Time-Fluid Field-Based Coordination

Danilo Pianini[1](✉), Stefano Mariani[2], Mirko Viroli[1], and Franco Zambonelli[2]

[1] Alma Mater Studiorum—Università Bologna, Cesena, Italy
{danilo.pianini,mirko.viroli}@unibo.it
[2] Università di Modena e Reggio Emilia, Reggio Emilia, Italy
{stefano.mariani,franco.zambonelli}@unimore.it

Abstract. Emerging application scenarios, such as cyber-physical systems (CPSs), the Internet of Things (IoT), and edge computing, call for coordination approaches addressing openness, self-adaptation, heterogeneity, and deployment agnosticism. Field-based coordination is one such approach, promoting the idea of programming system coordination declaratively from a global perspective, in terms of functional manipulation and evolution in "space and time" of distributed data structures, called *fields*. More specifically, regarding time, in field-based coordination it is assumed that local activities in each device, called *computational rounds*, are regulated by a fixed clock, typically, a fair and unsynchronized distributed scheduler. In this work, we challenge this assumption, and propose an alternative approach where the round execution scheduling is naturally programmed along with the usual coordination specification, namely, in terms of a *field of causal relations* dictating what is the notion of causality (why and when a round has to be locally scheduled) and how it should change across time and space. This abstraction over the traditional view on global time allows us to express what we call "time-fluid" coordination, where causality can be finely tuned to select the event triggers to react to, up to to achieve improved balance between performance (system reactivity) and cost (usage of computational resources). We propose an implementation in the aggregate computing framework, and evaluate via simulation on a case study.

Keywords: Aggregate computing · Fluidware · IoT · Internet of Things · Edge computing · Causality · Time · Reactive

1 Introduction

Emerging application scenarios, such as the Internet of Things (IoT), cyber-physical systems (CPSs), and edge computing, call for software design approaches addressing openness, heterogeneity, self-adaptation, and deployment agnosticism [19]. To effectively address this issue, researchers strive to define increasingly higher-level concepts, reducing the "abstraction gap" with the problems at hand, e.g., by designing new languages and paradigms. In the context of coordination models and languages, field-based coordination is one such approach [3,5,21,23,37,40]. In spite of its many variants and implementations, field-based coordination roots in the idea of programming system coordination declaratively and from a global perspective, in terms of distributed data structures called (computational) *fields*, which span the entire deployment in space (each device holds a value) and time (each device continuously produces such values).

Regarding time, which is the focus of this paper, field-based coordination typically abstracts from it in two ways: *(i)* when a specific notion of local time is needed, this is accessed through a sensor as for any other environmental variable; and *(ii)* a specification is actually interpreted as a small computation chunk to be carried on in *computation rounds*. In each round a device: *(i)* sleeps for some time; *(ii)* gathers information about state of computation in previous round, messages received by neighbors while sleeping, and contextual information (i.e. sensor readings); and *(iii)* uses such data to evaluate the coordination specification, storing the state information in memory, producing a value output, and sending relevant information to neighbors. So far, field-based coordination approaches considered computational rounds as being regulated by an externally imposed, fixed distributed clock: typically, a fair and unsynchronized distributed scheduler. This assumption however, has a number of consequences and limitations, both philosophical and pragmatic, which this paper aims to address.

Under a philosophical point of view, it follows a pre-relativity view of time that meets general human perception, i.e., where time is absolute and independent of the actual dynamics of events. This hardly fits with more modern views connecting time with a deeper concept of *causality* [22], as being only meaningful relative to the existence of events as in *relational* interpretations of space-time [30], or even being a mere derived concept introduced by our cognition [29]—as in Loop Quantum Gravity [31]. Under a practical point of view, consequences on field-based coordination are mixed. The key practical advantage is simplicity. First, the designer must abstract from time, leaving the scheduling issue to the underlying platform. Second, the platform itself can simply impose local schedulers statically, using fixed frequencies that at most depend on the device computational power or energetic requirements. Third, the execution in proactive rounds allows a device to discard messages received few rounds before the current one, thus considering non-proactive senders to have abandoned the neighborhood, and simply modeling the state of communication by maintaining the most recent message received from each neighbor.

However, there is a price to pay for such a simple approach. The first is that "stability" of the computation, namely, situations in which the field will not change after a round execution, is ignored. As a consequence, sometimes "unnecessary" computations are performed, consuming resources (both energy and bandwidth capacity), and thus reducing the *efficiency* of the system. Symmetrically, there is a potential *responsiveness* issue: some computations may require to be executed more quickly under some circumstances. For instance, consider a crowd monitoring and steering system for urban mass events as the one exemplified in [7]: in case the measured density of people gets dangerous, a more frequent evaluation of the steering advice field is likely to provide more precise and timely advices. Similar considerations apply for example to the area of landslide monitoring [28], where long intervals of immobility are interspersed by sudden slope movements: sensors sampling rate can and should be low most of the time, but it needs to get promptly increased on slope changes. This generally suggests a key unexpressed potential for field-based computation: the general ability to provide improved balance between performance (system reactivity) and cost (usage of computational resources). For instance, the crowd monitoring and landslide monitoring systems should ideally slow down (possibly, halt entirely) the evaluation in case of sparse crowd density or of absence of surface movements, respectively. And they should start being more and more responsive with growing crowd densities or in case of landslide activation.

The general idea that round execution distribution can actually dynamically depend on the outcome of computation itself, can be captured in field-based coordination by modeling time by a *causality field*, namely, a field programmable along with (and hence intertwined with) the usual coordination specification, dictating (at each point in space-time) what are the triggers whose occurrence should correspond to the execution of computation rounds. Programming causality along with coordination leads us to a notion of *time-fluid coordination*, where it is possible to flexibly control the balance between performance and cost of system execution. Accordingly, in this work we discuss a causality-driven interpretation of field-based coordination, proposing an integration with the field calculus [3] with the goal of evaluating a model for time-fluid, field-based coordination. In practice, we assume computations are not driven by time-based rounds, but by *perceivable local event triggers* provided by the platform (hardware/-software stack) executing the aggregate program, such as messages received, change in sensor values, and time passing by. The aggregate program specification itself, then, may affect scheduling of subsequent computations through policies (expressed in the same language) based on such triggers.

The contribution of this work can be summarized under three points of view. First, the proposed model enriches the *coordination abstraction* of field-based coordination with the possibility to explicitly and possibly reactively program the scheduling of the coordination actions; second, it enables a *functional description of causality and observability*, since manipulation of the interaction frequency among single components of the coordinated system reflects in changes in how causal events are perceived, and actions are taken in response to event triggers; third, the most immediate *practical implication* of a time-fluid coordination when compared to a traditional time-driven approach is improved efficiency, intended as improved responsiveness with the same resource cost.

The remainder of this work is as follows: Sect. 2 frames this work with respect to the existing literature on topic; Sect. 3 introduces the proposed time-fluid model and discusses its implications; Sect. 4 presents a prototype implementation in the framework of aggregate computing, showing examples and evaluating the potential practical implications via simulation finally, Sect. 5 discusses future directions and concludes the work.

2 Background and Related Work

Time and synchronization have always been key issues in the area of distributed and pervasive computing systems. In general, in distributed systems, the absence of a globally shared physical clock among nodes makes it impossible to rely on absolute notions of time. Logical clocks are hence used instead [17], realizing a sort of causally-driven notion of time, in which the "passing time" of a distributed computation (that is, the ticks of logical clocks) directly expresses causal relations between distributed events. As a consequence, any observation of a distributed computation that respects such causal relations, independently of the relative speeds of processes, is a consistent one [4]. Our proposal absorbs these foundational lessons, and brings them forward to consider the strict relations between the spatial dimension and the temporal dimension that situated aggregate computations have to account for.

In the area of sensor networks, acquiring a (as accurate as possible) globally shared notion of time is of fundamental importance [33], to properly capture snapshots of the distributed phenomena under observation. However, global synchronization 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 synchronized way. In most of existing proposals, though, this is done in awakening and communicating rounds of fixed duration, which makes it impossible to adapt to the actual dynamics of the phenomena under observation. Several proposals exist for adaptive synchronization in wireless sensor networks [1,13,16], dynamically changing the sampling frequency (and hence frequency of communication rounds) so as to adapt to the dynamics of the observed phenomena. For instance, in the case of crowd monitoring systems, it is likely that people (e.g, during an event) stay nearly immobile for most of the time, then suddenly start moving (e.g., at the end of the event). Similarly, in the area of landslide monitoring, the situation of a slope is stable for most of the time, with periodic occurrences of (sometimes very fast) slope movements. In these cases, waking up the nodes of the network periodically would not make any sense and would waste a lot of energy. Nodes should rather sleep most of the time, and wake up only upon detectable slope movements.

Such adaptive sampling approaches challenge the underlying notion of time, but they tend to focus on the temporal dimension only (i.e., adapting to the dynamics of a phenomena as locally perceived by the nodes). Our approach goes further, by making it possible to adapt in time and space as well: not only how fast a phenomenon changes in time, but how fast it propagates and induces causal effects in space. For instance, in the case of landslide monitoring or crowd monitoring, adapting to the dynamics of local perceived movements to the overall propagation speed of such movements across the monitored area.

Besides sensor networks, the issue of adaptive sampling has recently landed in the broader area of IoT systems and applications [35], again with the primary goal of optimizing energy consumption of devices while not losing relevant phenomena under observation. However, unlike what promoted in sensor networks, such optimizations typically take place in a centralized (cloud) [34] or semi-decentralized (fog) way [18], which again disregards spatial issues and the strict space-time relations of phenomena.

Since coordination models and languages typical address a crosscutting concern of distributed systems, they are historically concerned with the notion of time in a variety of ways. For instance, time is addressed in space-based coordination since Javaspaces [12], and corresponding foundational calculi for time-based Linda [6,20]: the general idea is to equip tuples and query operations with timeouts, which can be interpreted either in terms of global or local clocks. The problem of abstracting the notion of time became crucial when coordination models started addressing self-adaptive systems, and hence openness and reactivity. In [25], it is suggested that a tuple may eventually fade, with a rate that depends on a usefulness concepts measuring how many new operations are related to such tuple. In the biochemical tuple-space model [38], tuples have a time-dynamic "concentration" driven by stochastic coordination rules embedded in the data-space.

Field-based coordination emerged as a coordination paradigm for self-adaptive systems focusing more on "space" rather than "time", in works such as TOTA [24], field calculus [3,37], and fixpoint-based computational fields [21]. However, the need for dealing with time is a deep consequence of dealing with space, since propagation in space necessarily impacts "evolution". These approaches tend to abstract from the scheduling dynamics of local field evolution, in various ways. In TOTA, the update model for distributed "fields of tuples" is an asynchronous event-based one: anytime a change in network connectivity is detected by a node, the TOTA middleware provides for triggering an update of the distributed field structures so as to immediately reflect the new situation. In the field calculus and aggregate computing [5] as already mentioned, an external, proactive clock is typically used. In [21] this issue is mostly neglected since the focus is on the "eventual behavior", namely the stabilized configuration of a field, as in [36]. For all these models, scheduling of updates is always transparent to the application/programming level, so the application designer cannot intervene on coordination so as to possible optimize communication, energy expenses, and reactivity.

3 Time-Fluid Field-Based Coordination

In this section, we introduce a model for time-fluid field-based coordination. The core idea of our proposed approach is to leverage the field-based coordination itself for maintaining a *causality field* that drives the dynamics of computation of the application-level fields. Our discussion is in principle applicable to any field-based coordination framework, however, for the sake of clarity, we here focus on the field calculus [3].

3.1 A Time-Fluid Model

Considering a field calculus program **P**, each of its rounds can be though of as consuming: *i)* a set of valid messages received from neighbors, $M \in \mathcal{M}$; and *ii)* some contextual information $S \in \mathcal{S}$, usually obtained via so-called sensors. The platform or middleware in charge of executing field calculus programs has to decide when to launch the next evaluation round of **P**, also providing valid values for $\mathcal{M}$ and $\mathcal{S}$. Note that in general the platform could execute many programs concurrently.

In order to support causality-driven coordination, we first require the platform to be able to reactively respond to *local event triggers*, each representing some kind of change in the values of $\mathcal{M}$ or $\mathcal{S}$—e.g., "a new message is arrived", "a given sensor provides a new value", or "1 second is passed". We denote by $\mathcal{T}$ the set of all possible local event triggers the platform can manage.

Then, we propose to associate to every field calculus program **P** a guard policy **G** (policy in short), which itself denotes a field computation—and can hence be written with a program expressed in the same language of **P**, as will be detailed in next section. Most specifically, whenever evaluated across space and time, the field computation of a policy can be locally modeled as a function

$$f_G \colon (\mathcal{S}, \mathcal{M}) \rightarrow (\{0, 1\}, \mathcal{P}(\mathcal{T}))$$

where $\mathcal{P}(\mathcal{T})$ denotes the powerset of $\mathcal{T}$. Namely, a policy has the same input of any field computation, but specifically returns a pair of Boolean $b \in \{0, 1\}$ and a set of event triggers $T_c \subseteq \mathcal{T}$. T_c is essentially the *set of "causes"*: **G** will get evaluated next time by the platform only when a new event trigger is detected that belongs to T_c. Then, such an evaluation produces the second output b: when this is true (value 1) it means that the program **P** associated to the policy must be evaluated as soon as possible. On system bootstrap, every policy gets evaluated for the first time.

In the proposed framework, hence, computations are caused by a field of event triggers (the *causality field*) computed by a policy, which is used to *i)* decide whether to run the actual application round immediately, and *ii)* decide which event triggers will cause a re-evaluation of the policy itself. This mechanism thus introduces a sort of guard mediating between the evolution of the *causality field* and the actual execution of application rounds, allowing for fine control over the actual temporal dynamics, as exemplified in Sect. 4.2.

Crucially, the ability to sense context (namely, the contents of $\mathcal{S}$) and to express event triggers (namely, the possible contents of $\mathcal{T}$) has a large impact on the expressivity of the proposed model. For the remainder of this work, we will assume the platform or middleware hosting a field computation to provide the following set of features, which we deem reasonable for any such platform—this is for the sake of practical expressiveness, since even a small set of event triggers could be of benefit. First, $\mathcal{T}$ must include changes to any value of $\mathcal{S}$; this allows the computation to be reactive to changes in the device perception, or, symmetrically speaking, makes such changes the *cause* of the computation. Second, timers can be easily modeled as special Boolean sensors flipping their value from `false` to `true`; making the classic time-driven approach a special case of the proposed framework. Third, which specific event trigger caused the last computation should be available in $\mathcal{S}$, accessible through the appropriate sensor. Fourth, the most recent result of any field computation $\mathbf{P}$ that should affect the policy must be available in $\mathcal{S}$; this is crucial for field computations to depend on each other, or, in other words, for a field computation to be the cause of another, possibly more intensive field computation. For instance, consider the crowd sensing and steering application mentioned in Sect. 1 to be decomposed in two sub-field computations: the former, lightweight, computing the local crowd density under a policy triggering the computation anytime a presence sensor counts a different number of people in the monitored area; the latter, resource intensive, computing a crowd steering field guiding people out of the over-crowded areas, whose policy can leverage the value of the density field to raise the evaluation frequency when the situation gets potentially dangerous. Fifth, the conclusion of a round of any field program is a valid source of event triggers, namely, $\mathcal{T}$ also contains a Boolean indicating whether a field program of interest completed its round.

3.2 Consequences

Programming *the* Space-Time and Propagating Causality. As soon as we let the application affect its own execution policy, we are effectively programming *the* time (instead of *in* time, as is typically done in field-based coordination): evaluating the field computation at different frequencies would actually amount at modulating the perception of time from the application standpoint. For instance, sensors' values may be sampled more often or more sparsely, affecting the perception that the application has of its operating environment along the time scale. In turn, as stemming from the distributed nature of the communicating system at hand, such an adaptation along time would immediately cause adaptation across space too, by affecting the communication rate of devices, hence the rate at which events and information spread across the network. It is worth emphasizing that this a consequence of embracing a notion of time founded on causality. In fact, as we are aware of computational models adaptive to the time fabric, as mentioned in Sect. 2, we are not aware of any model allowing *programming* the perception of time at the application level.

Adapting to Causality. Being able to program the space-time fabric as described above necessarily requires the capability of being *aware* of the space-time fabric in the first place. When the notion of space-time is crafted upon the notion of causality between events, such a form of awareness translates to awareness of the *dynamics of causal relations* among events. Under this perspective, the application is no longer adapting to the passage of time and the extent of space, but to the temporal and spatial distribution of causal relations among events. In other words, the application is able to "chase" events not only as they travel across time and space, but also as their "traveling speed" changes. For instance, whenever in a given region of space some event happens more frequently, devices operating in the same area may compute more frequently as well, increasing the rate of communications among devices in that region, thus leading to an overall better recognition of the quickening dynamics of the phenomenon under observation.

Controlling Situatedness. The ability to control both the above mentioned capabilities at the application level enables unprecedented fine control over the *degree of situatedness* exhibited by the overall system, along two dimensions: the ability to decide the granularity at which event triggers should be perceived; and the ability to decide how to adapt to changes in events dynamics. In modern distributed and pervasive systems the ability to quickly react to changes in environment dynamics are of paramount importance [32]. For instance, in the mentioned case of landslide monitoring, as anomalies in measurement increase in frequency, intensity, and geographical coverage, the monitoring application should match the pace of the accelerating dynamics.

Co-causal Field Computation. On the practical side, associating field computations to programmable scheduling policies brings both advantages and risks (as most extensions to expressiveness do). One important gain in *expressiveness* is the ability to let field computation affect the scheduling policy of other field computations, as in the example of crowd steering or landslide monitoring: the denser some regions get, the faster will the steering field be computed; the more intense vibrations of the ground get, the more frequently monitoring is performed. On the other hand, this opens the door to *circular dependencies* among fields computations and the scheduling policies, which can possibly lead to *deadlocks* or *livelocks*. Therefore, it is good practice for time-fluid field coordination systems that at least one field computation depends solely on local event triggers, and that dependencies among diverse field computations are carefully crafted and possibly enriched with local control.

Pure Reactivity and Its Limitations. Technically, replacing a scheduler guided by a fixed clock with one triggering computations as consequence of events, turns the system from time-driven to *event-driven*. In principle, this makes the system *purely reactive*: the system is idle unless some event trigger happens. Depending on the application at hand, this may be a blessing or a curse: since pro-activity is lost, the system is chained to the dynamics of event triggers, and cannot act on its own will. Of course, it is easy to overcome such a limitation: assuming a clock is available in the pool of event triggers makes pro-activity a particular case of reactivity, where the tick of the clock dictates the granularity. Furthermore, since policies allow the specification of a set of event triggers causing re-evaluation, the designer can always design a "fall-back" plan relying on expiration of a timer: for instance, it's possible (and reasonable) to express a policy such as "trigger as soon as ϵ happens, or timer τ expires, whichever comes first".

4 Time-Fluid Aggregate Computing

The proposed model has been prototypically reified within the framework of aggregate computing [5]. In particular, we leveraged the Alchemist Simulator [26]'s pre-existing support for the Protelis programming language [27] and the Scafi Scala DSL [39], and we produced a modified prototype platform supporting the definition of policies using the same aggregate programming language used for the actual software specification. The framework has been open sourced and publicly released, and it has been exercised in a paradigmatic experiment.

In this section we first briefly provide details about the Protelis programming language, which we use to showcase the expressive power of the proposed system by examples, then we present an experiment showing how the time-fluid architecture may allow for improved precision as well as reduced resource use.

4.1 A Short Protelis Primer

This Protelis language primer is intended as a quick reference for understanding the subsequent examples. Entering the language details is out of the scope of this work, only the set of features used in this paper will be introduced. Protelis is a purely functional, higher-order, interpreted, and dynamically typed aggregate programming language interoperable with Java.

Programs are written in modules, and are composed of any number of function definitions and of an optional main script. `module some:namespace` creates a new module whose fully qualified name is `some:namespace`. Modules' functions can be imported locally using the `import` keyword followed by the fully qualified module name. The same keyword can be used to import Java members, with `org.protelis.Builtins`, `java.lang.Math`, and `java.lang.Double` being pre-imported. Similarly to other dynamic languages such as Ruby and Python, in Protelis top level code outside any function is considered to be the main script.

`def f(a, b) { code }` defines a new function named `f` with two arguments `a` and `b`, which executes all the expressions in `code` upon invocation, returning the value of the last one. In case the function has a single expression, a shorter, Scala/Kotlin style syntax is allowed: `def f(a, b) = expression`.

The `rep (v <- initial) { code }` expression enables stateful computation by associating `v` with either the previous result of the `rep` evaluation, or with the value of the `initial` expression, The `code` block is then evaluated, and its result is returned (and used as value for `v` in the subsequent round).

The `if(condition) {then} else {otherwise}` expression requires `condition` to evaluate to a boolean value; if such value is `true` the `then` block is evaluated and the value of its last expression returned, while if the value of `condition` is `false` the `otherwise` code block gets executed, and the value of its last expression returned. Notably, `rep` expressions that find themselves in a non-evaluated branch lose their previously computed state, hence restarting the state computation from the initial value. This behavior is peculiar of the field calculus semantics, where the branching construct is lifted to a distributed operator with the meaning of domain segmentation [3].

The `let v = expression` statement adds a variable named `v` to the local name space, associating its value to the value of the `expression` evaluation. Square brackets delimit tuple literals: `[]` evaluates to an empty tuple, `[1, 2,"foo"]` to a tuple of three elements with two numbers and a string. Methods can be invoked with the same syntax of Java: `obj.method(a, b)` tries to invoke method `member` on the result of evaluation of expression `obj`, passing the results of the evaluation of expressions `a` and `b` as arguments. Special keywords `self` and `env` allow access to contextual information. `self` exposes sensors via direct method call (typically leveraged for system access), while `env` allows dynamic access to sensors by name (hence supporting more dynamic contexts).

Anonymous functions are written with a syntax reminiscent of Kotlin and Groovy: `{ a, b, -> code }` evaluates to an anonymous function with two parameters and `code` as body. Protelis also shares with Kotlin the *trailing lambda convention*: if the last parameter of a function call is an anonymous function, then it can be placed outside the parentheses. If the anonymous function is the only argument to that call, the parentheses can be omitted entirely. The following calls are in fact equivalent:

```
[1, 2].map({ a -> a + 1 }) // returns [2, 3]
[1, 2].map() { a -> a + 1 } // returns [2, 3]
[1, 2].map { a -> a + 1 } // returns [2, 3]
```

4.2 Examples

In this section we exemplify how the proposed approach allows for a single field-based coordination language to be used for expressing both **P** and **G**. In the following discussion, event triggers provided by the platform (i.e., members of $\mathcal{T}$), will be highlighted in green. In our first example, we show a policy recreating the round-based, classic execution model, thus demonstrating how this approach supersedes the previous. Consider the following Protelis functions, which detect changes in a value:

```
def updated(current, condition) = rep(old <- current) {
  if (condition(current, old)) { current } else { old }
} == current
def changed(current) = updated(current){cur, old -> cur!=old}
```

where `current` is the current value of the signal being tracked, and `condition` is a function comparing the current with the previously memorized value and returning `true` if the new value should replace the old one. Function `changed` is the simplest use of `update`, returning true whenever the input signal `current` changes. In the showcased code, the second argument to `updated` is provided using the trailing lambda syntax (see Sect. 4.1). They can be leveraged for writing a policy sensitive to platform timeouts. For instance, in the following code, we write a policy that gets re-evaluated every second (we only return `TIMER(1)` of all the possible event triggers in $\mathcal{T}$), and whose associated program runs if at least one second passed since the last round.

```
import platform.EventType.TIMER
[updated(self.getCurrentTime()) { now, last -> now-last>1 },
  [TIMER(1)]]
```

On the opposite side of the spectrum of possible policies is a purely reactive execution: the local field computation is performed only if there is a change in the value of any available sensors (`SENSOR(".*")`); if a message with new information is received (`MESSAGE_RECEIVED`); or if a message is discarded from the neighbor knowledge base (`MESSAGE_TIMEOUT`), for instance because the sender of the original message is no longer available:

```
import platform.EventType.*
let reason = env.get("platform.event")
[reason == MESSAGE_TIMEOUT || reason == SENSOR(".*") // Regex
  || changed(env.get("platform.neighborstate")),
[MESSAGE_RECEIVED, MESSAGE_TIMEOUT, SENSOR(".*")]]
```

Finally, we articulate a case in which the result of an aggregate computation is the cause for another computation to get triggered. Consider the crowd steering system mentioned in Sect. 1: we would like to update the crowd steering field only when there is a noticeable change in the perceived density of the surroundings. To do so, we first write a Protelis program leveraging the SCR pattern [8] to partition space in regions 300 meters wide and compute the average crowd density within them. Functions `S` (network partitioning at desired distance), `summarize` (aggregation of data over a spanning tree and partition-wide broadcast of the result), and `distanceTo` (computation of distance) come from the Protelis-lang library shipped with Protelis [11].

```
module io:github:steering:density
import ...
let distToLeader = distanceTo(S(300)) // network partitioning
// sum of all the perceived people
let count=summarize(distToLeader,env.get("people_count"),sum)
// computes an upper bound to the radius
let radius = summarize(distToLeader, distToLeader, max)
count/(2*PI*radius)//approximate crowd density as people/area
```

Its execution policy could be, for instance, reactive to updates from neighbors and to changes in a “people counting sensor”, reifying the number of people perceived by this device (e.g. via a camera).

```
import platform.EventType.*
let reason = env.get("platform.event")
[reason == MESSAGE_TIMEOUT || reason == MESSAGE_RECEIVED ||
changed(env.get("people_count")), [SENSOR("people_count")]]
```

Now that density computation is in place, the platform reifies its final result as a local sensor, which can in turn be used to drive the steering field computation with a policy such as:

```
import ...
let density = "io:github:steering:density"
[changed(exponentialBackOff(env.get(density),0.1)){cur,old->
  abs(cur - old) > 0.5
}, [SENSOR(density)]]
```

in which a low pass filter `exponentialBackOff` avoids to get the program running in case of spikes (e.g. due to the density computation re-stabilization). Note that access to the density computation is realized by accessing a sensor with the same name of the `module` containing the density evaluation program, thus reifying a causal chain between field computations.

4.3 Experiment

We exercise our prototype by simulating a distance computation over a network of situated devices. We consider a 40×40 irregular grid of devices, each located randomly in a disc centered on the corresponding position of a regular grid; and a single mobile node positioned to the top left of the network, free to move at a constant speed $\boldsymbol{v}$ from left to right. Once the mobile device leaves the network, exiting to the right side, another identical one enters the network from the left hand side. Mobile devices and the leftmost device at bottom are "sources", and the goal for each device is to estimate the distance to the closest source.

Computing distance from a source without a central coordinator in arbitrary networks is a representative application of aggregate computing, for which several implementations exist [36]. In this work, since the goal is exploring the behavior of the platform rather than the efficiency of the algorithm, we use an adaptive Bellman-Ford [9], even though it's known not to be the most efficient implementation for the task at hand [2]. We choose to compute the distance from a source (a gradient) as our reference algorithm as it is one of the most common building block over which other, more elaborate forms of coordination

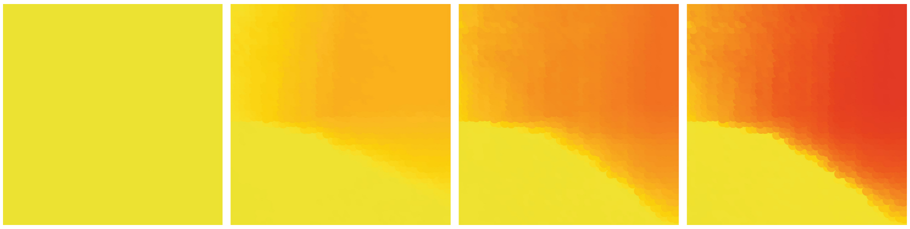

Fig. 1. Heat-map representation of executed rounds with time. Each device is depicted as a point located on its actual coordinates, time progresses from left to right. Devices start (left) with no round executed (yellow) and, with the simulation progression (left to right), execute rounds, changing their color to red. Devices closer to the static source (on the bottom left of the scenario) execute fewer rounds than those closer to the moving source, hence saving resources. (Color figure online)

get built [10,36]. We expect that an improvement in performance on this simple algorithm may lead to a cascading effect on the plethora [11] of algorithms based on it, hence our choice as a candidate for this experiment.

We let devices compute the same aggregate program with diverse policies. The baseline for assessing our proposal is the *classic* approach to aggregate computing: time-driven, unsynchronized, and fair scheduling of rounds set at 1 Hz. We compare the classic approach with time fluid versions whose policy is: *run if a new message is received or an old message timed out, and the last round was at least f^{-1} seconds ago.* The latter clause sets an upper bound to the number of event triggers a device can react to, preventing well-known limit situations such as the "raising value problem" for the adaptive Bellman-Ford [2] algorithm used in this work. We run several versions of the reactive algorithm, with diverse values for f; and we also vary $\|\boldsymbol{v}\|$. For each combination of f and $\|\boldsymbol{v}\|$, we perform 100 simulations with different random seeds, which also alter the irregular grid shape. We measure the overall number of executed rounds, which is a proxy metric for resource consumption (both network and energy), and the root mean square error of each device. The simulation has been implemented in Alchemist [26], writing the aggregate programs in Protelis [27]. Data has been processed with Xarray [14], and charts have been produced via matplotlib [15]. For the sake of reproducibility, the whole experiment has been automated, documented, and open sourced[1].

Intuitively, devices situated closer to the static source than to the trajectory of mobile sources should be able to execute less often. Figure 1 confirms such intuition: there is a clear border separating devices always closer to the static source, which execute much less often, from those that at times are instead closer to the mobile source. Figure 2 shows the precision of the computation for diverse values of $\|\boldsymbol{v}\|$ and f, compared to the baseline. The performance of baseline is equivalent with the performance of the time-fluid version with

[1] https://github.com/DanySK/Experiment-2020-Coordination-Time-Fluid-AC.

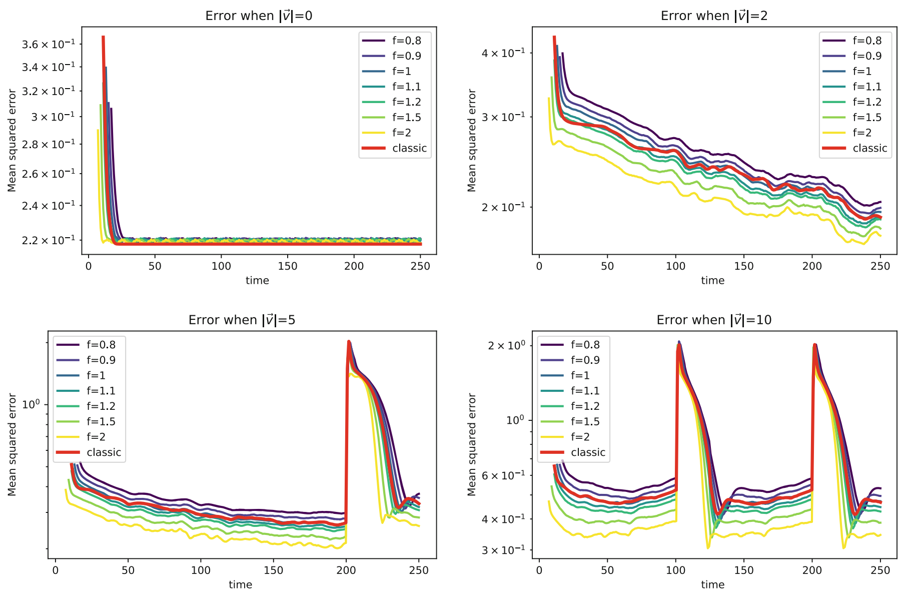

Fig. 2. Root mean squared error for diverse $\boldsymbol{v}$. When the network is entirely static (top left), after a short stabilization time the network converges to a very low error. Errors is lower with higher f values. The performance with $f = 1$ is equivalent with the performance of the baseline. When $\|\boldsymbol{v}\| \geq 5$, there is enough time for the mobile device to leave the system and for a new one to join, creating a spike in error and requiring a re-stabilization.

$f = 1\,\text{Hz}$. Higher values of f decrease the error, and lower values moderately increase it. Figure 3 depicts the cost to be paid for the algorithm execution. The causal version of the computation has a large advantage when there is nothing to recompute: if the mobile device is stands still, and the gradient value does not need to be recomputed, the computation is fundamentally halted. When $\|\boldsymbol{v}\| \neq 0$, the resource consumption grows; however, compared to the classic version, we can sustain $f = 1.5\,\text{Hz}$ with the same resource consumption. Considering that the performance of the classic version gets matched with $f = 1\,\text{Hz}$, and cost gets equalized at $f = 1.5\,\text{Hz}$, when $1\text{Hz} < f < 1.5\text{Hz}$ we achieve *both* better performance and lower cost. In conclusion, the time-fluid version provides a higher performance/cost ratio.

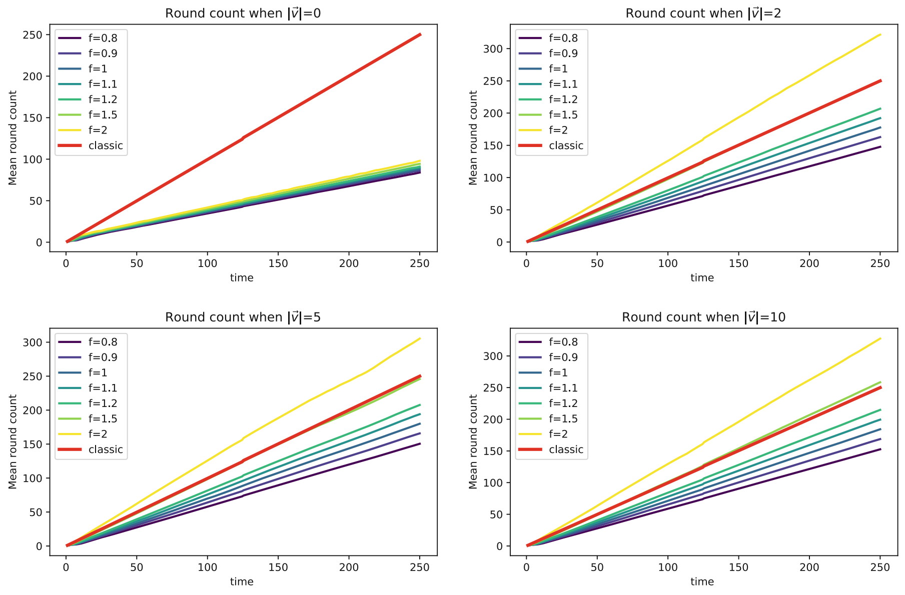

Fig. 3. Root mean squared error for diverse $\|\boldsymbol{v}\|$. When the network is entirely static (top left), raising f has a minimal impact on the overall cost of execution, as the network stabilizes and recomputes only in case of time outs. In dynamic cases, instead, higher f values come with a cost to pay. However, in the proposed experiment, the cost for the baseline algorithm matches the cost of the time fluid version with $f = 1.5\,\text{Hz}$, which in turn has lower error (as shown in Fig. 2).

5 Conclusion and Future Work

In this work we introduced a different concept of time for field-based coordination systems. Inspired by causal models of space-time in physics, we introduce the concept of *field of causality* for field computations, intertwining the usual coordination specification with its own actual evaluation schedule. We introduce a model that allows expressing the field of causality with the coordination language itself, and discuss the impact of its application. A model prototype is then implemented in the Alchemist simulation platform, supporting the execution of the aggregate computing field-based coordination languages Protelis, demonstrating the feasibility of the approach. Finally, the prototype is exercised in a paradigmatic experiment, highlighting the practical relevance of the approach by showing how it can improve efficiency—intended as precision in field evaluation over resource consumption.

Future work will be devoted to provide more in-depth insights by evaluating the impact of the approach in realistic setups, both in terms of scenarios (e.g. using real world data) and evaluation precision (e.g. by leveraging network simulators such as Omnet++ or NS3). Moreover, further work is required both for the current prototype to become a full fledged implementation, and for the model to be implemented in practical field-based coordination middlewares.

Acknowledgements. This work has been supported by the MIUR PRIN 2017 Project "Fluidware". The authors want to thank dr. Lorenzo Monti for the fruitful discussion on causality, the shape and fabric of space and time, and physical models independent of time.

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, September 2008. https://doi.org/10.1109/ispcs.2008.4659224
2. Audrito, G., Damiani, F., Viroli, M.: Optimal single-path information propagation in gradient-based algorithms. Sci. Comput. Program. **166**, 146–166 (2018). https://doi.org/10.1016/j.scico.2018.06.002
3. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Trans. Comput. Logic **20**(1), 1–55 (2019). https://doi.org/10.1145/3285956
4. Babaoğlu, O., Marzullo, K.: Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, pp. 55–96. ACM Press/Addison-Wesley Publishing Co., New York/Boston (1993)
5. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Comput. **48**(9), 22–30 (2015). https://doi.org/10.1109/MC.2015.261
6. Busi, N., Gorrieri, R., Zavattaro, G.: Process calculi for coordination: from Linda to JavaSpaces. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 198–212. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45499-3_16
7. Casadei, R., Fortino, G., Pianini, D., Russo, W., Savaglio, C., Viroli, M.: Modelling and simulation of opportunistic IoT services with aggregate computing. Future Gener. Comput. Syst. **91**, 252–262 (2018). https://doi.org/10.1016/j.future.2018.09.005
8. Casadei, R., Pianini, D., Viroli, M., Natali, A.: Self-organising coordination regions: a pattern for edge computing. In: Riis Nielson, H., Tuosto, E. (eds.) COORDINATION 2019. LNCS, vol. 11533, pp. 182–199. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22397-7_11
9. Dasgupta, S., Beal, J.: A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm. In: 55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, 12–14 December 2016, pp. 7282–7287 (2016). https://doi.org/10.1109/CDC.2016.7799393
10. Fernandez-Marquez, J.L., Serugendo, G.D.M., Montagna, S., Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. Nat. Comput. **12**(1), 43–67 (2013). https://doi.org/10.1007/s11047-012-9324-y
11. Francia, M., Pianini, D., Beal, J., Viroli, M.: Towards a foundational API for resilient distributed systems design. In: 2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, 18–22 September 2017, pp. 27–32 (2017). https://doi.org/10.1109/FAS-W.2017.116

12. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces: Principles, Patterns, and Practice. Addison-Wesley, Boston (1999)
13. Ho, Y., Huang, Y., Chu, H., Chen, L.: Adaptive sensing scheme using naive Bayes classification for environment monitoring with drone. IJDSN **14**(1), 1550147718756036 (2018). https://doi.org/10.1177/1550147718756036
14. Hoyer, S., Hamman, J.: Xarray: N-D labeled arrays and datasets in Python. J. Open Res. Softw. 5(1) (2017). https://doi.org/10.5334/jors.148
15. Hunter, J.D.: Matplotlib: a 2D graphics environment. Comput. Sci. Eng. **9**(3), 90–95 (2007). https://doi.org/10.1109/MCSE.2007.55
16. Kho, J., Rogers, A., Jennings, N.R.: Decentralized control of adaptive sampling in wireless sensor networks. TOSN **5**(3), 19:1–19:35 (2009). https://doi.org/10.1145/1525856.1525857
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563
18. 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, 3–6 July 2018, pp. 395–397 (2018). https://doi.org/10.1109/ICUFN.2018.8436601
19. de Lemos, R., et al.: Software engineering for self-adaptive systems: research challenges in the provision of assurances. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Software Engineering for Self-Adaptive Systems III. Assurances. LNCS, vol. 9640, pp. 3–30. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-74183-3_1
20. Linden, I., Jacquet, J.: On the expressiveness of timed coordination via shared dataspaces. Electron. Notes Theor. Comput. Sci. **180**(2), 71–89 (2007). https://doi.org/10.1016/j.entcs.2006.10.047
21. Lluch-Lafuente, A., Loreti, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. Log. Methods Comput. Sci. **13**(1) (2017). https://doi.org/10.23638/LMCS-13(1:13)2017
22. Lobo, F.S.: Nature of time and causality in physics. In: Psychology of Time, pp. 395–422. Emerald Group Publishing Limited, Bingley (2008)
23. Mamei, M., Zambonelli, F.: Field-Based Coordination For Pervasive Multiagent Systems. Springer Series on Agent Technology. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-27969-5
24. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: the TOTA approach. ACM Trans. Softw. Eng. Methodol. **18**(4), 15:1–15:56 (2009). https://doi.org/10.1145/1538942.1538945
25. Menezes, R., Wood, A.: The fading concept in tuple-space systems. In: Haddad, H. (ed.) Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, 23–27 April 2006, pp. 440–444. ACM (2006). https://doi.org/10.1145/1141277.1141379
26. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with Alchemist. J. Simul. **7**(3), 202–215 (2013). https://doi.org/10.1057/jos.2012.27
27. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, 13–17 April 2015, pp. 1846–1853 (2015). https://doi.org/10.1145/2695664.2695913
28. Rosi, A., et al.: Landslide monitoring with sensor networks: experiences and lessons learnt from a real-world deployment. IJSNet **10**(3), 111–122 (2011). https://doi.org/10.1504/IJSNET.2011.042195

29. Rovelli, C.: Quantum mechanics without time: a model. Phys. Rev. D **42**(8), 2638–2646 (1990). https://doi.org/10.1103/physrevd.42.2638
30. Rovelli, C.: Relational quantum mechanics. Int. J. Theor. Phys. **35**(8), 1637–1678 (1996). https://doi.org/10.1007/bf02302261
31. Rovelli, C.: Loop quantum gravity. Living Rev. Relativ. **1**(1) (1998). https://doi.org/10.12942/lrr-1998-1
32. Schuster, D., Rosi, A., Mamei, M., Springer, T., Endler, M., Zambonelli, F.: Pervasive social context: taxonomy and survey. ACM TIST **4**(3), 46:1–46:22 (2013)
33. Sundararaman, B., Buy, U., Kshemkalyani, A.D.: Clock synchronization for wireless sensor networks: a survey. Ad Hoc Netw. **3**(3), 281–323 (2005). https://doi.org/10.1016/j.adhoc.2005.01.002
34. 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, 24–27 September 2017, pp. 586–597 (2017). https://doi.org/10.1145/3127479.3131621
35. Trihinas, D., Pallis, G., Dikaiakos, M.: Low-cost adaptive monitoring techniques for the Internet of Things. IEEE Trans. Serv. Comput. 1 (2018). https://doi.org/10.1109/tsc.2018.2808956
36. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. **28**(2), 1–28 (2018). https://doi.org/10.1145/3177774
37. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From field-based coordination to aggregate computing. In: Di Marzo Serugendo, G., Loreti, M. (eds.) Coordination Models and Languages. COORDINATION 2018. LNCS, vol 10852, pp. 252–279. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92408-3_12
38. Viroli, M., Casadei, M.: Biochemical tuple spaces for self-organising coordination. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 143–162. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02053-7_8
39. Viroli, M., Casadei, R., Pianini, D.: Simulating large-scale aggregate MASs with Alchemist and Scala. In: Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, 11–14 September 2016, pp. 1495–1504 (2016). https://doi.org/10.15439/2016F407
40. Viroli, M., Pianini, D., Beal, J.: Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 212–229. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30829-1_15