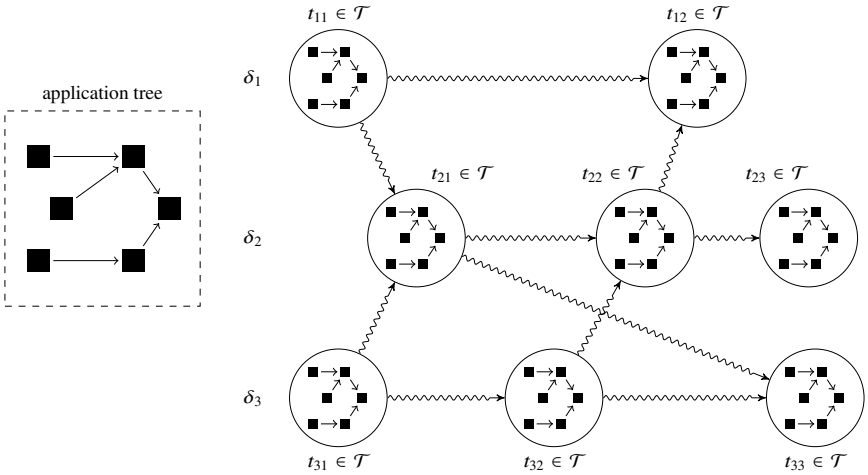


Fig. 6.2: This is an example of an event structure (structurally equivalent to that of Figure 6.1) where each event (circle) is an evaluation of a tree of field calculus programs (the tree of black squares shown in the dashed box on the left). The labels on the nodes indicate the platform triggers causing the events to happen. The application tree has a root program (the right-most node) and schedulers as children and within its sub-trees; the nodes with no incoming arrows are scheduled merely on the basis of platform triggers.

6.3.2 Causality Function and Temporal Dynamics

The scheduling impact of a guard policy is governed by the *causality function*, f_C, defined at the platform level. For a given round and program P, it takes the current round's trigger, the result of evaluating P at the previous round, and the results of

its n lower-level schedulers, returning a Boolean. This Boolean denotes whether P is scheduled for evaluation at the current round, creating a *causality field* that acts as a self-regulated clock.

In this framework, a hierarchy of schedulers processes low-level triggers, culminating in a top-level causality field that schedules application-level computation. This structured guard facilitates precise temporal control.

The model's expressivity is influenced by the ability to sense context ($\mathcal{S}$) and express event triggers ($\mathcal{T}$). The platform can generate events due to triggers involving changes to any $\mathcal{S}$ value, making the computation reactive to device perception. Triggers flipping from `false` to `true` can model timers, encompassing the classic time-driven approach as a special case.

6.3.3 Implications

The introduced model has significant implications for the expressiveness of field-based coordination.

6.3.3.1 Programming Time and Propagating Causality.

Enabling an application to influence its own execution policy allows programming time directly. Evaluating a field computation at different frequencies modulates the application's perception of time, impacting sensor sampling rates and communication dynamics across space. This consequence arises from adopting a causality-based notion of time, distinguishing this model from others.

6.3.3.2 Adapting to Causality.

The model's capability to influence the space-time fabric requires awareness of the dynamics of causal relations among events. The application adapts not only to events' movements across time and space but also to changes in their "traveling speed". This dynamic adaptation enhances the system's ability to respond to varying environmental conditions.

6.3.3.3 Controlling Situatedness.

Fine control over the degree of situatedness can be achieved by deciding the granularity of perceived event triggers and how to adapt to changes in event dynamics. This capability is crucial for rapidly reacting to dynamic environmental changes in distributed systems.

6.3.3.4 Co-causal Field Computation.

Associating field computations with programmable scheduling policies allows expressive control but also introduces risks of circular dependencies. While offering increased expressiveness, circular dependencies must be carefully managed to avoid undesirable global behaviours like deadlocks or livelocks. The model implementation can and should be provided with static analysis and simulation tools to assess and mitigate such issues.

6.3.3.5 Pure Reactivity and its Limitations.

The model shifts the system from time-driven to event-driven, resulting in purely reactive behaviour. However, the inclusion of clock triggers and retroactive feedback mechanisms in guard policies provides flexibility and mitigates this limitation, allowing for proactive strategies in coordination policies. For instance, it is possible (and reasonable) to express a policy such as "trigger as soon as event ϵ happens, or timer τ expires, whichever comes first".

6.3.4 Time Fluid Aggregate Computing

Building upon the previously discussed model, we introduce the notion of *time fluid aggregate computing guards*. These guards represent a mechanism capable of dynamically adjusting the scheduling of field computations in response to environmental dynamics. The following sections are dedicated to presenting an extension of the field calculus semantics, specifically designed to integrate this model (Figure 6.3). Additionally, we will detail an implementation of this model within the Protelis programming language, demonstrating the practical applicability and effectiveness of time fluid aggregate computing guards in adaptive distributed systems.

6.3.5 Device-wise Configuration and Semantics

This section formalise a time-fluid extension of the field calculus operational semantics (see Figure 6.3), which is a distributed coordination model for field-based computations. A global field-calculus scheduled program, denoted as $\mathsf{e}[\overline{\pi}]$, integrates a standard expression e with a sequence of scheduling programs $\overline{\pi}$. This structure forms a hierarchical tree where the root expression yields the field computation result, intermediate nodes represent scheduler expressions for their parent nodes, and leaf nodes are expressions scheduled directly by the platform, indicated as $\mathsf{e}[\bullet]$. The evaluation of these expressions involves exchanging an exported mes-

a) Field calculus syntactic elements:

$\mathtt{v}$	field calculus value	$\mathtt{e}$	field calculus expression
θ	field calculus value-tree	δ	device unique identifier
$\mathtt{n}$	sensor name		

b) Field calculus abstracted semantics:

$\delta; \overline{\delta} \mapsto \overline{\theta}; \overline{\mathtt{n}} \mapsto \overline{\mathtt{v}} \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \theta$ — round semantics judgment

c) Local and scheduling configurations:

$\pi ::= \mathtt{e}[\overline{\pi}]$	scheduling program
$\mu ::= \theta[\overline{\mu}]$	exported message
$\Lambda ::= \overline{\delta} \mapsto \overline{\mu}$	local status field
$x ::= \overline{\mathtt{n}} \mapsto \overline{\mathtt{v}}$	local sensor state

d) Scheduling semantics: $\boxed{\delta; \Lambda; x \vdash \pi \Downarrow^s \mu}$

$$\frac{\{\forall i,\ \delta; \overline{\delta} \mapsto \overline{\mu}^i; x \vdash \pi_i \Downarrow^s \mu_i,\ \mu_i = \theta_i^o[\overline{\mu}^0]\} \quad \begin{cases} \theta = \theta_j & \text{if } \neg\mathtt{sch}(\theta_j, \overline{\theta}^o, x(\mathtt{trigger})) \\ \delta_j; \overline{\delta} \mapsto \overline{\theta}; x \vdash \mathtt{e} \Downarrow \theta & \text{otherwise} \end{cases}}{\delta_j; \overline{\delta} \mapsto \overline{\theta}[\overline{\mu}^1, \ldots, \overline{\mu}^n]; x \vdash \mathtt{e}[\overline{\pi}] \Downarrow^s \theta[\overline{\mu}]} \text{ [ROUND]}$$

e) System configurations:

I	$::= \overline{\delta}$	neighbourhood
τ	$::= \overline{\delta} \mapsto \overline{I}$	topology field
Σ	$::= \overline{\delta} \mapsto \overline{x}$	sensor field
Ψ	$::= \overline{\delta} \mapsto \overline{\Lambda}$	status field
Env	$::= \tau : \Sigma$	environment
N	$::= \langle Env; \Psi \rangle$	network configuration

f) Environment well-formedness:

$WFE(\tau : \Sigma)$ holds iff $\mathbf{dom}(\tau) = \mathbf{dom}(\Sigma)$ and $\{\delta\} \subseteq \tau(\delta) \subseteq \mathbf{dom}(\Sigma), \forall \delta \in \mathbf{dom}(\Sigma)$

g) Network semantics: $\boxed{N \xrightarrow{act} N}$

$$\frac{Env = \tau : \Sigma \qquad \delta; \Psi(\delta); \Sigma(\delta), \mathtt{trigger} \mapsto \mathtt{v}_t \vdash \pi_{\mathtt{main}} \Downarrow^s \mu \qquad \Psi_1 = \tau(\delta) \mapsto \{\delta \mapsto \mu\}}{\langle Env; \Psi \rangle \xrightarrow{\delta:\mathtt{v}_t} \langle Env; \Psi[\Psi_1] \rangle} \text{ [N-FIR]}$$

$$\frac{WFN(Env') \qquad Env' = \overline{\delta} \mapsto \overline{I} : \overline{\delta} \mapsto \overline{x} \qquad \Psi_0 = \overline{\delta} \mapsto \overline{I} \mapsto \mu_\perp^{\pi_{\mathtt{main}}}}{\langle Env; \Psi \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi] \rangle} \text{ [N-ENV]}$$

Fig. 6.3: Operational semantics for time-fluid coordination in the field calculus.

sage μ structured as a value tree $\theta[\overline{\mu}]$, reflecting the tree's hierarchical nature and applying field calculus operational semantics node-wise.

The operational semantics for scheduling are encapsulated in the formula "$\delta; \Lambda; x \vdash \pi \Downarrow^s \mu$", interpreting the evaluation of a scheduling program π to an exported message μ on a device δ considering its local status field Λ and sensor state x. Here, Λ maps neighboring devices to their messages, and x maps sensor names to their values.

Evaluation proceeds bottom-up within the tree at each round, contingent upon a scheduling context that includes the outcomes of lower-level scheduler programs,

the previous round's evaluation result of the expression, and the trigger event information. Depending on whether the scheduling context activates the evaluation, the expression is either re-evaluated, updating the exported message with a new value tree, or the previous value tree is reused.

The formal difference between these scenarios is captured by the [ROUND] rule, which processes a program $\mathtt{e}[\overline{\pi}]$ against a neighborhood. It recursively evaluates each scheduler π_i, generating a segment of the exported message μ_i, and then determines whether the main expression `e` should be evaluated based on the `sch`() predicate. This predicate abstractly determines the necessity of re-evaluation by considering the scheduling context, which includes the scheduler evaluations, the latest value tree of the device, and the trigger sensor's value.

6.3.6 System-wise Configuration and Semantics

Building upon the scheduling semantics, the operational semantics for the distributed system's behavior is derived, closely following the methodology presented in [4], adapted for scheduling formalization. The system assumes the existence of a unique aggregate program $\pi_{\mathtt{main}}$ and defines a network configuration N as a combination of an environment *Env* and a global status field Ψ. The environment comprises a topology field τ and a sensor state field Σ, mapping devices to their neighbors and sensor states, respectively, and Ψ represents the communication status across devices.

A network configuration is considered valid if the sensor and topology fields align in domain, with the topology being *reflexive* and *domain-closed*. The operational semantics ensure the preservation of environment well-formedness through transitions, defined by two rules: [N-FIR], modeling a scheduling round triggered by a device, and [N-ENV], representing environmental changes. The [N-FIR] rule updates the global status field to reflect the message exchange between neighbors post-evaluation, while [N-ENV] accounts for changes in topology or sensor states, ensuring the global status field is updated to maintain system integrity and message compliance.

6.3.7 Time-Fuild Scheduling in Protelis

The operational semantics presented in Figure 6.3 need concrete implementations for certain abstracted components. These include:

1. identification of triggers ($\mathtt{v}_t$ from sensor `trigger`) causing a local round via rule [N-FIR];
2. definition of the scheduling tree, allowing the application developer to integrate a main script `e` with its dependent scheduling programs $\overline{\pi}$;

3. implementation and specification of the `sch`() predicate to manage scheduling logic based on the aforementioned scheduling structure.

In our model, scripts form a tree structure of program nodes, each representing a named Protelis module. Dependencies between modules ($\pi \rightarrow \pi'$) denote program node relationships, mapping a program node to π (equivalent to $\mathtt{e}[\overline{\pi}]$), with Protelis modules corresponding to `e` and dependencies to $\overline{\pi}$. Execution scheduling follows the operational semantics, requiring only the specification of the Protelis module at the tree's root to schedule the entire tree. Execution precedence is given to modules without dependencies, followed by those whose dependencies have been executed, culminating with the root module.

Execution consideration involves evaluating the $\mathtt{sch}(\theta, \overline{\theta}^o, x(\mathtt{trigger}))$ predicate for each module, a task delegated to the platform. Implementation can vary, such as a unified function for the entire tree or individual functions per module. We adopt the latter, introducing an API for Protelis modules to define their scheduling predicates via a `scheduler` function, as demonstrated in Listing 1. This function, invoked with parameters injected via annotations, determines module execution.

Annotations `@Input` and `@Changed` are utilized for binding sensor values and detecting value changes, respectively, facilitating local scheduling logic without aggregate constructs. Static analysis of Protelis programs referenced by the main script enables automatic inference of the program tree from source code without additional configuration.

```
1  module a:b
2  def scheduler(
3    @Input("trigger") t,
4    @Input("a:b") old,
5    @Input("c:d") someSchedulerOutput,
6    @Changed("c:d") didChange
7  ) = // local scheduling logic
8
9  // function declarations
10 // main expression
```

Listing 1: Implementation of the $\mathtt{sch}(\theta, \overline{\theta}^o, x(\mathtt{trigger}))$ predicate per module.

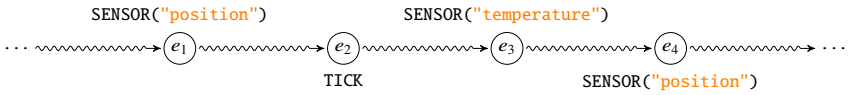
6.3.8 Examples of Time-Fluid Guards

This section presents examples that leverage our scheduling approach for time-fluid aggregate computations. We focus on scenarios where platform-provided triggers (i.e., sensor `trigger` values) play a crucial role. These triggers, highlighted in blue, often lead to scheduler functions acting as logical disjunctions (ORs) of input

conditions or simple predicates, aligning with the intuition that a computation should proceed whenever *any* of its potential triggers occur.

6.3.8.1 Timer-based Scheduling.

This example showcases a policy recreating the classic, timer-based execution model, thus demonstrating how this approach subsumes the original execution model of field computations [4]. Consider a chain of events (and the corresponding triggers) local to one device such as the following.

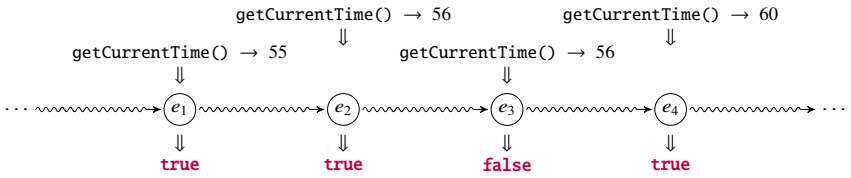


Assuming the platform exposes a sensor for obtaining the current local timestamp in seconds, it is possible to define a timer-like scheduler through a program as in Listing 2. There is no declared scheduler, so it runs in every event. Its output is

```
1  module timefluid:every_second
2
3  // Applies condition to current and to the last value that made
       condition
4  // true, returning the truth value and storing the new state if needed
5  def updated(init, current, condition) =
6    rep(state <- [init, true]) {
7      let old = state.first()
8      if (condition(current, old)) {
9        [current, true]
10     } else {
11       [old, false]
12     }
13   }.get(1)
14
15 updated(-1, self.getCurrentTime()) { now, last -> now-last>=1 }
```

Listing 2: Timer-like scheduler.

a Boolean field mapping a Boolean value to each event. We use notation $\Downarrow$ in the following picture to indicate the input (portion) and output of an event corresponding to a certain program execution.



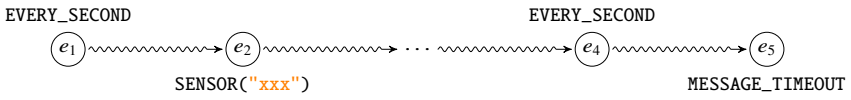
Notice that you could not actually schedule "every second" if the underlying platform events are not triggered with a second or sub-second frequency; moreover, here we only generally consider soft real-time tasks. So, for instance, a downstream program that needs to run at most once per second will not be scheduled in round e_3. Such a program can be defined with a scheduling predicate logic that merely reuses the output of module `timefluid:every_second` as in Listing 3. Function `scheduler` is

```
1 module timefluid:some_program
2
3 // Special function 'scheduler' maps input scheduling fields to Boolean
4 def scheduler(@Input("timefluid:every_second") new_t) = new_t
5
6 // main expression
```

Listing 3: A scheduling predicate that reuses `timefluid:every_second`.

used by the platform to control the scheduling of `some_program`: it is invoked with the set of triggers of the current event, and other input parameters corresponding to the upstream scheduling programs.

An alternative would be to define a base timer trigger `EVERY_SECOND` at the platform level that is issued every second.



Once we have a user-defined program or platform-level timer, derived timers could be defined as in Listing 4.

Reacting to Local Context Changes.

On one end of the policy spectrum, we have a *purely* reactive execution model. Here, local computations are triggered only under specific conditions: a change in sensor values (`SENSOR(".*")`), the receipt of new information (`MESSAGE_RECEIVED`), or the expiration of information from a neighbor (`MESSAGE_TIMEOUT`). An example of such a policy is shown below: This approach ensures computations are only performed

```
module timefluid:every_minute

import platform.Triggers.EVERY_SECOND

// OPTION A) restricting the domain of a high-level scheduler
def scheduler(@Input("timefluid:every_second") s) = s
// OPTION B) restricting the domain of a low-level, platform scheduler
def scheduler(@Input("trigger") t) = t == EVERY_SECOND

// returns true once every 60 timer ticks
rep(old <- -1) { old + 1 }  60 == 0
```

Listing 4: Example timers derived from domain restriction.

```
module timefluid:some-program

import platform.Triggers.*

def scheduler(@Input("trigger") t) =
  t == SENSOR(".*") ||
  t == MESSAGE_RECEIVED("timefluid:some-program") ||
  t == MESSAGE_TIMEOUT("timefluid:some-program")
```

Listing 5: A purely reactive scheduling policy.

when necessary, based on sensor data or message flow, promoting efficiency and responsiveness.

Leveraging Aggregate Computations for Scheduling.

In scenarios like crowd monitoring at large events, two main tasks emerge: estimating crowd density and steering the crowd to prevent congestion. Efficient management requires updating steering directives only upon significant changes in crowd density.

We achieve this by employing the SCR pattern to partition space and calculate average crowd density within 300-meter regions, as demonstrated in the Protelis program Listing 6.

The density computation's result then triggers another computation for steering, as shown next, which is scheduled based on changes in the density estimate (Listing 7). Finally, the steering logic itself is predicated on the output of the scheduler, forming a causal chain of computations that adaptively respond to crowd dynamics (Listing 8), also depicted in Figure 6.4

```
module timefluid:steering:density
import ...

def scheduler(@Input("trigger") t) =
  t == SENSOR("people_count") ||
  t == MESSAGE_RECEIVED("timefluid:steering:density") ||
  t == MESSAGE_TIMEOUT("timefluid:steering:density")

let distToLeader = distanceTo(S(300))
let count = summarize(distToLeader, env.get("people_count"), sum)
let radius = summarize(distToLeader, distToLeader, max)
count / (2 * PI * radius)
```

Listing 6: Estimating local crowd density using the SCR pattern.

```
module timefluid:steering:steering-scheduler

def scheduler(@Changed("timefluid:steering:density") d) = d

changed(exponentialBackOff(env.get("timefluid:steering:density"), 0.1))
{
  cur, old -> abs(cur - old) > 0.5
}
```

Listing 7: Scheduling based on changes in crowd density.

```
module timefluid:steering:steering

def scheduler(@Input("timefluid:steering:steering-scheduler") sched) =
    sched
```

Listing 8: Crowd steering program scheduled by density changes.

6.4 Fluidity in space

In this section, we devise distributed coordination strategies for the estimation of a spatial phenomenon through collaborative adaptive sampling. Our design is based on the idea of dynamically partitioning space into regions that compete and grow/shrink to provide sampling preserving the required accuracy.

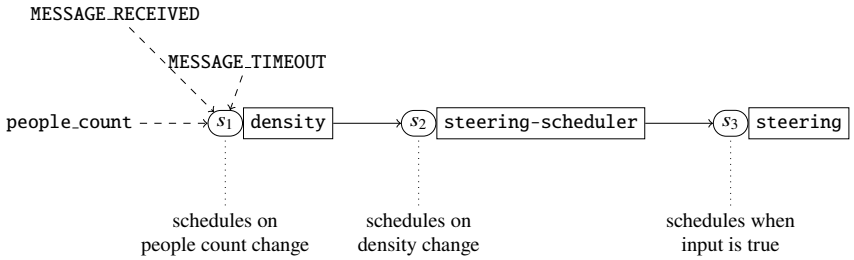


Fig. 6.4: This figure illustrates how Protelis modules, represented as square nodes, and their schedulers, shown as rounded rectangles, interact through triggers and data flows, depicted by solid and dashed arrows, respectively. This design enables dynamic and efficient crowd management by closely linking computation to changes in the environment.

6.4.1 Problem definition

We start by introducing the notion of *regional partition*, which is a finite set of non-overlapping contiguous clusters of devices: a notion that prepares the ground to that of an *aggregate sampling* which we introduce in this chapter.

Definition 6.1 (Regional partition field, contiguous regions) Let $\mathbf{E} = \langle E, \rightsquigarrow, d \rangle$ be a stabilising environment static since event ϵ_0. A *regional partition field* is a stabilising field $f : E \to \mathbb{V}$ on $\mathbf{E}$ such that:

- **(finiteness)** the image $Img(f) = \{f(x) \mid x \in E\}$ is a finite set of values;
- **(eventual contiguity)** there exists an event $\epsilon_0' > \epsilon_0$ such that for any pair of events $\epsilon_1, \epsilon_n \in T^+_{\epsilon_0'}$, $f(\epsilon_1) = f(\epsilon_n)$ implies that there is a sequence of events $\epsilon_1 < ... < \epsilon_n$ connecting ϵ_1 to ϵ_n where $f(\epsilon_i) = f(\epsilon_1) = f(\epsilon_n)\ \forall 1 \leq i \leq n$.

Note that the set of domains of regions induced by f is defined by $regions(f) = \{f^{-1}(v) : v \in Img(f)\}$. Moreover, given two regions $E, E' \in regions(f)$, we say that they are *contiguous* if $\exists \epsilon \in E, \epsilon' \in E' : \epsilon \rightsquigarrow \epsilon' \vee \epsilon' \rightsquigarrow \epsilon$.

An example of a regional partition field is shown in Figure 6.5. Notice that for any pair of events in the same space-time region there exists a path of events entirely contained in that region. Also, notice that, by this definition, different disjoint regions denoted by the same value r are not possible.

Definition 6.2 (Aggregate sampling) An *aggregate sampling* is a stabilising computation $\Phi_S : \mathbb{F}_{E,\mathbb{V}} \to \mathbb{F}_{E,\mathbb{V}}$ that, given an input field to be sampled, yields as output a regional partition field.

Once we have defined an aggregate sampling process in terms of its inputs, outputs, and stabilising dynamics, we need a way to measure the error introduced by the aggregate sampling. To this purpose, we introduce the notion of a *stable snapshot*,

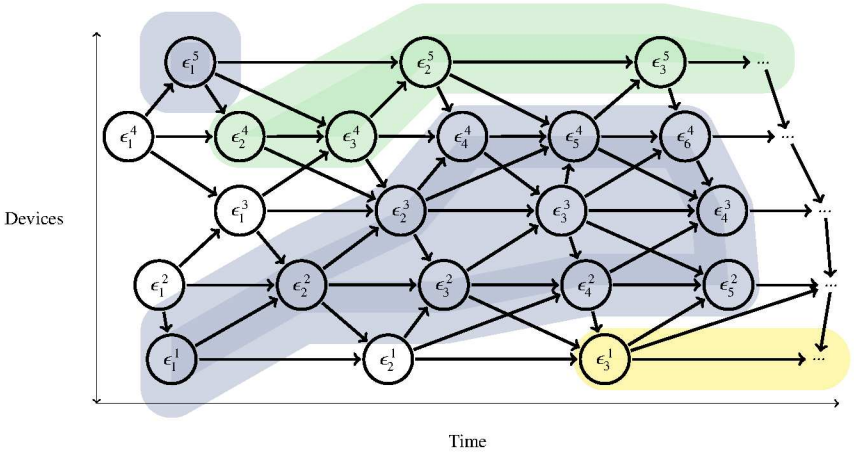


Fig. 6.5: Example of a regional partition field with regions r_{blue}, r_{green}, r_{yellow}, r_{white} (the background is used to denote the output of the field). Notice that contiguity does not hold everywhere and anytime but only since event ϵ_2^3.

namely a field consisting of a sample of one event per device from the stable portion of a stabilising field.

Definition 6.3 (Stable snapshot) Let $\mathbf{E} = \langle E, \rightsquigarrow, d \rangle$ be an event structure, and $f : E \to \mathbb{V}$ be a stabilising field on $\mathbf{E}$ which provides stable output from $\epsilon_0 \in E$. We define a *stable snapshot* of field f as a field obtained by restricting f to a subset of events in the future event cone of ϵ_0 *and* with exactly one event per device, i.e., a field $f_S : E_S \to \mathbb{V}$ such that $E_S \subseteq T^+_{\epsilon_0}$, *and* $\forall \epsilon, \epsilon' \in E_S : d(\epsilon) = d(\epsilon') \implies \epsilon = \epsilon'$, *and* $\forall \epsilon \in T^+_{\epsilon_0}, \exists \epsilon' \in E_S : d(\epsilon') = d(\epsilon)$.

Definition 6.4 (Stable snapshot error-distance) We call *stable snapshot error-distance* any metric $\mu : \mathbb{F}_{E,\mathbb{V}} \times \mathbb{F}_{E,\mathbb{V}} \to \mathbb{R}^+_0$ over stable snapshots that feature same domain (event structure) and codomain (set of values).

We are now able to characterise adequacy properties for a sampling operator, intuitively capturing the fact that sampling correctly trades-off the size of regions with their accuracy. We first start by introducing a notion that handles accuracy, stating that any of the produced regions will not cause the error-distance to be over a certain threshold.

Definition 6.5 (Aggregate sampling error) Let $\Phi_{\mathbf{E}} : \mathbb{F}_{E,\mathbb{V}} \to \mathbb{F}_{E,\mathbb{V}}$ be an aggregate sampling, and consider an input field $f_i : E \to \mathbb{V}$ and corresponding output regional partition $f_o : E \to \mathbb{V}$. We say that f_o *samples* f_i *within error* η *according to error-distance* μ, if the error-distance of stable snapshots of f_i and f_o in any region is not bigger than η, that is: let f_i^s and f_o^s be stable snapshots of f_i and f_o, then for any region $E' \in regions(f_o^s)$, we have $\mu(f_i^s|_{E'}, f_o^s|_{E'}) \leq \eta$.

Note that *accuracy* can be generally achieved simply by partitions defining many small regions—up to the corner case in which all regions include just one device, hence trivially induce zero error-distance. Therefore, we are also interested in *efficiency*, namely the ability of a regional partition to rely on as few regions as possible. Without a centralised approach, however, partitioning is necessarily sub-optimal, since it can rely only on local interaction/competition among regions, hence it should be expected that some regions will stop "expanding" as they reach a smaller threshold. Additionally, there can also be corner cases where regions with very small error-distance are created, e.g., because what remains to be covered in an iterative selection of regions is simply a very small part of the network, or one with rather uniform values introducing little sampling errors. What we may require from an adequate sampling operator, however, is that such regions are somewhat not the norm. This is formally captured by the following definition, essentially introducing a "lower bound" for the error-distance of regions.

Definition 6.6 (Local optimality of a regional partition) Let $\Phi_{\mathbf{E}} : \mathbb{F}_{E,\mathbb{V}} \to \mathbb{F}_{E,\mathbb{V}}$ be an aggregate sampling, consider an input field f_i and corresponding output regional partition f_o such that f_o samples f_i within error η according to distance μ, and denote with f_i^s and f_o^s the stable snapshots of f_i and f_o, respectively (cf. Definition 6.5). We say that f_o is *locally optimal* under error η and with efficiency k ($k > 0$) if all pairs of contiguous regions $E', E'' \in regions(f_o)$ are such that $\mu(f_i^s|_{E' \cup E''}, f_o^s|_{E' \cup E''}) \geq k \cdot \eta$.

For example, we will show that the algorithm we propose guarantees $k = 0.5$ (see Section 6.4.5). Note that we call this notion "local optimality" to stress the fact that an identified partition is not necessarily the best one that could be found, but it is one that cannot be significantly improved with a small change, such as combining two regions—a small improvement is possible, depending on the efficiency factor k. This notion well fits our goal of dealing with dynamic phenomena and large-scale environments, where one is more geared towards finding good heuristics for self-organising behaviour.

So, we are now ready to define the goal operator.

Definition 6.7 (Effective sampling operator) An *effective sampling operator* with efficiency k is a self-stabilising operator P_η, parametric in the error bound η, such that in any stabilising environment $\mathbf{E}$ and stabilising input f_i, a locally optimal regional partition with efficiency k and within error η is produced.

6.4.2 Aggregate Computing-based Solution

In this section, we define a *space-based adaptive sampling* algorithm, called AggregateSampler, discuss its implementation in *aggregate computing* [5, 34], and prove the algorithm is a self-stabilising, effective sampler with efficiency at least $k = 0.5$.

6.4.3 AggregateSampler Algorithm for Adaptive Spatial Sampling

The problem of creating partitions in a self-organising way is very much related to a problem of multi-leader election [26, 24]. Building on this idea, our approach starts by solving a *sparse leaders election problem* [20], for which self-stabilising solutions exist [22, 8, 24].

Leaders are used as *samplers* of the input field. During the election of leaders/samplers, we associate them with larger and larger regions of "follower devices" that will provide the sampled value as output. During execution of the algorithm, such regions will expand until the desired error-distance can be kept under the threshold η. This process is managed so that there will not be any overlap with other regions, and so that no devices of the network remain outside of some region (i.e., each device will follow exactly one leader).

To ensure that regions are connected, and will not overcome the threshold independently of the chosen leader, we adopt as error-distance one based on "distance among devices", as follows. The algorithm can be configured to adopt any strategy that is able to turn input and output fields into a metric m for devices: such a metric is as usual a function mapping a pair of neighbour devices to a non-negative real number, called the "local sampling distance" of the two devices—intuitively, the higher the physical distance of devices and the higher the difference of input values and output values of the two devices, the higher is m for that pair. Given this metric, any pair of devices of the network can be associated with a *path sampling error*, which is the size of the shortest path (according to the metric) connecting the two devices. The proposed algorithm will then produce regions adopting as error-distance μ the maximum path sampling error of any pair of devices in the region, and it will turn out that any pair of contiguous regions combined will necessarily give error-distance greater than $0.5 * \eta$ (efficiency $k = 0.5$).

The algorithm is defined as follows (see Figure 6.6):

1. each device announces its candidature for leadership;
2. each device propagates to its neighbours the candidature of the device it currently recognises as leader, its sampled value, and the path sampling error from it, fostering the expansion of its corresponding region;
3. devices discard candidatures whose path sampling error from the leader exceeds half the expected threshold ($\eta/2$);
4. in case multiple valid candidatures (i.e., those that are not discarded) reach a device, one is selected based on a *competition policy*.

The specific strategies for computing the local sampling distance and the leader competition policy are application-dependent.

Competition and leader strength.

Although competition among leaders could be realised in several ways, many techniques may lead to non-self-stabilising behaviour: for instance, if the winning leader

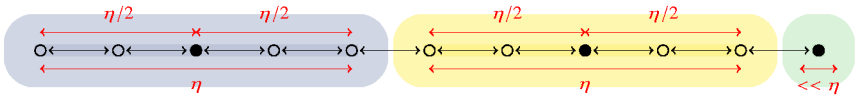


Fig. 6.6: Example of a regional partitioning (with three contiguous regions) created by the algorithm on a simplified system where devices are arranged on a line. Notation: black dots denote the leaders/samplers; the coloured areas denote regions; and the red extension lines are used to denote the error-distances. Note that no device in a region can have path sampling error greater than $\eta/2$ with respect to the leader, and that very small regions can still exist in corner cases (e.g., the green region on the right).

is selected randomly in the set of those whose error is under threshold, regions may keep changing even in a static environment. In this work, we propose a simple strategy: every leader associates its candidature with the local value of a field that we call *leader strength*; in case of competing candidatures, the highest such value is selected as winner, breaking the symmetry. The leader strength can be of any orderable type, and its choice impacts the overall selection of the regions by imposing a selection priority over leaders (hence on region-generation points). If two candidate leaders have the same strength, then we prefer the closest one. If we are in the (unlikely) situation of perfect symmetry, with two equally-strong candidate leaders at the same distance, then their device identifier is used to break symmetry.

Region expansion and path sampling error.

Inspired by previous work on distributed systems whose computation is independent of device distribution [6], the proposed approach essentially accumulates the path sampling error along the path from the leader device towards other devices along a gradient, a distributed data structure that can be generated through self-stabilising computations [33]. We thus have two major drivers:

1. the leader strength affects the creation of regions by influencing the positions of their source points;
2. the path sampling error influences the expansion in space of the region across all directions, mandating its size and (along with the interaction with other regions) its shape.

For instance, a metric could be the absolute value of the difference in the perceived signal (e.g., a value sampled from a sensor) between two devices: devices perceiving very different values would tend not to cluster together (even if spatially close), as they would perceive each other as farther away (leading to irregular shapes).

6.4.4 Aggregate Computing-based Implementation

```
1  // Definition of the record (product type) Sample + accessor functions
2  def Sample(symmetryBreaker, distance, leaderId) = [symmetryBreaker,
       distance, leaderId]
3  def breakSymmetry(sample) = sample.get(0)
4  def sampleDistance(sample) = sample.get(1)
5  def areaCenter(sample) = sample.get(2)
6  def discard() = Sample(POSITIVE_INFINITY, POSITIVE_INFINITY,
       POSITIVE_INFINITY)
7  // Logic to control the propagation of candidacies
8  def expansionLogic(sample, localId, radius) =
9    mux (areaCenter(sample) == localId || sampleDistance(sample) >= radius
       ) {
10     discard()
11   } else {
12     sample
13   }
14
15 def AggregateSampler(mid, radius, symmetryBreaker, metric) {
16   let local = Sample(-symmetryBreaker, 0, mid)
17   areaCenter(
18     share (received <- local) {
19       let candidacies = received.set(1, received.get(1) + metric())
20       let filtered = expansionLogic(candidacies, mid, radius)
21       min(local, foldMin(discard(), filtered))
22     }
23   )
24 }
25
```

Fig. 6.7: Source code of the algorithm.

An implementation of the algorithm expressed in the Protelis aggregate programming language [27] is shown in Figure 6.7. In aggregate computing, a so-called *aggregate program* such as the one shown in Figure 6.7, is repeatedly run by all the devices: it expresses a logic for mapping the *local context* (given by sensor readings and messages from neighbours) to an *output value* and an *output message* to be sent to all the neighbours. In other words, in aggregate computing each event denotes a full execution of the aggregate program against the event's inputs, determining the message payload passed to receiving events.

The core of the program is function AggregateSampler, which consists of the following main elements:

- *sampler candidacies are modelled as ordered triplets*, using 3-element tuples, with corresponding accessor functions (Figure 6.7, Lines 1–6), of the elements:

1. `simmetryBreaker`: a value used to break symmetry, capturing the "strength" of a candidacy;
2. `distance`: a value capturing the distance to the sampler node of a candidacy;
3. `leaderId`: holding the device identifier of the candidate sampler;

- function `expansionLogic` (Figure 6.7, 8–13) is defined to determine when a candidacy has to be discarded, i.e., when it comes from the device itself or when the distance of the candidate sampler is greater or equal than a `radius` parameter;
- function AGGREGATESAMPLER (Figure 6.7, 15–24) is the entry point of the algorithm, parametrised in terms of the executing device identifier (`mid`), a maximum spatial range of candidacies (`radius`), a value to break symmetry (`symmetryBreaker`), and a `metric` function providing distances to neighbours;
- **`share`**`(x <- init){ e }` is a bidirectional communication construct [3], that works as follows: the declared variable `x`, which is set to `init` at the first round, collects the evaluations of the overall **`share`** expression in neighbour devices (including the device itself), and the new value for the current device (which is the data item that will be sent to neighbours) is obtained by evaluating expression `e`;
- *the distance field of the neighbour candidacies is updated*, (Figure 6.7, Line 19), by adding to each candidacy provided by a neighbour the local distance w.r.t. that neighbour[1] (as provided by `metric`);
- *neighbours' candidacies that are too far or support the current device are deprioritised* (Figure 6.7, Lines 20), by function `expansionLogic()`; and, finally
- *selecting the winner over the processed candidacies through minimisation*, by `min` and `foldMin` (Figure 6.7, Line 21), which minimise over the filtered candidacy triplets, with default candidacy as the one with the lowest priority (provided by `discard()`), and against the `local` candidacy.

The algorithm described above is an effective sampling operator (Definition 6.7) as long as (i) *half the path sampling error η is used as parameter* `radius`, so that function `expansionLogic` does not expand regions beyond the given error η, and (ii) *an additive* `metric` *is used*, so that it is impossible to decrease the error by expanding any given region (at best, it will stay the same).

6.4.5 Formal Analysis

In this section, we prove that the proposed solution is self-stabilising and that it represents an effective sampling operator leading to a bounded-error locally optimal regional partition (cf. Definition 6.2, Definition 6.6, Definition 6.7).

Since our aggregate sampling must be a stabilising computation (see Definition 6.2), we start by proving that our algorithm is self-stabilising. We do so by exploiting the framework in [33, 3], which defines a set of self-stabilising *fragments* which can be composed together to yield self-stabilising operators. In particular,

[1] Note that this operation, together with the **`share`** application, essentially provides the same structure as the basic gradient algorithm.

in [33] it is proved that any closed expression in the self-stabilising fragment is self-stabilising, by structural induction on the syntax of expressions and programs ([33], Appendix E, Lemma 2): values and variables are already self-stabilised, a function application self-stabilises (by the inductive hypothesis) if its arguments are self-stabilising, and similar considerations can be done for the other program fragments.

```
1  def updateDistance(x, metric) {
2    x.set(1, x.get(1) + metric())
3    x
4  }
5  def fR(x, prev) = x // raising function
6  def fMP(x, localId, radius, metric) = // monotonic progr.
7    expansionLogic(updateDistance(x, metric), localId, radius)
8  def minHoodLoc(e, loc) = // minimum of loc and e's values
9    min(loc, foldMin(discard(), e))
10 def AggregateSampler(mid, radius, symmetryBreaker, metric) {
11   let local = Sample(-symmetryBreaker, 0, mid)
12   share (x <- local) {
13     fR(minHoodLoc(fMP(x, mid, radius, metric), local), x)
14   }.get(2)
15 }
16
```

Fig. 6.8: Protelis code from Figure 6.7 rewritten to conform to the *minimising share* self-stabilising pattern as per the Proof.

Theorem (AggregateSampler is self-stabilising)

Proof In [3], it is proved that an expression of the following form (called a *minimising share* pattern) is self-stabilising:

```
1  share(x <- e) { fR(minHoodLoc(fMP(x), e), x) }
```

where (see Section 5.2 in [33] and, especially, Section 4.7 and Figure 7 in [3]):

- `fR(x, prev)` is a "raising function", with respect to partial orders, of `x` and `prev` (the value of `x` at the previous round);
- `fMP` is a monotonic progressive function of `x`, which can take further arguments as far as they are self-stabilising expressions that do not contain the `share`-bounded variable `x`; and
- `minHoodLoc(e, loc)` selects the minimum among the neighbours' values of expression `e` and the current device's local value `loc`.

Now, the block of Protelis code (Figure 6.7) in Lines 15–24 can be rewritten as shown in Figure 6.8 that conforms to the minimising `share` pattern, where:

- the raising function `fR` is an identity on the first parameter, which is a trivially valid raising function (see Example 5.5 in [33]);
- function `expansionLogic` is a valid monotonic progressive function `fMP` of `x`, since it transforms neighbours' candidacies supporting the current device and those at a distance farther than `radius` to the highest value for the data type (cf. `discard()`), leaving the others unaltered; none of the provided additional arguments (`id`, `radius`, and `metric`) contains the **`share`**-bounded variable `x`.

More gradually, the transformation can be obtained by:

1. renaming `received` to `x`, and defining functions `fR`, `fMP`, and `minHoodLoc`;
2. realising that `fMP` is a valid replacement for the combination of distance field update and call to `expansionLogic` and replacing accordingly;
3. replacing `areaCenter` with its definition;
4. replacing the **`share`** body with `minHoodLoc`, as they perform the same operation;
5. adding a call to `fR`, which is an identity function, does not alter the behaviour of the code and leads directly to the code in Figure 6.8.

The other elements in the program are only operations on local data which are also self-stabilising expressions. Since the AggregateSampler function consists exclusively of self-stabilising expressions, it is in turn self-stabilising [33, 3]. □

Theorem (AggregateSampler is an effective sampling operator)

Proof To prove that our algorithm represents an effective sampling operator P_η = AggregateSampler, we have to prove that it yields, on any stabilising input field f_i, an output stabilising field f_o of locally optimal regional partitions. As per AggregateSampler is an effective sampling operator 1, P_η is self-stabilising, so on a stable input it will yield a stable output: let f_o^s be its snapshot, and f_i^s the corresponding input.

On the one hand, accuracy is guaranteed since AggregateSampler ensures that no device has path sampling error greater than $\eta/2$ from the leader: for the triangular inequality property of metric spaces ($m(a,b) \leq m(a,c) + m(c,b)$) this ensures that stable snapshot distance μ does not overcome η.

On the other hand, for local optimality under error η and distance μ, there must not exist two contiguous regions $E', E'' \in regions(f_o^s)$, with samplers δ' and δ'', where $\mu(f_i^s|_{E' \cup E''}, f_o^s|_{E' \cup E''}) \leq \eta/2$. Suppose two such regions exists, and let δ' be stronger than δ'' (i.e., it has higher symmetry breaker). Then, the path sampling error between δ' and δ'' is necessarily higher than $\eta/2$, because of the steps 3 and 4 of the algorithm: in fact, if it were smaller than $\eta/2$ then δ'' would have followed δ', and would not have been a leader (cf. Figure 6.9). □

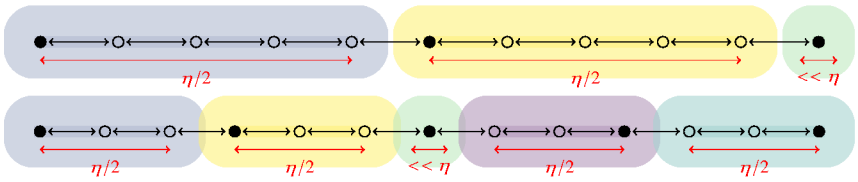


Fig. 6.9: Examples of regional partitionings with efficiency 0.5. Notation: black dots denote the leaders; the coloured areas denote regions; and the red extension lines are used to denote the error-distances. Note how the union of the green singleton region (associated to the weakest leader) with its neighbouring regions would make the error-distance of the latter exceed $\eta/2$—if that would not be the case, then the former region would not have existed in the first place (cf. Proof 2).

6.5 Conclusion

In this chapter we presented two works presenting a first operational definition of, respectively, time-fluid and space-fluid distributed computations, modelled and implemented in the conceptual and technological framework offered by the field calculus. These works should give both an intuition and a first formal account of how the kind of fluid computations envisioned by FluidWare can be actually enabled, supported, and leveraged by currently available models and platforms.

About time-fluid computations, a natural extension of such model is to consider learning mechanisms to adapt the scheduling of field computations [1] as discussed in the chapter "Learning Opportunities in Collective Adaptive Systems" of this book.

Conversely, a natural next step along these lines would be to integrate the two proposed approaches into a unique time-space fluid computing platform, where computations can seamlessly adapt to the temporal evolution of the phenomena they deal with, as well as to their spatial dynamics. The fact that both approaches have been both theoretically and technically grounded in the field calculus formal model and programming toolchain can greatly help in this endeavour. However, care must be used regarding two concerns: first of all, there is an overhead in supporting the described adaptation mechanisms directly at the platform (i.e. middleware level), that must be measured and taken into account when considering applications with some sort of timing and performance constraints; second, the quest toward autonomous adaptation that we are pursuing must not let control slip away from our hands, as system engineers must have a way to intervene on these platform-level adaptation mechanisms when due, and "override" their emergent behaviour with the desired one.

We are well aware of these concerns, and hope that the proposed first step in the direction of time and space-fluid computations that we have taken can pave the way for others to explore similar research avenues.

References

1. Ageev, A., Macii, D., Flammini, A.: Towards an adaptive synchronization policy for wireless sensor networks. In: 2008 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication. IEEE (Sep 2008). https://doi.org/10.1109/ispcs.2008.4659224, https://doi.org/10.1109/ispcs.2008.4659224
2. Aguzzi, G., Casadei, R., Viroli, M.: Addressing collective computations efficiency: Towards a platform-level reinforcement learning approach. In: Casadei, R., Nitto, E.D., Gerostathopoulos, I., Pianini, D., Dusparic, I., Wood, T., Nelson, P.R., Pournaras, E., Bencomo, N., Götz, S., Krupitzer, C., Raibulet, C. (eds.) IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2022, Virtual, CA, USA, September 19-23, 2022. pp. 11–20. IEEE (2022). https://doi.org/10.1109/ACSOS55765.2022.00019, https://doi.org/10.1109/ACSOS55765.2022.00019
3. Audrito, G., Beal, J., Damiani, F., Pianini, D., Viroli, M.: Field-based coordination with the share operator. Log. Methods Comput. Sci. **16**(4) (2020), https://lmcs.episciences.org/6816
4. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Transactions on Computational Logic **20**(1), 1–55 (jan 2019). https://doi.org/10.1145/3285956, https://doi.org/10.1145/3285956
5. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. IEEE Computer **48**(9), 22–30 (2015). https://doi.org/10.1109/MC.2015.261, http://dx.doi.org/10.1109/MC.2015.261
6. Beal, J., Viroli, M., Pianini, D., Damiani, F.: Self-adaptation to device distribution in the internet of things. ACM Transactions on Autonomous and Adaptive Systems **12**(3), 12:1–12:29 (2017). https://doi.org/10.1145/3105758, https://doi.org/10.1145/3105758
7. Bicocchi, N., Mamei, M., Zambonelli, F.: Self-organizing virtual macro sensors. ACM Transactions on Autonomous and Adaptive Systems **7**(1), 2:1–2:28 (2012). https://doi.org/10.1145/2168260.2168262, https://doi.org/10.1145/2168260.2168262
8. Burman, J., Chen, H., Chen, H., Doty, D., Nowak, T., Severson, E.E., Xu, C.: Time-optimal self-stabilizing leader election in population protocols. In: PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021. pp. 33–44. ACM (2021). https://doi.org/10.1145/3465084.3467898, https://doi.org/10.1145/3465084.3467898
9. Casadei, R., Mariani, S., Pianini, D., Viroli, M., Zambonelli, F.: Space-fluid adaptive sampling by self-organisation. Logical Methods in Computer Science **Volume 19, Issue 4** (Dec 2023). https://doi.org/10.46298/lmcs-19(4:29)2023
10. Fischer, M.J.: The consensus problem in unreliable distributed systems (a brief survey). In: International conference on fundamentals of computation theory. pp. 127–140. Springer (1983)
11. Gershenson, C.: Design and control of self-organizing systems. CopIt Arxives (2007)
12. Hélary, J.M., Hurfin, M., Mostefaoui, A., Raynal, M., Tronel, F.: Computing global functions in asynchronous distributed systems with perfect failure detectors. IEEE Transactions on Parallel and Distributed Systems **11**(9), 897–909 (2000)
13. Ho, Y., Huang, Y., Chu, H., Chen, L.: Adaptive sensing scheme using naive bayes classification for environment monitoring with drone. International Journal of Distributed Sensor Networks **14**(1) (2018). https://doi.org/10.1177/1550147718756036, https://doi.org/10.1177/1550147718756036
14. Jelasity, M., Montresor, A., Babaoglu, Ö.: Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems **23**(3), 219–252 (2005). https://doi.org/10.1145/1082469.1082470, https://doi.org/10.1145/1082469.1082470
15. Kho, J., Rogers, A., Jennings, N.R.: Decentralized control of adaptive sampling in wireless sensor networks. ACM Transactions on Sensor Networks **5**(3), 19:1–19:35

(2009). https://doi.org/10.1145/1525856.1525857, https://doi.org/10.1145/1525856.1525857
16. Lee, E.K., Viswanathan, H., Pompili, D.: SILENCE: distributed adaptive sampling for sensor-based autonomic systems. In: Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011. pp. 61–70. ACM (2011). https://doi.org/10.1145/1998582.1998594, https://doi.org/10.1145/1998582.1998594
17. Lee, J., Yoon, G., Choi, H.: Monitoring of iot data for reducing network traffic. In: Tenth International Conference on Ubiquitous and Future Networks, ICUFN 2018, Prague, Czech Republic, July 3-6, 2018. pp. 395–397 (2018). https://doi.org/10.1109/ICUFN.2018.8436601, https://doi.org/10.1109/ICUFN.2018.8436601
18. Lin, Y., Megerian, S.: Sensing driven clustering for monitoring and control applications. In: 4th IEEE Consumer Communications and Networking Conference, CCNC 2007, Las Vegas, NV, USA, January 11-13, 2007. pp. 202–206. IEEE (2007). https://doi.org/10.1109/CCNC.2007.47, https://doi.org/10.1109/CCNC.2007.47
19. Liu, Z., Xing, W., Zeng, B., Wang, Y., Lu, D.: Distributed spatial correlation-based clustering for approximate data collection in WSNs. In: 27th IEEE International Conference on Advanced Information Networking and Applications, AINA 2013, Barcelona, Spain, March 25-28, 2013. pp. 56–63. IEEE Computer Society (2013). https://doi.org/10.1109/AINA.2013.26, https://doi.org/10.1109/AINA.2013.26
20. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
21. Mattern, F., et al.: Virtual time and global states of distributed systems. Univ., Department of Computer Science (1988)
22. Mo, Y., Audrito, G., Dasgupta, S., Beal, J.: A resilient leader election algorithm using aggregate computing blocks. IFAC-PapersOnLine **53**(2), 3336–3341 (2020). https://doi.org/10.1016/j.ifacol.2020.12.1497, https://doi.org/10.1016/j.ifacol.2020.12.1497
23. Parunak, H.V.D., Brueckner, S.A.: Software engineering for self-organizing systems. Knowl. Eng. Rev. **30**(4), 419–434 (2015). https://doi.org/10.1017/S0269888915000089
24. Pianini, D., Casadei, R., Viroli, M.: Self-stabilising priority-based multi-leader election and network partitioning. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2022, Virtual, CA, USA, September 19-23, 2022. pp. 81–90. IEEE (2022). https://doi.org/10.1109/ACSOS55765.2022.00026, https://doi.org/10.1109/ACSOS55765.2022.00026
25. Pianini, D., Casadei, R., Viroli, M., Mariani, S., Zambonelli, F.: Time-fluid field-based coordination through programmable distributed schedulers. Log. Methods Comput. Sci. **17**(4) (2021). https://doi.org/10.46298/lmcs-17(4:13)2021, https://doi.org/10.46298/lmcs-17(4:13)2021
26. Pianini, D., Casadei, R., Viroli, M., Natali, A.: Partitioned integration and coordination via the self-organising coordination regions pattern. Future Generation Computing Systems **114**, 44–68 (2021). https://doi.org/10.1016/j.future.2020.07.032, https://doi.org/10.1016/j.future.2020.07.032
27. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015. pp. 1846–1853 (2015). https://doi.org/10.1145/2695664.2695913, http://doi.acm.org/10.1145/2695664.2695913
28. Singh, V.K., Singh, G., Pande, S.: Emergence, self-organization and collective intelligence - modeling the dynamics of complex collectives in social and organizational settings. In: UKSim. pp. 182–189. IEEE (2013). https://doi.org/10.1109/UKSim.2013.77
29. Sundararaman, B., Buy, U., Kshemkalyani, A.D.: Clock synchronization for wireless sensor networks: a survey. Ad Hoc Networks **3**(3), 281–323 (2005). https://doi.org/10.1016/j.adhoc.2005.01.002, https://doi.org/10.1016/j.adhoc.2005.01.002

30. Szczytowski, P., Khelil, A., Suri, N.: Asample: Adaptive spatial sampling in wireless sensor networks. In: IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, SUTC 2010 and IEEE International Workshop on Ubiquitous and Mobile Computing, UMC 2010, 7-9 June 2010, Newport Beach, California, USA. pp. 35–42. IEEE Computer Society (2010). https://doi.org/10.1109/SUTC.2010.37, https://doi.org/10.1109/SUTC.2010.37
31. Traub, J., Breß, S., Rabl, T., Katsifodimos, A., Markl, V.: Optimized on-demand data streaming from sensor nodes. In: Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017. pp. 586–597 (2017). https://doi.org/10.1145/3127479.3131621, https://doi.org/10.1145/3127479.3131621
32. Trihinas, D., Pallis, G., Dikaiakos, M.: Low-cost adaptive monitoring techniques for the internet of things. IEEE Transactions on Services Computing pp. 1–1 (2018). https://doi.org/10.1109/tsc.2018.2808956, https://doi.org/10.1109/tsc.2018.2808956
33. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Transactions on Modeling and Computer Simulation **28**(2), 1–28 (mar 2018). https://doi.org/10.1145/3177774, https://doi.org/10.1145/3177774
34. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. Journal of Logical and Algebraic Methods in Programming **109**, 100486 (Dec 2019). https://doi.org/10.1016/j.jlamp.2019.100486, https://doi.org/10.1016/j.jlamp.2019.100486
35. Virrankoski, R., Savvides, A.: TASC: topology adaptive spatial clustering for sensor networks. In: IEEE 2nd International Conference on Mobile Adhoc and Sensor Systems, MASS 2005, November 7-10, 2005, The City Center Hotel, Washington, USA. p. 10. IEEE Computer Society (2005). https://doi.org/10.1109/MAHSS.2005.1542850, https://doi.org/10.1109/MAHSS.2005.1542850