TIME-FLUID FIELD-BASED COORDINATION THROUGH PROGRAMMABLE DISTRIBUTED SCHEDULERS

DANILO PIANINI [a], ROBERTO CASADEI [a], MIRKO VIROLI [a], STEFANO MARIANI [b], AND FRANCO ZAMBONELLI [b]

[a] Alma Mater Studiorum—Università di Bologna, Cesena (FC), Italy
e-mail address: danilo.pianini@unibo.it
e-mail address: roby.casadei@unibo.it
e-mail address: mirko.viroli@unibo.it

[b] Università degli Studi di Modena e Reggio Emilia, Reggio Emilia (RE), Italy
e-mail address: stefano.mariani@unimore.it
e-mail address: 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 are regulated by a fair and unsynchronised fixed clock working at the platform level. In this work, we challenge this assumption, and propose an alternative approach where scheduling is programmed in a natural way (along with usual field-based coordination) in terms of *causality fields*, each enacting a programmable 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 performance (system reactivity) and cost (resource usage) of computations. We formalise the proposed scheduling framework for field-based coordination in the context of the field calculus, discuss an implementation in the aggregate computing framework, and finally evaluate the approach via simulation on several case studies.

1. Introduction

Emerging application scenarios, such as cyber-physical systems (CPSs), the Internet of Things (IoT), and edge computing, call for software design approaches addressing openness, self-adaptation, heterogeneity, and deployment agnosticism [dLGG$^+$13]. To effectively address this issue, researchers strive to define increasingly higher-level concepts, reducing the "abstraction gap" with the problems at hand, e.g., by designing new languages and paradigms. In the context of coordination models and languages, field-based coordination is one such approach [MZ06, AVD$^+$19, VBD$^+$18, BPV15, LLM17, VPB12]. 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 is *situated* and holds a value) and time (each device continuously compute, updating its value).

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 (or aggregate program) 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 the state of the computation in the previous round, messages received by neighbours while sleeping, and contextual information (i.e. sensor readings); and *(iii)* uses such information to evaluate the coordination specification, storing state information in memory, producing an output value, and sending relevant information to neighbours. So far, field-based coordination approaches considered computation rounds as being regulated by local clocks, typically asynchronous with respect to clocks in other devices, and independent from the actual outcomes of computations: altogether, they may be seen as resulting from a fixed fair distributed scheduler working at the platform level. This assumption holds for many other distributed approaches to coordination (e.g. channel-based [Arb04], tuple-based [CROM19, OZ98], attribute-based [ADNL$^+$15]), but it has a number of consequences and limitations, both philosophical and pragmatic.

From 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* [Lob08], as being only meaningful relative to the existence of events as in *relational* interpretations of space-time [Rov96], or even being a mere derived concept introduced by our cognition [Rov90]—as in Loop Quantum Gravity [Rov98]. From a practical point of view, consequences are mixed. The key practical advantage for field-based coordination is simplicity. First, the designer can 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 mostly 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 neighbourhood, and simply modelling the state of communication by maintaining the most recent message received from each neighbour.

However, there is a price to pay for such simplicity. The first is that "stability" of the computation, namely, situations in which the field will not change after execution of a round, is ignored. As a consequence, "unnecessary" computations may be performed,

consuming resources (including, e.g., 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 [CFP⁺18]: in case the measured density of people gets dangerous, a more frequent evaluation of the field delivering steering advices is likely to provide more precise and timely recommendations. Similar considerations apply for example to the area of landslide monitoring [RBB⁺11], 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 when slope changes. This generally suggests a key unexpressed potential for field-based computation: the general ability to adaptively balance performance (system reactivity) and cost (usage of computational resources) of computations, and most importantly, to do so based on a high-level, domain-dependent concept of "cause" for computation. For instance, the crowd and landslide monitoring systems should ideally slow down (possibly, halt entirely) the evaluation in case of sparse crowd density or absence of surface movements, respectively; however, they should start being more and more responsive with growing crowd densities or in case of landslide activation.

So, we observe that the scheduling of distributed rounds of computation should not be fixed, but rather *(i)* dynamically depend on the outcome of the computation itself, *(ii)* be based on high-level notions of "cause", which can result from space-time properties of computation, and *(iii)* be programmable as a part of the coordination specification—and hence, possibly, using similar constructs. This general idea can be captured in field-based coordination by modelling time (i.e., the clock that regulates executions of computations) as a *causality field*, namely as a field programmable along with (and hence intertwined with) the usual coordination specification, dictating at each point in space-time where a computation round occurs, i.e., whether a given field computation has to be scheduled for execution or not. To allow for higher expressiveness, then, causality fields can be stacked in multiple layers, to model increasingly higher-level notions of time. 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 present a framework to support causality-driven field-based coordination in the context of the field calculus [AVD⁺19], a core calculus capturing the essential elements of so-called *aggregate computing* [BPV15, VBD⁺19]. Technically, this is achieved by the idea of organising field calculus programs into a tree structure, where leafs intercept platform triggers and hence enact low-level causality fields; then, these are used by parent programs to enact higher-level causality fields until scheduling the top-level program, which is considered the application-level one. At each level hence, a field calculus program is scheduled based on the outcome of its execution (self-regulation), on platform triggers (low-level, platform events), and on the execution outcome of leaf programs (called its schedulers, or also, guard policies).

More specifically, the contribution of this work (extending upon [PMVZ20]) can be summarised as follows:

- first, we propose a model that enriches field-based coordination approaches with the possibility to explicitly program the scheduling of coordination actions, by enabling a *functional description of causality*, relying on a tree-based composition to express high-level notions of time;

- second, we formally define the operational semantics of such a model in the context of the field calculus, as an extension and variation of its network semantics in [AVD$^+$19];
- third, we evaluate the proposed model in the context of aggregate programming, showcasing its expressiveness and practicality in achieving improved efficiency and, in some cases, performance.

The remainder of this work is structured as follows. Section 2 frames the work with respect to the existing literature on the topic. Section 3 informally describes the proposed model, discussing its goals, motivations, and potential implications. Section 4 formalises the proposed time-fluid model. Section 5 presents a prototype implementation in the framework of aggregate computing, along with practical programming examples. Section 6 evaluates the model and the prototype via simulation of relevant case studies. Finally, Section 7 discusses future directions and concludes the work.

2. Background and Related Work

Time and synchronisation have always been key issues in the area of distributed and pervasive computing systems. In fact, the absence of a globally shared physical clock among nodes makes it impossible to rely on absolute notions of time, and thus makes it hard to evaluate distributed properties.

To observe and evaluate the distributed state of a computation, logical clocks can be adopted [Lam78] to realise a sort of causally-driven notion of time: 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 [BM93]. Our proposal somewhat goes in that direction, by trying to make coordination activities abstract from the passing of physical time and rather be guided by some domain-specific notion of time.

Many algorithms exist for computing in a distributed way aggregation functions over data (e.g., local variables or locally sensed values) held by a set of distributed processes [JMB05]. However, when such data dynamically vary with time, the aggregation functions must be periodically recomputed, which introduces the problem of identifying the appropriate frequency at which to recompute them. Most solutions typically adopt a predefined frequency [JMB05, BMZ12], but this strategy does not take into account that such frequency should be properly tuned to match the dynamics of the data being aggregated. Also, synchronising processes to let them organise aggregation in successive rounds is far from easy in asynchronous systems lacking a shared physical clock [HHM$^+$00, Fis83]. Our proposal starts from similar problems, but is specifically conceived for distributed field-based aggregation, 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 [SBK05] 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. 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 synchronisation in wireless sensor networks [AMF08, KRJ09, HHCC18], 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, by adapting to the dynamics of a phenomena as locally perceived by the nodes. Instead, there is need of going 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, this amounts to 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 [TPD18], 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) [TBR+17] or semi-decentralised (fog) way [LYC18], 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 [FHA99], and corresponding foundational calculi for time-based Linda [BGZ00, LJ07]: 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 [MW06, MO12], it is suggested that a tuple may eventually fade, with a rate that depends on a usefulness concept measuring how many new operations are related to such tuple. In the biochemical tuple-space model [VC09], 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 [MZ09], field calculus [AVD+19, VBD+18], and fixpoint-based computational fields [LLM17]. However, the need for dealing with time is a deep consequence of dealing with space, since propagation in space necessarily implies "evolution" along time (as travelling in space requires time). 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 [BPV15], as already mentioned, an external, proactive clock is typically used. In [LLM17]
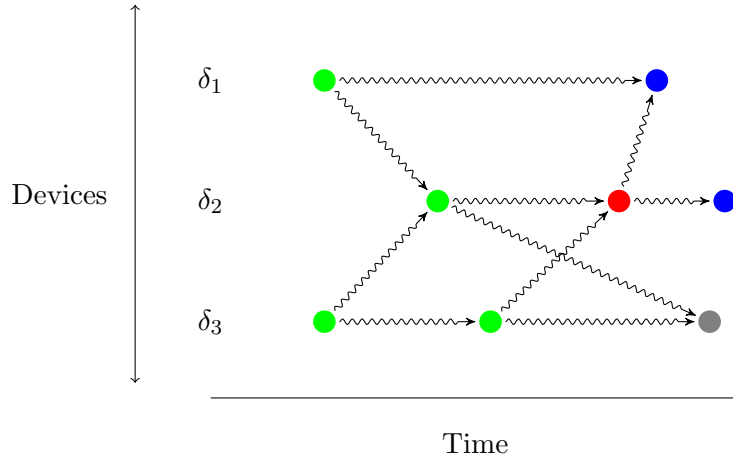
Figure 1: Example of an event structure. Each node is an event (i.e., a computation round happening at a given device), and the curly arrows denote the causal relationship between events resulting from communication, which defines a partial order on the events. Given a reference event (red), through 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.).

this issue is mostly neglected since the focus is on the "eventual behaviour", namely the stabilised configuration of a field, as in [VAB+18]. For all these models, scheduling of updates is always transparent to the application/programming level, so the application designer cannot intervene on the relationship between the passing of time and coordination actions, hence cannot possibly optimise communications, energy expenses, and reactivity depending on the dynamics of the application at hand.

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 field-based coordination itself for maintaining a *causality field* that drives the dynamics of computations 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 general framework originated from the field calculus [AVD+19].

Execution of programs in the field calculus are assumed to result in so-called *computational fields*, which are mappings from *events* to computational values, where an event

is seen as a space-time point (as in physics), namely, as the execution of a computation round at a given device. Communication can happen between events, which are hence called *neighbour events*; so, the execution of a system of computing and interacting devices can be modelled as a directed acyclic graph of events, also known as an *event structure* [M+88]—see Figure 1 for a graphical example. Locally, an event is typically scheduled by the middleware platform sustaining field computations, and is hence considered as a low-level (asynchronous, distributed) "clock", on top of which higher-level, fluid notions of time can be defined. In the case of a landslide monitoring distributed application, for instance, the events could be due to sensors' readings measuring whether the terrain is moving, how fast, etc., and the corresponding computational field could be the probability of a landslide being triggered in the next T minutes.

3.1. **A time-fluid model.** Considering a field calculus program $\mathbf{P}$, each of its rounds can be thought of as consuming: *i)* a set of valid messages received from neighbours, $M \in \mathcal{M}$; and *ii)* some contextual information $S \in \mathcal{S}$, usually obtained via so-called sensors. In the case of landslide monitoring, S could include measurements about terrain movements, and M could include messages about how neighbouring sensors measured such movements in their areas. The platform or middleware in charge of executing field calculus programs has to decide whether at a given event it should launch the next evaluation round of $\mathbf{P}$, 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 associate to each event a *(local event) trigger*, representing the cause for that event to happen, typically involving some kind of change at the application level (in $\mathcal{M}$) or at the physical level (in $\mathcal{S}$). Typical examples of triggers include "a new message has arrived", "a given sensor provides a new value" (as in landslide monitoring), or "1 second has passed". We here denote by $\mathcal{T}$ the set of all possible local event triggers the platform can manage.

As a second step, we introduce a specific type of field calculus program G, called a *guard policy*, which will be used as a scheduler for a "parent" field computation: as such it will be expressed in the same language of $\mathbf{P}$, as detailed in next sections. More specifically, whenever evaluated across space and time, a guard policy can be locally viewed as a function f_G of the kind $f_G \colon (\mathcal{S}, \mathcal{M}) \to \mathcal{V}$. Namely, a policy has the same input as any field computation, and can produce any output (values in $\mathcal{V}$).

Differently from the work in [PMVZ20], where a field computation could be assigned a single guard policy, here we extend this approach by allowing many guard policies to be associated to an actual field computation, and since guard policies are particular kinds of field computations as well, we also allow guard policies to be stacked at multiple layers, forming a *tree structure*. So, essentially, platform triggers schedule low-level guard policies, the leafs of the tree, which are field computations enacting low-level (causality-based) "clocks". Such clocks can be combined together and feed higher-level guard policies so as to enact higher-level "clocks", recursively. Ultimately, such clocks will then schedule evaluation of the top-level field computation, the root of the tree, which is the application-level one. In our landslide monitoring example, platform triggers may be the events related to the aforementioned sensors readings, low-level guard policies may use such readings to decide whether to compute the probability of an imminent landslide or not (which is a field computation), and higher levels scheduling policies could use such probability to compute a landslide frontier and visualise it on a map. Additionally, the result of evaluation of a guard
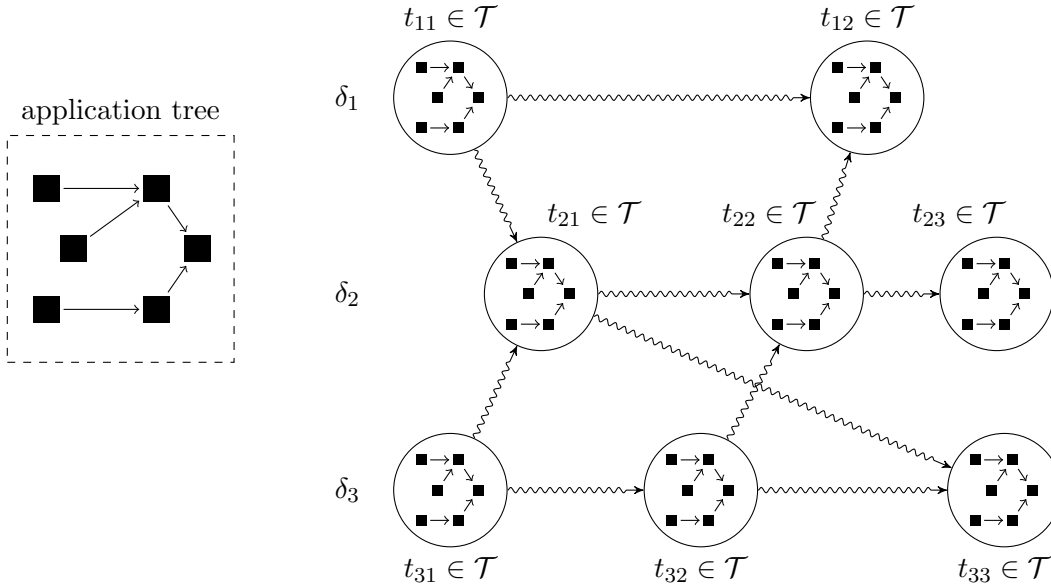
Figure 2: This is an example of an event structure (structurally equivalent to that of Figure 1) where each event (circle) is an evaluation of a tree of field calculus programs (the tree of black squares shown in the dashed box on the left). The labels on the nodes indicate the platform triggers causing 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.

policy at a given time can also be used as a feedback to schedule its next evaluation: this is used all the times the dynamics of evolution of a certain field computation should be used to self-regulate its actual timing. Consider Figure 2 for a graphical example.

The mechanisms by which the result of a guard policy affects scheduling are captured by the following function, defined at the platform level, called the *causality function*:

$$f_C \colon (\mathcal{T}, \mathcal{V}, \mathcal{V}_1, \ldots, \mathcal{V}_n) \mapsto \{0, 1\}$$

In a given round, and for a given program P (hence also for guard policies), it takes the trigger of the current round in the first argument, the result of evaluating P at the previous round (feedback) in second argument, and then the results of evaluating all its n lower-level schedulers, and returns a Boolean stating whether this program is to be scheduled for evaluation at the current round. Hence, function f_C maps any event to Booleans, hence denoting a Boolean field acting as a *causality field* for P, namely, enacting a programmable, distributed and self-regulated clock achieved by the collective behaviour of all the involved devices.

In the proposed framework, a field of low-level triggers is processed by a hierarchical set of schedulers, ultimately defining a top-level causality-field that schedules the application-level computation. This mechanism thus overall introduces a structured guard mediating between the evolution of *platform triggers* and the actual execution of application rounds, allowing for fine control over the actual temporal dynamics, as exemplified in Section 5.3.

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. In general, it is reasonable to assume that a platform or middleware hosting a field computation can generate events due to triggers that 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) and that triggers flipping their value from `false` to `true` can model timers, making the classic time-driven approach a special case of the proposed framework (see Section 5.3.1 for a concrete example).

3.2. **Consequences.** The above informal introduction to our proposed model allows us to emphasise in more detail some of the implications on expressiveness of field-based coordination.

3.2.1. *Programming* the *space-time and propagating causality.* As soon as we let an 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 a field computation at different frequencies actually amounts 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 emphasising 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 Section 2, we are not aware of any model allowing *programming* the perception of time at the application level.

3.2.2. *Adapting to causality.* Being able to affect 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.

3.2.3. *Controlling situatedness.* The ability to control both the above mentioned capabilities at the application level enables 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 [SRM$^+$13]. 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.

3.2.4. *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 computations 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 (provided that field computations can perform *actuation* which in turn results into changes in sensor values) this opens the door to *circular dependencies* among field computations and scheduling policies. Although the scheduling graph is always acyclic, in fact, inter-round communication could be carried out through sensors and actuators. This situation may lead to undesirable global behaviours, such as deadlocks or livelocks; however, such circular dependencies must be explictly programmed: they appear by design, not by chance. Hence, the programmer should design carefully the scheduling policies of the system at hand not to incur in these problematic situations—a care that should be taken when programming in any expressive language. In this regard, "fragments" of composable Protelis programs which are "safe" to run can be detected, as something similar has already been done with field calculus [VAB$^+$18]. Finally, for any case in which circular dependencies are actually desired by the designer, simulations can be used for assessing (i.e., with a reasonable confidence) whether the system does not incur in unpredicted behaviours.

3.2.5. *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 that 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 guard policies allow the specification of retroactive feedback on their scheduling, the designer can always design a "fall-back" plan relying on expiration of a timer: 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".

a) Field calculus syntactic elements:

v	field calculus value		e	field calculus expression
θ	field calculus value-tree		δ	device unique identifier
n	sensor name			

b) Field calculus abstracted semantics:

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

c) Local and scheduling configurations:

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

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

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

[ROUND] $\delta_j; \overline{\delta} \mapsto \overline{\theta}[\overline{\mu}^1, \ldots, \overline{\mu}^n]; \sigma \vdash \mathtt{e}[\overline{\pi}] \Downarrow^s \theta[\overline{\mu}]$

e) System configurations:

I	::=	$\overline{\delta}$	neighbourhood
τ	::=	$\overline{\delta} \mapsto \overline{I}$	topology field
Σ	::=	$\overline{\delta} \mapsto \overline{\sigma}$	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: $\boxed{N \xrightarrow{act} N}$

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

[N-FIR]

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

[N-ENV]

Figure 3: Operational semantics for time-fluid coordination in the field calculus.

4. Formal Semantics in Field Calculus

We now formalise our proposal of time-fluid field-based coordination. It is based on the field calculus [AVD+19], the prominent formal framework to capture the essential aspects of computational fields. This calculus is at the core of implementations such as Protelis [PVB15, BPV15], which will in turn be used in the next section to explain the impact on aggregate programming.

Figure 3 provides the whole formalisation, as a variation and extension of the network semantics for the field calculus presented in [AVD+19]. Starting from a black-box description of the field calculus syntax and semantics, and ending up with the operational semantics of network evolution, each part is described in the following.

The formalisation to come adopts standard conventions used in other calculi for programming languages (like Featherweight Java and its descendants [IPW01]): we use the overbar notation to represent sequences, such that if e is a meta-variable over expressions, then a sequence of n expressions is denoted $\bar{e}$ or equivalently $e_1, \ldots, e_n$, and a list of zero elements is sometimes denoted by symbol $\bullet$. The notation is then abused using the overbar notation over 2 (or more) symbols in a term to mean a sequence of terms constructed out of those 2 (or more) sequences, and this is typically used to model functions as sequence of mappings between pairs of terms. As an example, a field will be written as $\bar{\delta} \mapsto \bar{v}$ to mean $\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n$. This models a function ϕ that associates a device identified by δ_j to value v_j (for any j), hence we shall also use the natural function application notation $\phi(\delta_j)$ to mean v_j. We then use the update operator $\phi[\phi']$ for functions, to mean the function obtained by updating any mapping already present in ϕ with a possible new value as indicated in ϕ'. For instance, $(\delta_1 \mapsto v_1, \delta_2 \mapsto v_2)[\delta_2 \mapsto v'_2, \delta_3 \mapsto v_3]$ gives $\delta_1 \mapsto v_1, \delta_2 \mapsto v'_2$.

4.1. Field calculus syntactic elements and abstracted semantics.
The key idea of the field calculus is to represent the behaviour of a distributed program in term of a functional expression, whose denotational semantics is defined in terms of a resulting *computational field*, namely, a map from space-time events (specific moments of time in which a specific device performs a computation) to computational values (the result of such computation). More specifically, such expressions (ranged over by e) comprise usual mechanisms of function definition and call, use of built-in operators (for working with arithmetic, logic, and useful data structures), access to local sensors (whose name is ranged over by n, each providing values ranged over by v), as well as field operators to deal with evolution in time (`rep`), interaction with neighbours (`nbr`), and space-time branching (`if`). The actual syntax of field calculus is of no interest here, for it is orthogonal to the management of schedulers we ought to formalise; though, the examples and case studies presented later in this paper will make use of its incarnation in the Protelis language, conveniently described in the next section.

Expressions are to be understood as being repetitively and asynchronously evaluated in each device of the network, that is, in computation rounds. A network is considered equipped with a dynamically evolving and reflexive *neighbourhood* relation, supported at the platform level: a device can send messages only to its current neighbours. When a round occurs at a device with id δ, a context is available that includes messages received from its neighbours (only the last message from each neighbour is considered) and locally sensed information: the expression is evaluated against such a context (namely, is affected by it) and an output message is produced, which is assumed to be broadcasted to all the neighbours in turn. Crucially, such output messages – called *value-trees*, or *vtrees* in short, ranged over

by θ – are structured as a tree of values, essentially reflecting the dynamic unfolding of expression evaluation. For instance, the vtree of expression $\mathtt{e}_1 + \mathtt{e}_2$ is a tree whose root is tagged with the resulting value $\mathtt{v}$ of expression evaluation, and whose two sub-trees are the vtrees of $\mathtt{e}_1$ and $\mathtt{e}_2$, recursively—and similarly for any other built-in operator or construct of the language. A device gathers the vtrees of neighbours shipped with messages (there including the vtree of its own latest local round) into a so-called *value-tree environment* (a mapping $\overline{\delta} \mapsto \overline{\theta}$ from neighbours to vtrees). Since a value-tree environment is available during expression evaluation, it is then possible to deal with fields expressing evolution over time and communication with neighbours. Additionally, to support the branching mechanism of field calculus [AVD$^+$19], it is possible to distinguish, for each sub-expression, which neighbours evaluated it or not due to branching, correctly restricting observation of values in neighbours (as of construct $\mathtt{nbr}$)[1]. The result of an expression evaluation (the root of the vtree) at each device at each moment of time is what defines the computational field resulting from "distributed execution" of that expression.

These mechanisms are precisely captured by the field calculus round semantics, defined by a judgment of the form $\delta; \overline{\delta} \mapsto \overline{\theta}; \overline{\mathtt{n}} \mapsto \overline{\mathtt{v}} \vdash \mathtt{e_{main}} \Downarrow \theta$ to be read as "expression $\mathtt{e_{main}}$ evaluates to value-tree θ on device δ with respect to the value-tree environment $\overline{\delta} \mapsto \overline{\theta}$ and sensor state $\overline{\mathtt{n}} \mapsto \overline{\mathtt{v}}$". This judgment is defined by syntax-directed rules of a big-step operational semantics—the reader can refer to [AVD$^+$19] for details.

Example 4.1 (Minimum temperature in neighbourhood). Consider a simple program expression, $\pi_{\mathtt{main}} = \mathtt{fold}(\infty, \mathtt{min}, \mathtt{nbr}(\mathtt{temp}))$, where each device in the network computes the minimum temperature value sensed in its neighbourhood, assuming every device has a built-in sensor $\mathtt{temp}$ available for local temperature sensing. It works by accumulating the neighbours' temperature values with built-in function $\mathtt{fold}$ applied to the minimum function $\mathtt{min}$ and starting value ∞; neighbour values are collected through field calculus construct $\mathtt{nbr}(e)$, where e is the expression whose value is to be shared. As per the field calculus big-step operational semantics, program $\pi_{\mathtt{main}}$ evaluates to a vtree as follows.

After a device evaluates $\pi_{\mathtt{main}}$, it shares the resulting vtree with neighbours for supporting coordination and contributing to the "aggregate" system behaviour. Now, suppose we have three devices $\delta_1, \delta_2, \delta_3$ and to evaluate the program in event $\epsilon_{1,2}$ (the second event of δ_1), which can use information from $\epsilon_{1,1}$ (the previous event of the same device) as well as $\epsilon_{2,1}$ and $\epsilon_{3,1}$ (the first event of devices δ_2 and δ_3)—see the example event structure above. Suppose such input information consists of the mapping $\overline{\delta} \mapsto \overline{\theta}$ from neighbour devices to vtrees where $\delta_1 \mapsto \theta_{1,1} = 14\langle\infty, \mathtt{min}, (\delta_1 \mapsto 14)\langle 14\langle \mathtt{temp}\rangle\rangle\rangle$ (the output of the previous computation),

---

[1] This latter notion, called *alignment*, does not play a crucial role in this paper, but is key to enable safe composition of expressions in field calculus: the idea is that different branches of computation are unrelated and, given a certain point in the computation path, only the devices that reached that point during evaluation are actually allowed to interact (these devices are said to be *aligned* in that vtree position).

$\delta_2 \mapsto \theta_{2,1} = 15\langle\infty, \texttt{min}, (\delta_2 \mapsto 15)\langle 15\langle\texttt{temp}\rangle\rangle\rangle$, $\delta_3 \mapsto \theta_{3,1} = 12\langle\infty, \texttt{min}, (\delta_3 \mapsto 12)\langle 12\langle\texttt{temp}\rangle\rangle\rangle$, and that the $\texttt{temp}$ sensor yields 13 in $\epsilon_{1,2}$. Then, we have the big-step "$\delta_1; \{\delta_1 \mapsto \theta_{1,1}, \delta_2 \mapsto \theta_{2,1}, \delta_3 \mapsto \theta_{3,1}\}; \{\texttt{temp} \mapsto 13\} \vdash \texttt{e}_{\texttt{main}} \Downarrow 12\langle\infty, \texttt{min}, (\delta_1 \mapsto 13, \delta_2 \mapsto 15, \delta_3 \mapsto 12)\langle 13\langle\texttt{temp}\rangle\rangle\rangle$", where the root value 12 is the output of the computation.

4.2. **Local scheduling configuration and semantics.** We now describe the formalisation of scheduling as proposed in this paper, extending on top of the field calculus standard syntax and semantics. A global field-calculus scheduled program π has the form $\texttt{e}[\overline{\pi}]$: it consists of a standard expression $\texttt{e}$ scheduled by zero, one, or many scheduling programs $\overline{\pi}$. Therefore, a program is essentially a tree of expressions where the root expression is used to provide the result of field computation, child nodes are expressions used as schedulers for parents (possibly organised in many layers), and finally leafs have the form $\texttt{e}[\bullet]$—they are expressions directly scheduled by the platform as usual in field calculus. Given that a scheduled program is a tree of expressions, and evaluation of each expression is essentially isolated, devices will now exchange not just a vtree, but rather an exported message μ with a corresponding structure $\theta[\overline{\mu}]$, that is, a tree of elements θ, each obtained by node-wise application of the field calculus operational semantics recalled in Section 4.1 to the tree. We are hence ready to define the operational semantics of scheduling, which is given in terms of a judgment "$\delta; \Lambda; \sigma \vdash \pi \Downarrow^s \mu$" to be read as "scheduling program π evaluates to the exported message μ on device δ with respect to the local status field Λ and sensor state σ", where a local status field Λ is a mapping $\overline{\delta} \mapsto \overline{\mu}$ from neighbours to messages, and sensor state σ is a mapping $\overline{\texttt{n}} \mapsto \overline{\texttt{v}}$ as usual.

Informally, the idea is that, at any round, expressions will all be *considered* for evaluation bottom-up in the tree. Evaluation of an expression will actually take place (that is, its field computation is actually scheduled), depending on a *scheduling context* composed by (i) the results of evaluations of its low-level scheduler programs, (ii) the result of evaluation of the same expression at its latest round, and (iii) information about the trigger that caused the platform to start this round. If scheduling is activated, the expression is evaluated as usual and the resulting vtree will be added to the export message; if it is not activated, the latest vtree is simply reused without re-evaluation.

Formalisation of these two cases is provided by rule [ROUND] in Figure 3 (d). It deals with a program $\texttt{e}[\overline{\pi}]$ executed against a neighbourhood $\overline{\delta}$ where each device δ_k previously sent message $\theta_k[\mu_k^1, \ldots, \mu_k^n]$—hence altogether the local status field is written $\overline{\delta} \mapsto \overline{\theta}[\overline{\mu}^1, \ldots, \overline{\mu}^n]$. Then, it proceeds evaluating each scheduler $\pi_i \in \overline{\pi}$ recursively, each yielding its part of the exported message μ_i whose top-value tree is named θ_i^o: note that to do so we should enter the local field status accordingly, using $\overline{\delta} \mapsto \overline{\mu}^i$ for any i. Following, we check whether the main expression $\texttt{e}$ of current program $\texttt{e}[\overline{\pi}]$ can be evaluated: this is achieved by predicate $\texttt{sch}()$, whose evaluation on all events realise the causality field for top level expression $\texttt{e}$ as discussed in previous section. It takes the vtrees obtained from evaluation of schedulers, the vtree of current device at latest round (the current device is δ_j in list $\overline{\delta}$, hence its latest vtree is θ_j as can be extracted from the local status field), and the value read from $\texttt{trigger}$ sensor. Note that we abstract from this predicate: it depends on the specific incarnation of the model, which should dictate how to extract and combine meaningful information for scheduling from the aforementioned scheduling context.

Rule [ROUND] then differentiates based on whether the predicate prevents evaluation or not. In the first case, simply the latest vtree θ_j is used for the exported message, since

no evaluation of e takes place; in the second case instead, expression e is evaluated using standard field calculus approach as described in previous subsection, which gives θ as result, and which is then used to create the resulting export message $\theta[\overline{\mu}]$.

4.3. **System configuration and Network semantics.** On top of the scheduling semantics, dictating which expressions of the entire scheduling program get actually evaluated at each round, and what is the shape of the exported message, it is possible to derive an operational semantics of the overall distributed system behaviour, directly following the approach presented in [AVD+19]—which we tailor to the end of formalising scheduling. As with standard field calculus, it is then assumed that a single aggregate program $\pi_{\texttt{main}}$ exists. As shown in Figure 3(e), we let I range over set of devices (typically forming a neighbourhood), and introduce three kinds of fields used to define the snapshot of a system configuration at a given time: τ is a field of neighbours (a map from devices to their neighbours), representing system topology; Σ is a field of sensor state (a map from devices to sensor states σ), representing information sensed from the environment; and Ψ is a global status field (a map from devices to local status field Λ), representing the pending messages available across the system, and hence, the actual status of field computations over at a given time. We shall write $\mu_\perp^{\pi_{\texttt{main}}}$ to denote a default (empty) message compliant with program $\pi_{\texttt{main}}$: it will be automatically available in the global status field when a new device enters the network. A network configuration N is hence a pair of an environment Env and a global status field Ψ, where the environment is itself a pair of topology and sensor fields.

As described in Figure 3(f) an environment is considered well-formed if the sensor field has the same domain of the topology field, and a topology is reflexive and closed under such a domain. The operational semantics, defined in Figure 3(g) is constructed so as to guarantee that well-formedness of the environment is preserved across transitions. This is defined by two rules, one modelling scheduling ([N-FIR]) and one changes in the environment ([N-ENV]).

Rule [N-FIR] is labelled $\delta : \texttt{v}_t$, modelling scheduling of a round at device δ with trigger $\texttt{v}_t$. It performs one round of evaluation of $\pi_{\texttt{main}}$ using the local context of δ ($\Psi(\delta); \Sigma(\delta)$), and adding to the sensor state the mapping $\texttt{trigger} \mapsto \texttt{v}_t$. To model emission of the resulting export message μ to neighbours, we update the global status field with a new component Ψ_1, which adds to each neighbour of δ the term $\delta \mapsto \mu$, representing message μ received from δ.

Rule [N-ENV] is labelled *env*, and models a generic change in the environment (topology or sensors), moving from Γ to any Γ' that is well-formed. Let the new topology be the mapping $\overline{\delta} \mapsto \overline{I}$, it first constructs a complete global status field Ψ_0 with compliant empty messages $\mu_\perp^{\pi_{\texttt{main}}}$ available everywhere needed, namely, each device has one such message per neighbour. We then update Ψ_0 with the previous global status field Ψ.

Example 4.2 (Network semantics and scheduling). Consider the event structure in Figure 4. This (and any other) event structure can be modelled as a small-step evolution of a global state, also keeping track of the payload of messages and data available in any event (not shown graphically). For instance, the topology τ of a global time instant corresponding to the first event of δ_2 may be given by $\{\delta_1 \mapsto \{\delta_1, \delta_2\}, \delta_2 \mapsto \{\delta_1, \delta_2, \delta_3\}, \delta_3 \mapsto \{\delta_2, \delta_3\}\}$. Rule [N-ENV] can be used to evolve topology, sensor state, and status field to represent the state of a system and a device's local context corresponding to a particular event.

Each event represents an application of rule [N-FIR], which can fire if the sensor $\texttt{trigger}$ local to a device provides a value. Suppose that, beside $\texttt{trigger}$, the devices have
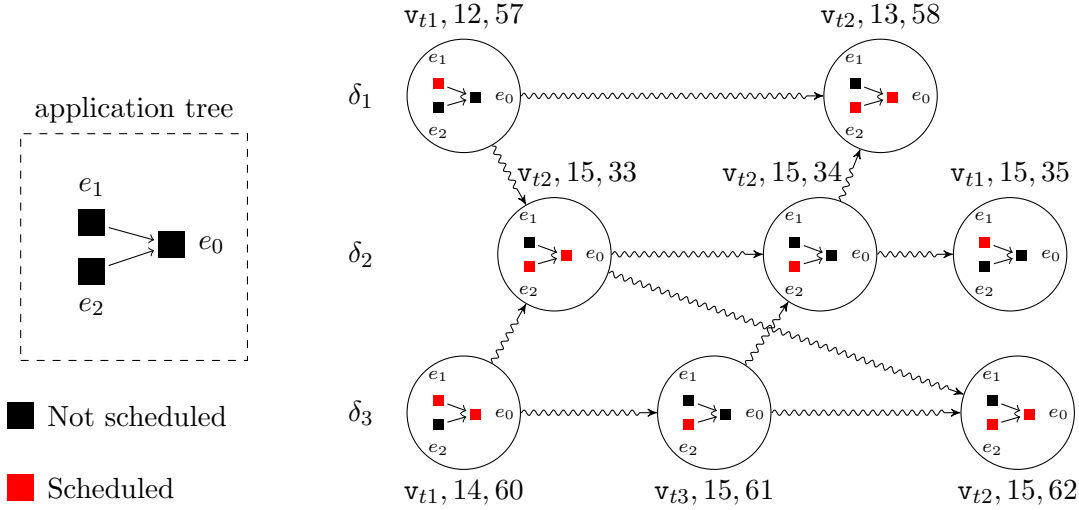
Figure 4: Application tree and event structure for Example 4.2. The labels of events denote sensor values for the three sensors `trigger`, `temp`, `time`.

a temperature sensor `temp` and a timestamp sensor `time`. The sensor field Σ may initially be as follows: $\{\delta_1 \mapsto \{\texttt{trigger} \mapsto \mathtt{v}_{t1}, \texttt{temp} \mapsto 12, \texttt{time} \mapsto 57\}, \delta_2 \mapsto \{\texttt{temp} \mapsto 15, \texttt{time} \mapsto 33\}, \delta_3 \mapsto \{\texttt{trigger} \mapsto \mathtt{v}_{t1}, \texttt{temp} \mapsto 14, \texttt{time} \mapsto 60\}\}$, showing that δ_1 and δ_3 can fire (but not δ_2 yet).

Then, consider a scheduled program $\pi_{main} = \mathtt{e}_0[\mathtt{e}_1, \mathtt{e}_2]$. The platform supports two local triggers, i.e., sensor `trigger` returns either $\mathtt{v}_{t1}$ or $\mathtt{v}_{t2}$.

- $\mathtt{e}_1$ is fired by a "local timestamp update" trigger $\mathtt{v}_{t1}$ and computes a vtree θ_1 whose root is `True` if `time` mod 60 equals 0 or `False` otherwise (hence yielding a field which is `True` once a minute);
- $\mathtt{e}_2$ is fired by either a "message received" trigger $\mathtt{v}_{t2}$ or a "tempature has changed" trigger $\mathtt{v}_{t3}$, and computes a vtree θ_2 whose root is `True` if the minimum temperature value in the neighbourhood (cf. Example 4.1) has changed from the last round;
- $\mathtt{e}_0$ may be a program that outputs a log message about the minimum temperature in the neighbourhood, to be scheduled if any of the outputs of $\mathtt{e}_1$ or $\mathtt{e}_2$ is `True`.

In particular, the scheduling predicate $\mathtt{sch}(\theta_j, \overline{\theta}^o, \sigma(\texttt{trigger}))$ can be as follows:

$$
\begin{cases}
\texttt{True} & \text{if } \overline{\theta}^o \neq \emptyset \wedge \text{any of } \overline{\theta}^o \text{ yields } \texttt{True} \\
\texttt{True} & \text{if } \overline{\theta}^o = \emptyset \wedge \sigma(\texttt{trigger}) \in \{\mathtt{v}_{t1}, \mathtt{v}_{t2}, \mathtt{v}_{t3}\} \\
\texttt{False} & \text{otherwise}
\end{cases}
$$

It prescribes the triggering of top-level schedulers by the aforementioned low-level triggers, and the triggering of the downstream program based on the results of such top-level schedulers. Then, for a certain event (rule [N-FIR]), the scheduling predicate is used to determine which programs of the program tree are to be evaluated (rule [ROUND]), leading e.g. to the executions shown in Figure 4.

5. Time-fluid Aggregate Computing

The proposed model, formalised in Section 4, has been implemented within the framework of aggregate computing [BPV15, VBD+19], which is based on the computational fields abstraction, as detailed in this section. For the implementation, we leveraged *Alchemist* [PMV13], an extensible simulator with pre-existing support for the Protelis programming language [PVB15] and the Scala-internal DSL ScaFi [CVAD20]. Specifically, we developed an extension of the simulator supporting the definition of trees of scheduling programs using the same aggregate programming language used for the actual software specification. The framework has been open-sourced, released at a public repository[2], extensively evaluated in three paradigmatic cases, covered in Section 6.

In this section, we first briefly introduce the basics of programming in the Protelis language (Section 5.1), for the sake of self-containedness. Then, we describe how the formalised model is implemented into an Alchemist-Protelis time-fluid incarnation (Section 5.2). Finally, we provide examples of time-fluid aggregate program specifications to showcase the expressive power of the proposed approach (Section 5.3).

5.1. **A short Protelis primer.** Protelis is an incarnation of the field calculus, in terms of a purely functional, higher-order, interpreted, and dynamically typed aggregate programming language interoperable with Java. This Protelis language primer is intended as a quick reference for understanding the subsequent examples—more details can be found in [PVB15].

Programs are written in *modules*, and are composed of any number of *function definitions* and of an optional *main script*, as exemplified in Listing 1.

```
1  // src/main/protelis/some/namespace.pt
2  // Module declaration
3  module some:namespace
4  // Imports
5  import java.util.HashSet;
6  // Function definitions
7  def f1(a,b,c) { /* ... */ }
8  def f2() = // ...
9  // (Optional) main script
10 f1(1,2,3) + f2()
```

Listing 1: The structure of a Protelis module.

Declaration `module some:namespace` defines 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 imported implicitly by default. 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. Definition `def f(a, b) { code }` introduces a new function named `f` with two arguments `a` and `b`. Upon invocation, the function body `code` – consisting of a series of expressions – is executed, and the function returns the value of the last evaluated expression. In case the function has a single expression, a shorter, Scala-/Kotlin-style syntax is allowed:

---

[2]`https://github.com/DanySK/Experiment-2020-LMCS-TimeFluid`

def f(a, b) = expression. *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 calls depicted in Listing 2 are in fact equivalent.

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

Listing 2: Trailing lambda convention in Protelis.

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). An example of their use is presented in Listing 3.

```
1 import java.lang.System.out
2 let currentTime = self.getCurrentTime()
3 let temperature = env.get("temperature")
4 out.println("Temperature " + temperature + " at time " + currentTime)
```

Listing 3: Example interaction with the runtime: the Java standard output is imported, the local timer is accessed, a sensor is read, and finally both these values are printed.

Then, Protelis supports the field calculus constructs as well, whose semantics is recalled in Section 4.1. 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). For instance, in Listing 4 a field of local round counts is maintained.

```
1 rep(k <- 0) { k + 1 } // Initially 0, increment and store in each round
```

Listing 4: A counter of locally executed rounds.

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 behaviour is peculiar of the field calculus semantics, where the branching construct is lifted to a distributed operator with the meaning of domain segmentation [AVD+19]. For instance, the expression in Listing 5 builds an "integer pendulum" bouncing indefinitely from 1 to -1 and vice versa. The logic is the following:

round 0: `old` is initially `0`, so the `else` branch of the `if` is selected. Evaluation of the inner `rep` block sets `count` to `0`, then evaluates `count + 1` returning `1`, which is also the result of the evaluation of the `if` expression and thus of the outer `rep`'s body and of the overall program. *Result:* `1`.

round 1: `old`'s value is `1` from the previous iteration: the "then" branch of the `if` expression is selected, and the `rep`'s body has no previous value, so `count` is set to `0`. Evaluation of the `rep`'s body yields `-1`, which is again also the result of the evaluation of the `if` expression and thus of the body of the outer `rep` and of the complete program. *Result:* `-1`.

round 2: `old` is `-1` from the previous round, the `else` branch is selected as in *round 0*, but it did not compute in the previous round, so the value of `count` is re-initialized to `0`, causing the same behaviour of the first iteration, with the program returning `1`. *Result:* `1`.

```
1  rep(old <- 0) { // begin from zero
2    if (old > 0) {
3      rep (count <- 0) { count - 1 }
4    } else {
5      rep (count <- 0) { count + 1 }
6    }
7  }
```

Listing 5: A "pendulum" swinging at each round between `1` and `-1`.

The other fundamental field calculus construct is `nbr`, which captures communication with neighbours in both directions at once. Given an expression `nbr(e)` the device evaluates expression `e`, producing a value that will be shared with its neighbours, and the whole construct evaluates to a map from neighbours to corresponding evaluations of `e` there. Such maps (also called neighbouring fields) are then typically collapsed into a single value via reduction and folding operations functions such as `foldMin`, `foldMax`, and the like. A simple but fundamental example involving `nbr` is the implementation of a *gradient* [ACDV17] depicted in Listing 6, i.e., a self-healing field of minimum distances from any device to source devices—which will also be exercised in the following sections. It works as follows: source devices (i.e., those for which field `source` is `true`) return `0` (a source is at distance zero from itself), whereas non-source devices return the minimum value of the gradient value `d` shared by neighbours augmented by the corresponding distance. This algorithm adapts to changes in the source set and in the connectivity structure as devices move or enter/quit the system, eventually stabilising to the correct values after some transient.

5.2. **A Protelis Incarnation for Time-Fluid Scheduling.** In this section, we discuss how the model formalised in Section 4 can be instantiated, and cover our implementation in Alchemist-Protelis. Indeed, the operational semantics in Figure 3 abstracts over certain

```
1  def distanceTo(source, metric) {
2    rep(d <- POSITIVE_INFINITY) {
3      let shortestPathViaNeighbours = foldMin( // must be always executed
4          POSITIVE_INFINITY, // base value
5          nbr(d) + metric()
6      )
7      if (source) { 0 } else { shortestPathViaNeighbours }
8    }
9  }
```

Listing 6: A simple self-healing gradient in Protelis. Note that `nbr(d) + metric()` must be computed outside of the `if` branches: information is sent and received when `nbr` gets executed, so even sources must run that instruction.

aspects, which must be filled in by a proper implementation. Specifically, an implementation must decide upon the following elements:

(1) what are the triggers (i.e., the values v_t assumed by sensor `trigger`) that cause a local round to be executed (i.e., an activation of rule [N-FIR]);
(2) how the scheduling tree is specified, i.e., how the application developer can define a whole program π in terms of a main script `e` together with its dependencies to other scheduling programs $\overline{\pi}$;
(3) how the predicate `sch()` is implemented and specified, to effectively control the logic of scheduling upon the structure defined as per the previous point.

Alchemist [PMV13] is an event-driven simulator where simulations are usually defined in a declarative way defining a network of nodes (there implementing the *neighbouring* relation—see Section 4.1), and, for each node, the script(s) to be executed when a certain platform event (trigger) occurs. In the proposed model, the script is a tree of program nodes, where a *program node* is a named Protelis module and an arrow $\pi \to \pi'$ between program nodes defines a dependency. So, a program node maps to π, i.e., to a $e[\overline{\pi}]$, where the corresponding Protelis module maps to the field calculus program expression `e`, and the dependencies on other program nodes capture the relationship to children $\overline{\pi}$. Therefore, following the operational semantics, it is sufficient to specify the Protelis module at the root of the tree for considering, in a round, the scheduling of the entire tree. This way, the Protelis modules with no dependencies will be considered for execution first, subsequently those whose children nodes (i.e., schedulers) have already been considered, and finally the root module.

Considering a module for execution means checking predicate $\mathtt{sch}(\theta, \overline{\theta}^o, \sigma(\mathtt{trigger}))$ for it. It is the responsibility of the platform to call such a predicate with actual parameters. This could actually be implemented in several ways: e.g., as a single function for the whole program tree, or as a separate function per individual module. In the following, we use the latter approach and propose an API where each Protelis module can specify its scheduling predicate using a special function `scheduler` whose inputs are obtained by parameter injection. For instance, in Listing 7 the module `a:b`, when considered for execution, is effectively ran if its `scheduler` function returns `true`. Annotation `@Input` is used to bind sensor values, e.g., corresponding to the `trigger` that activated the round, the previous value of module `a:b` itself (or `null` on the first execution), and the previous value of another module `c:d` (or `null` on the first execution), respectively. Notice that the logic of `scheduler`

is purely local—i.e., no aggregate constructs are admitted there. Annotation `@Changed` binds the parameter to a Boolean stating whether the sensor value has changed from the last evaluation of the `scheduler` function; this is a shorthand to avoid the creation of change-tracking scheduler modules using `rep` (cf. Section 5.3.1). Note that by statically analysing all the Protelis programs referenced from the main script, it is possible to infer the program tree from source code files with no 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 7: Per-module implementation of predicate $\texttt{sch}(\theta, \overline{\theta}^o, \sigma(\texttt{trigger}))$.

5.3. **Examples of Time-Fluid Aggregate Computing.** In this section, we provide examples adopting our scheduling approach for time-fluid aggregate computations. In the following, triggers provided by the platform (i.e., the values returned by sensor `trigger`—see Section 4.1) will be highlighted in blue. As we will see, it is often the case that the `scheduler` functions (proxies for the calculus parameter `sch()`) are essentially n-ary logical disjunctions (ORs) of their inputs (when Booleans) or simple predicates on these: this is natural, since a computation generally needs to run as soon as *at least one* of its potential causes actually happened.

5.3.1. *Timer-based scheduling.* In our first example, we show a policy recreating the classic, timer-based execution model, thus demonstrating how this approach subsumes the original execution model of field computations [AVD+19].

Consider a chain of events (and the corresponding triggers) local to one device such as the following.
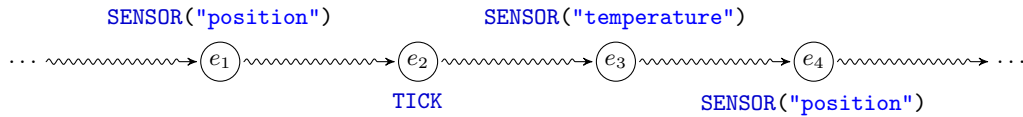
Assuming the platform exposes a sensor for obtaining the current local timestamp in seconds, it is possible to define a timer-like scheduler through a program as in Listing 8.

There is no declared scheduler, so it runs in every event. Its output is 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.

```
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)) { [current, true] } else { [old, false] }
9    }.get(1)
10
11 updated(-1, self.getCurrentTime()) { now, last -> now-last >=1 }
```

Listing 8: Timer-like scheduler.
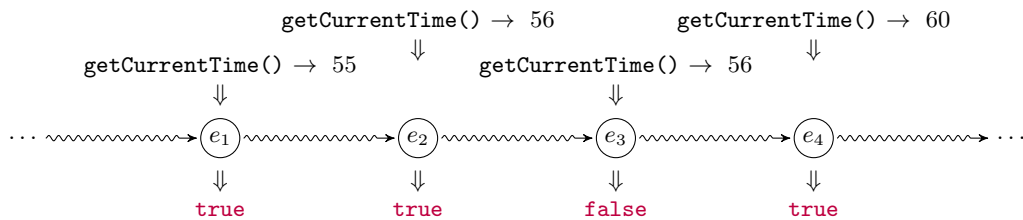
Notice that you could not actually schedule "every second" if the underlying platform events are not triggered with a second or sub-second frequency; moreover, here we only generally consider soft real-time tasks. So, for instance, a downstream program that needs to run at most once per second will not 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 9. Function `scheduler` is used by the platform to

```
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 9: A scheduling predicate logic that merely reuses the output of module `timefluid:every_second`.

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.
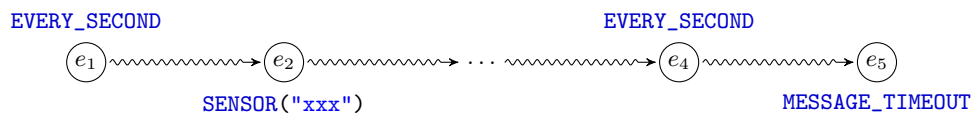
Once we have a user-defined program or platform-level timer, derived timers could be defined as in Listing 10.

```
1  module timefluid:every_minute
2
3  import platform.Triggers.EVERY_SECOND
4
5  // OPTION A) restricting the domain of a high-level scheduler
6  def scheduler(@Input("timefluid:every_second") s) = s
7  // OPTION B) restricting the domain of a low-level, platform scheduler
8  def scheduler(@Input("trigger") t) = t == EVERY_SECOND
9
10 // returns true once every 60 timer ticks
11 rep(old <- -1) { old + 1 } % 60 == 0
```

Listing 10: Example timers derived from domain restriction.

5.3.2. *Reacting to changes in the local context.* On the opposite side of the spectrum of possible policies we have 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 neighbour knowledge base (MESSAGE_TIMEOUT), for instance because the sender of the original message is no longer available—as in Listing 11.

```
1  module timefluid:some-program
2
3  import platform.Triggers.*
4
5  def scheduler(@Input("trigger") t) =
6    t == SENSOR(".*") || // Regular expression ".*" matches any sensor name
7    t == MESSAGE_RECEIVED("timefluid:some-program") ||
8    t == MESSAGE_TIMEOUT("timefluid:some-program")
9
10 // main expression
```

Listing 11: A purely reactive scheduling policy.

In general, reacting to sensors provides a way to wake a computation up, and reacting to messages enables such an activation to spread around the system as well as to keep the system computing until the output stabilises.

5.3.3. *Aggregate computations as schedulers: crowd density estimation driving alerting and crowd steering.* Now, we articulate a case in which the result of an aggregate computation is the cause for another computation to get triggered. Monitoring crowds in large events is a typical scenario for field-based coordination, as mentioned in Section 1. There, two tasks are usually performed: monitoring people density across the event area, to detect presence and growth of crowds, and steering people away from such crowds to avoid further increase in density and potential fatal events. Both tasks can be implemented in aggregate computing via appropriate fields: one for the estimated density, one for steering people. For the sake of efficiency, 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, in Listing 12 we write a Protelis program leveraging the SCR pattern [CPVN19] to partition space in regions 300 meters-wide and compute the average crowd density within them.

```
1  module timefluid:steering:density
2  import ...
3
4  def scheduler(@Input("trigger") t) =
5    t == SENSOR("people_count") ||
6    t == MESSAGE_RECEIVED("timefluid:steering:density") ||
7    t == MESSAGE_TIMEOUT("timefluid:steering:density")
8
9  let distToLeader = distanceTo(S(300)) // network partitioning
10 // sum of all the perceived people
11 let count = summarize(distToLeader, env.get("people_count"), sum)
12 // computes an upper bound to the radius
13 let radius = summarize(distToLeader, distToLeader, max)
14 count / (2 * PI * radius)//approximates crowd density as (people count/
       area)
```

Listing 12: Local crowd density estimation through the SCR pattern.

Functions `S` (network partitioning with the desired grain), `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 [FPBV17]. 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 another computation, framed in Listing 13, that determines when the steering computation should be scheduled.

```
1  module timefluid:steering:steering-scheduler
2
3  def scheduler(@Changed("timefluid:steering:density") d) = d
4
5  changed(exponentialBackOff(env.get("timefluid:steering:density"),0.1)){
6    cur, old -> abs(cur - old) > 0.5
7  }
```

Listing 13: A computation depending on the current local crowd density estimate.

There, a low pass filter `exponentialBackOff` avoids to get the program running in case of spikes (e.g. due to the density computation re-stabilisation). Note that access to the density computation is realised by accessing a sensor with the same name of the `module` containing the density evaluation program, thus reifying a causal chain between field computations. Finally, the steering program in Listing 14 would predicate on the corresponding scheduler; these modules altogether define the following application tree.

We represent Protelis modules as square nodes. Sometimes, we may also explicitly show their corresponding `scheduler` functions, as rounded rectangles left-adjacent to the program nodes. We also denote potential triggers as labels. Solid arrows denote potential causal dependencies between Protelis programs. Dashed arrows denote potential causal dependencies *on triggers*. Therefore, arrows capture the inputs to `scheduler` functions. Dotted lines, instead, are used to connect parts of the picture to informal comments.
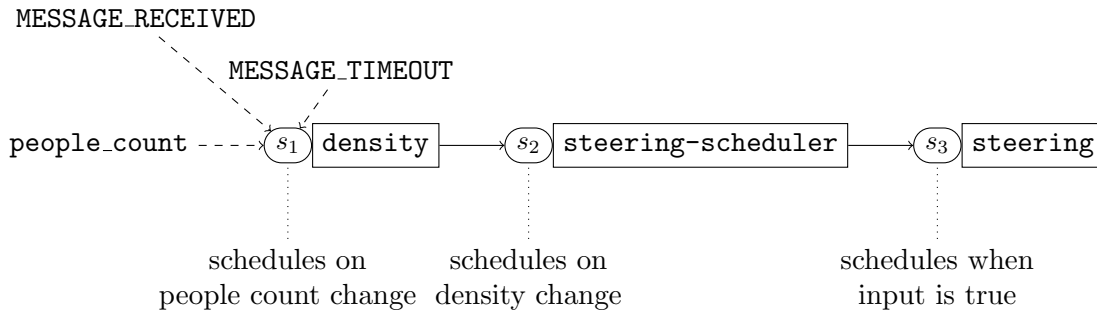
```
1 module timefluid:steering:steering
2
3 def scheduler(@Input("timefluid:steering:steering-scheduler") sched) =
     sched
4
5 // Steering logic (see, e.g., [BPV15])
```

Listing 14: The scheduling of the final crowd steering program.

5.3.4. *Self-scheduling: the case of a gradient program that self-determines whether to slow down or speed up.* As formalised in the previous section, other than due to triggers and lower-level schedulers, a program could self-schedule itself. For instance, consider the program in Listing 15 that computes a gradient (see Section 5.1) and then analyses the gradient output to determine whether it should slow down or not.

```
1  module timefluid:gradient
2  import protelis:state:time
3  import ...
4
5  def scheduler(@Input("trigger") t,
6                @Input("timefluid:every_minute") every_minute,
7                @Input("timefluid:gradient") g) =
8    // run at least once every minute
9    every_minute ||
10   // run faster when self says so
11   (t == "every_second" && g.get(1) == "fast")
12
13 let g = distanceTo(env.get("source"))
14 [g, if (isSignalStable(g, THRESHOLD)) { "slow" } else { "fast" }]
```

Listing 15: The scheduling of the final crowd steering program.

If the gradient value `g` does not change for some time `THRESHOLD` (tracked by the library function `isSignalStable`), then the computation can be slowed down: this instruction is reified as a string (`"slow"` in this case) that is returned as the second element of the 2-element-tuple output. Then, the matcher function `scheduler` leverages two platform-level timers, `every_second` and `every_minute`, to set the scheduling frequency according to the self-instruction (`g.get(1)`).
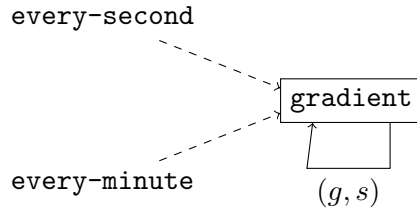
This pattern can promptly switch the computation frequency from `"fast"` to `"slow"`, since it re-evaluates its state every second, while switching from `"slow"` to `"fast"` can be only done at the frequency set for the `"slow"` progression (in this case, every minute).

6. Evaluation

In this section, we leverage our implementation to get insights on how a time-fluid version of a field-based coordination system compares to a classic time-driven version, showing that the proposed approach can in many relevant cases achieve *both* better performance (lower error) and lower global resources usage. Then, we discuss how the time-fluid version of a field-based coordination system can be finely tuned to respond to the amount of changes in a system, enabling the designer to focus on the desired performance, and having the system autonomously slow down (or stop altogether) when the "amount of change" happening is sufficiently low.

6.1. **Experimental setup.** We consider three different setups, that we will shortly refer to with the *gradient*, *moving*, and *channel* experiments. In all the three scenarios, we leverage a distance estimate between each node and a target (a *gradient* [ACDV17]—see Section 5.1). In the former two cases, the gradient is also the object of our experiments, while in the latter multiple distance estimations are exploited to build a broadcast [FPBV17], and a spatially-redundant, self-adaptive communication channel [BVPD17] between two communicating endpoints. Distance estimation is central in our evaluation as it is one of the most common building blocks over which other, more elaborate forms of coordination get built [FSM+13, VAB+18, PCVN21]. Computing distance from a source without a central coordinator in arbitrary networks is a representative application of aggregate computing, for which several implementations exist [VAB+18, ACDV17]. In this work, since the goal is to explore the behaviour of the time-fluid program version rather than the efficiency of the distance estimation algorithm itself, we use a variant of the adaptive Bellman-Ford [DB16] algorithm exploiting the recent `share` primitive [ABD+20], even though it is known not to be the most efficient implementation for the task at hand [ACDV17].

The baseline for assessing our proposal is a *classic* implementation in the framework of aggregate computing: time-driven, unsynchronised, and fair scheduling of rounds set at 1Hz. We compare the classic approach with a time fluid version whose structure will be presented in the experiment-specific sections that follow. The time-fluid version is purely *reactive*, and its performance depends on two parameters:

ϵ (*tolerance*), measured in meters, is the distance required for a node to consider itself to have moved from its previous position;

λ (*mean arrival frequency*), measured in Hz, represents the reciprocal of the mean time required by a network message to be prepared, sent to a neighbour, and decoded by such neighbour.

Name	Description	Values
λ^{-1}	mean network latency	$\{0.01, 0.03, 0.1, 0.3, 1\}\, s$
ϵ	distance tolerance	$\{0, 0.01, 0.03, 0.1, 0.3, 1, 3\}\, m$
$\|\vec{v}\|$	mobile nodes' speed	$\{0, 0.01, 0.03, 0.1, 0.3, 1, 3\}\, m/s$
algorithm	implementation type	classic, time-fluid
scenario	position and program of devices	gradient, moving, channel

Table 1: Independent variables for the evaluation.

Name	Description	Notes
δ	error of a single device w.r.t an oracle	
ρ	rounds executed by a single device	
$\mathbb{E}(\delta)$	mean error w.r.t. an oracle	proxy for the global error
$\mathbb{E}(\rho)$	mean round count	proxy for resource cost
$\sigma(\rho)$	standard deviation of round count	proxy measure of asymmetry
$\int_0^t \mathbb{E}(\delta)\, dt \cdot \mathbb{E}(\rho)$	cumulative error times round count	proxy for efficiency

Table 2: Metrics for the evaluation.

Higher values of ϵ lead to less frequent evaluation by the time fluid version of the system, at the price of a greater expected error. Values of λ mostly depend on the performance of devices composing the system and of the communication network. We suppose devices to have similar performance, and we model the network latencies to be Weibull-distributed as suggested by the literature on traffic modelling [APMW13]. We set the Weibull distribution shape $\alpha = 1$ and scale $\beta = \lambda^{-1}$; this way, the expected value for samples of such distribution is $\mathbb{E} = \beta = \lambda^{-1}$. In other words, with this network model, we have a direct link between the value of λ and the mean time required to deliver a message, which will be λ^{-1} seconds. In the remainder of this work, we will refer to λ^{-1} to indicate the network performance, which can be interpreted as the mean time required for a message to be delivered and processed by the receiver.

Devices are deployed differently in each example, and details of each deployment are presented in the subsection detailing each experiment. In all experiments, though, mobile nodes share the same constant speed $\|\vec{v}\|$, which is among the controlled variables. All controlled (independent) variables are summarised in Table 1. For each combination in the Cartesian product of the variables' values, we perform multiple simulation runs (50 for the channel scenario, and 100 for the gradient and moving scenarios), changing the simulation seed.

Our metrics are summarised in Table 2. We measure the error δ of any device with respect to an oracle. In the case of the gradient, the error is the distance between the correct value as provided by an oracle and the value as computed by the executing algorithm; in the case of the channel, $\delta = 0$ if the algorithm being executed and the oracle agree on whether a device is part of the channel, and $\delta = 1$ otherwise. We count, for each device, how many rounds have been executed (ρ). For the classic version of the algorithm, every round contributes to the count with a unitary increment of the value. For the time-fluid versions, a round increments ρ only if the target algorithm is effectively scheduled for execution and
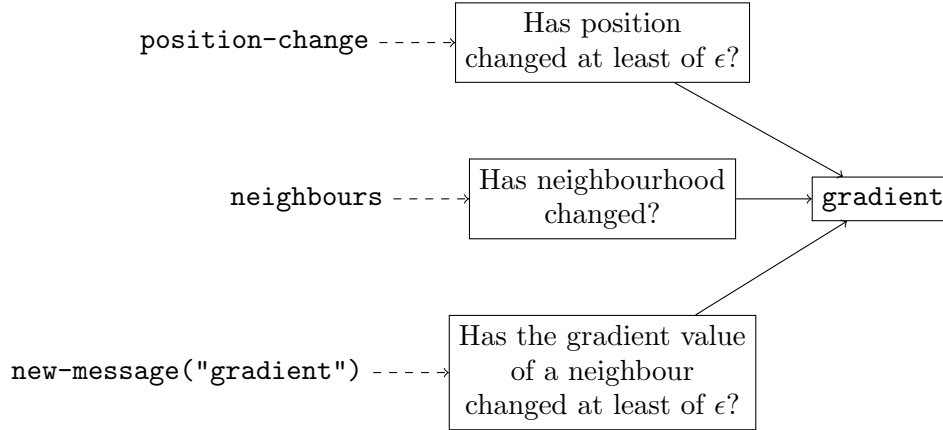
Figure 5: Application tree for the *gradient* experiment.

performed, the evaluation of upstream schedulers is not considered. Starting from these two raw metrics, we derive the following ones:

- $\mathbb{E}(\delta)$: mean error across all devices, as a proxy for the global network error;
- $\mathbb{E}(\rho)$: mean count of rounds across all devices, as a proxy for global resource usage (more rounds imply more network communications and power usage);
- $\sigma(\rho)$: standard deviation of rounds across all devices, as a proxy for asymmetry in execution among devices, with higher values indicating that some devices run more frequently than others—higher standard deviations also witness a more noticeable difference between the time-fluid and the classic version (whose $\sigma(\rho)$ is the one closer to zero because most devices run at a similar frequency);
- $\int_0^t \mathbb{E}(\delta)\, dt \cdot \mathbb{E}(\rho)$: we multiply the mean cumulative count of rounds (resource usage) by the cumulative mean error of the network, as a proxy for efficiency (how much the error is reduced with respect to the additional rounds executed).

The simulation has been implemented in Alchemist [PMV13], writing the aggregate programs in Protelis [PVB15]. Data has been processed with Xarray [HH17], and charts have been produced via matplotlib [Hun07]. All the experiments and the production of the charts presented in this work has been automated, open-sourced, documented, and made available for easy reproduction[3] [Pia21]. Besides the source code, we provide detailed instructions, as well as all the means to execute the experiment in a containerised environment (including a pre-configured Kubernetes pod, in case the reader has access to a cluster that can speed up the computation).

6.1.1. *Gradient experiment.* The goal of this setup is to measure how the time-fluid versions of the field-based coordination compare to a classic implementation when a part of the network is stable, and another is instead constantly changing due to the movement of a small subset of its components. The time-fluid program is graphically shown in Figure 5.

The simulation is executed in a 21x21 irregular grid of devices displaced in a square arena, each located randomly within a disc centred 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

---

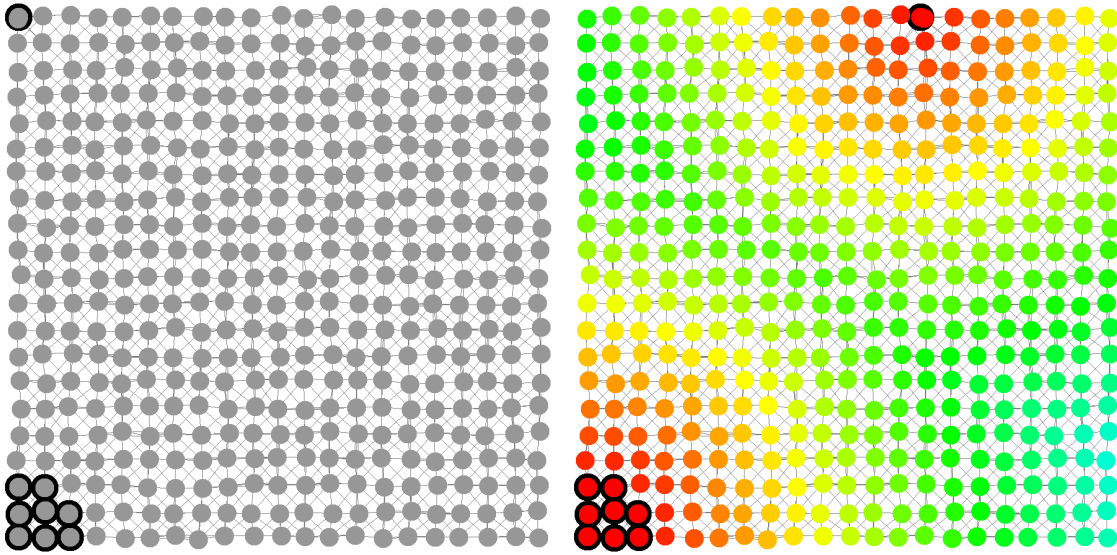[3]https://github.com/DanySK/Experiment-2020-LMCS-TimeFluid

Figure 6: Snapshots of the simulation running in the gradient scenario. Nodes are initially
displaced in an irregular grid, with and additional device on the top left corner.
This device and those at the bottom left corner, depicted with a black border, are
sources. When running, the additional node on top oscillates between the left and
right edges of the arena, and the gradient must adjust accordingly. We use colors
to denote the gradient values: warmer colors denote lower gradient values (i.e.,
smaller distances, or greater proximity to a source), whereas cooler colors denote
higher gradient values (i.e., bigger distances, or greater remoteness from sources).

velocity $\vec{v}$ whose speed $\|\vec{v}\|$ is constant, and whose direction reverses when it reaches the
leftmost and rightmost limits of the arena. The mobile device and the leftmost devices at
bottom are "sources", and the goal for each device is to estimate the distance to the closest
source. Snapshots of the deployment and its ongoing simulation are depicted in Figure 6.

6.1.2. *Moving experiment.* This experiment's goal is to analyze how time-fluid and classic
coordination compare when the system is entirely dynamic. In particular, we expect this
case to show that the time-fluid version can autonomously tune the frequency at which
devices compute depending on what error is considered to be "acceptable" (in our case, the
value of ϵ). Devices are initially displaced in a 21x21 irregular grid, enclosed in a square
arena. An additional device, marked as source, is located initially at the centre of the arena.
Devices are free to move within such arena with velocity $\vec{v}$; speed $\|\vec{v}\|$ is constant within
every experiment (see Table 2), while direction is determined with a uniform probability and
changed when nodes hit the boundary of the arena or after they walked a length determined
by samples of a Pareto distribution with scale $k = 1/2$ and shape $\alpha = 1$—thus creating
a variant of Lévy Walks [ZDK15] for a physically constrained environment. This kind
of walks have selected as they reasonably approximate the walking behaviour of human
beings [RSH+11]. Snapshots of the deployment and its ongoing simulation are depicted in
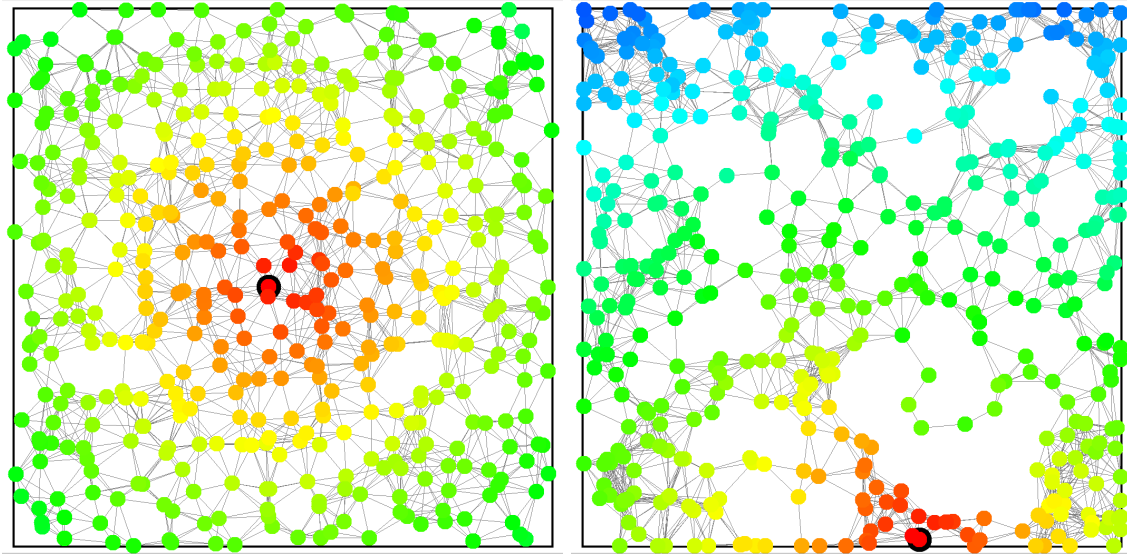Figure 7.

Figure 7: Snapshots of the simulation running in the moving scenario. Nodes are randomly displaced in a square arena. A device at the centre of the network is a source, depicted with a black border. When running, all nodes move (including the source), and the gradient must adjust accordingly.

6.1.3. *Channel experiment.* This experiment is meant to investigate if and how the effects of time-fluid computations "stack" when a more elaborate setup is in place. To this end, in this experiment we do not just run a gradient, but we build a redundant communication channel between two devices. This is done in a purely distributed setup by:

(1) propagating a gradient for each of the devices willing to establish the channel, thus computing for each device i two distance values d_i^0 and d_i^1, respectively measuring distance from one communication end (labelled device 0 and 1);

(2) propagating along another gradient (i.e., broadcasting) the distance that one of the two communication ends perceives from the other, e.g., d_0^1 (or, equivalently in Euclidean folds, d_1^0); and

(3) considering as part of the channel all those nodes for which $d_i^0 + d_i^1 < d_0^1 + w$, with w being the *width* of the channel.

In Protelis this can be coded as in Listing 16.

```
1 def distanceBetween(a,b) = broadcast(a, distanceTo(b))
2 def channel(a,b,w) = distanceTo(a)+distanceTo(b) <= distanceBetween(a,b)+w
```

Listing 16: A simple channel pattern in Protelis.

This algorithm lends itself to a decomposition in its building blocks (two gradients and a gradient plus a broadcast), and thus to a time-fluid modelling that is responsive to changes in the perceived distances, as shown in Figure 8, where the `distanceBetween` block is implemented with the homonym function of the Protelis-lang library [FPBV17].
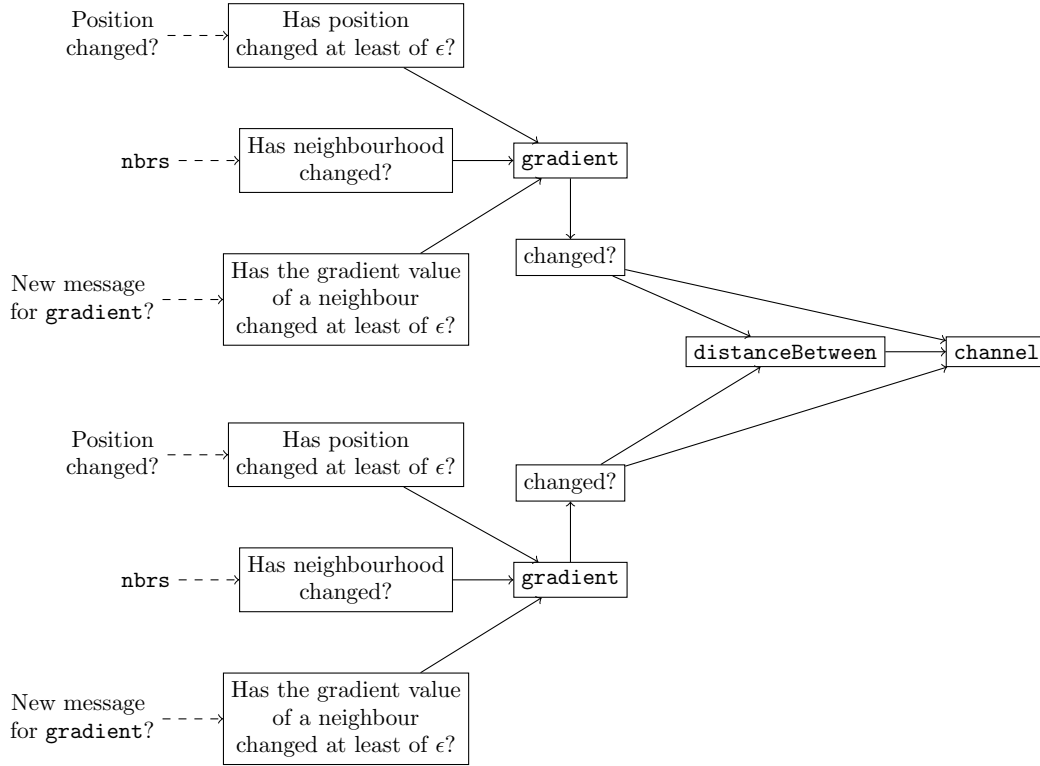
Figure 8: Application tree for the *channel* experiment.

In order to create a challenging environment, we displace devices in a 42x21 irregular grid, and we enclose the left-hand half of them within a square arena, letting communication cross such a boundary. Devices outside the square arena remain still, while devices within the arena move with the same Lévy Walks described for the moving experiment in Section 6.1.2. Snapshots of the deployment and its ongoing simulation are depicted in Figure 9.

6.2. **Classic and time-fluid coordination compared.** The first expected benefit of a time-fluid version of a field-based computation is its ability to slow down and even stop altogether if there are no changes in the environment. This base functionality emerges clearly from Figure 10, where we depict the mean round count $\mathbb{E}(\rho)$ (our proxy metric for resource usage), and we set $\|\vec{v}\|=0$, making the nodes stand still. While in the classic implementation $\mathbb{E}(\rho)$ grows linearly with time, hence consuming resources just to "maintain" the coordination fields, the time-fluid versions stop computing altogether after a stabilisation time. Interestingly, the time-fluid versions that execute on a faster network $(\lambda^{-1} = 0.1s)$ have a quicker transient, and initially show a higher resource usage compared to the classic version, which is bounded by its fixed working frequency. In these cases, the time fluid version both converges more quickly and saves resources in the long run.

The error analysis results are depicted in Figure 11 for the gradient scenario, in Figure 12 for the moving scenario, and in Figure 13 for the channel scenario. The periodicity of the gradient experiment clearly appears in the experimental data, in particular when compared to the more chaotic behaviour of the other two versions, moving a much larger part of the
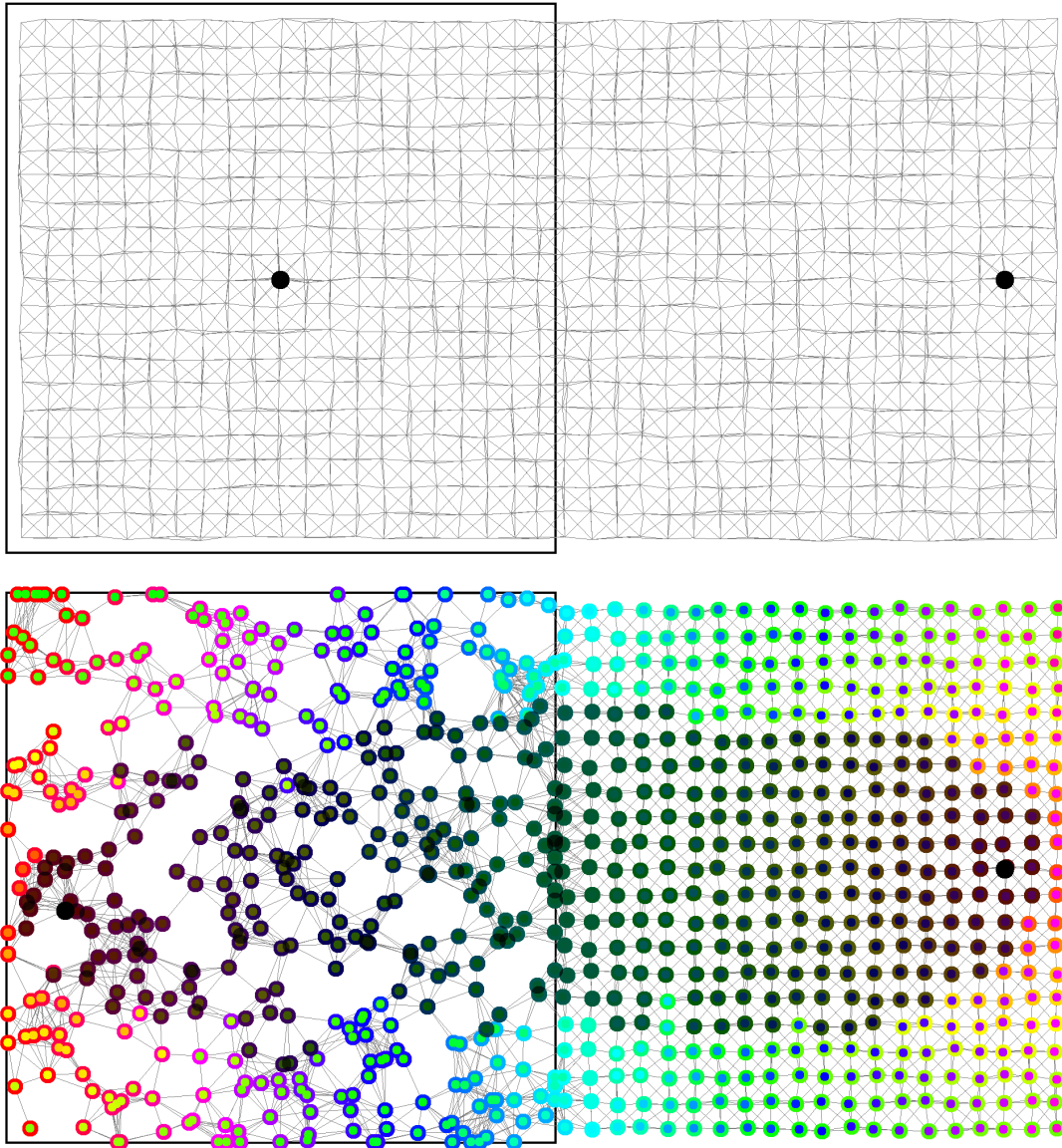
Figure 9: Snapshots of the simulation running in the channel scenario. Nodes are randomly displaced in a rectangular irregular grid. The left half of such grid is enclosed in a square arena, where devices are free to roam. The two devices at the centre of each half of the network are the communication ends of the channel (marked in black). When running, nodes within the square arena move (including the source), and the system must keep the redundant channel up. The dark (or shadowed) nodes in the bottom snapshot are the devices that make up (or "belong to") the channel, i.e., for which the `channel` function returns `true`.

devices with Lévy walks. The time-fluid implementation with low tolerance consistently outperforms all the other proposals under test. The other combinations (fast network, low
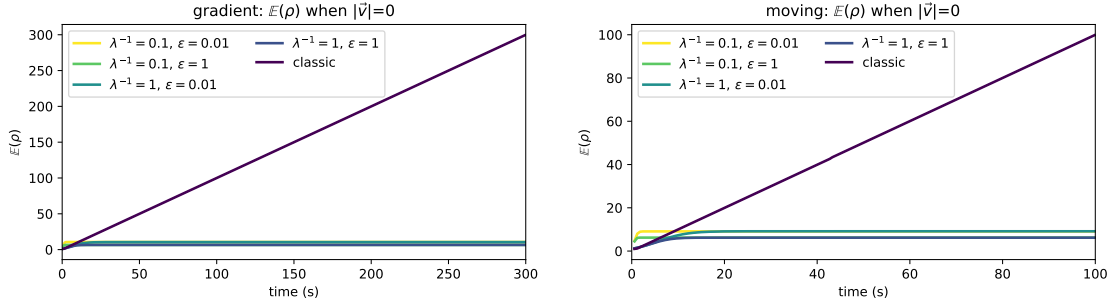
Figure 10: Compared cost (estimated using the count of executed rounds ρ as proxy metric) in case of still devices, for the gradient scenario (left) and the moving scenario (right). Data shows the ability of the time-fluid version of the system to pause its computation when no relevant changes happen. Faster devices on faster network with lower tolerance to error initially run more rounds than the classic version. However, once stabilisation is reached, they stop, while the classic version keeps computing in order to sustain the existence of the computational field.

tolerance; and low tolerance with fast and slow network) are usually competitive with the classic implementation. Data shows several interesting behaviours:

- In the gradient case, low speeds are tolerated much less than in the classic version. Since both versions with high tolerance perform similarly despite a much different network performance, we can say that this behaviour is due to the tolerance being high enough to account for the amount of error, which in fact remains pretty much stable across different values of $\|\vec{v}\|$, while all other versions accumulate progressively more error at higher speeds.
- In the moving case, the continuous disruption induced by the movement of the whole network induces a lower impact of the tolerance: the network is continuously prompted to recalculate, and hence the network performance assumes greater relevance (as they ultimately dictate how quickly new information can propagate). As a consequence, both the time-fluid versions (low and high tolerance) running on a fast network outperform the classic version, while the ones limited by the network performance perform on average worse.

Pure performance, however, is only a part of the story. Figure 14, Figure 15, and Figure 16 show our efficiency proxy metric results for, respectively, the gradient, moving, and channel scenarios. This simple metric is obtained by multiplicating the cumulative error the system produced for the duration of its operation ($\int_0^t \mathbb{E}(\delta)\, dt \cdot \mathbb{E}(\rho)$) by the total number of rounds performed up to that time ($\mathbb{E}(\rho)$). This metric provides an estimate of the operating cost relative to the precision required by the system. To better understand how much of the overall cost metric is due to the executed round, we also present separated results for the total number of rounds perfomed ($\mathbb{E}(\rho)$) in Figure 17, Figure 18, and Figure 19 for the gradient, moving, and channel scenarios respectively. Data shows that the time-fluid version of the field coordination system is generally more efficient than the time-driven counterpart. It can be more expensive to maintain, but this price delivers lower error; and on the other hand, it can perform comparably, but with a much lower cost. This is especially true in cases in which there are localised changes whose propagation is limited. In fact, while
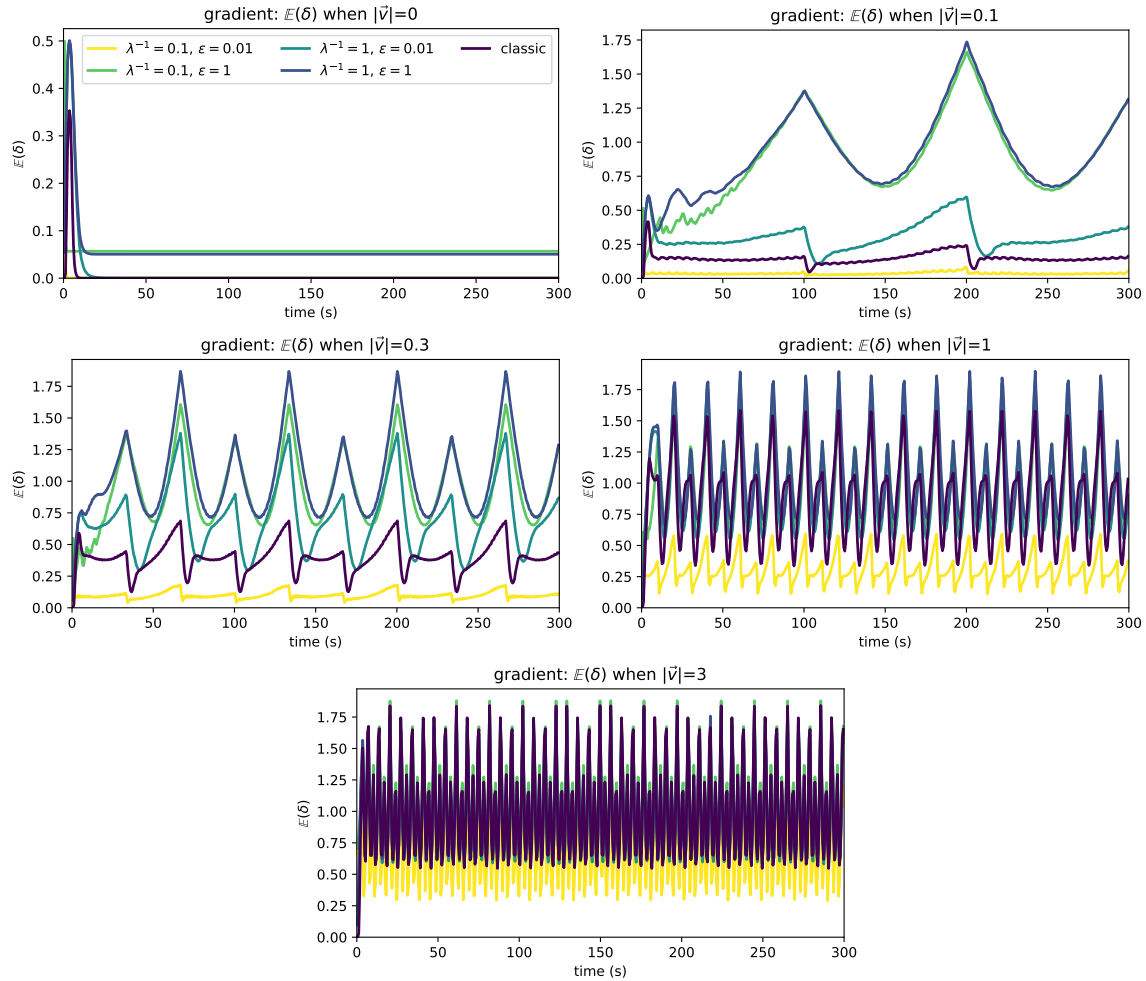
Figure 11: Compared error in the *gradient* scenario. The classic version is the darkest (purple in color) line, and is compared with four time-fluid variants, with tolerance of 0.01m and 1m, executed with delays of 0.01s and 1s. The time fluid version with low tolerance executed on a reasonably fast network consistently outperforms the classic version of the algorithm. The time-fluid version is sensible to slow networking: with a mean network delivery time of 1 ($\lambda^{-1} = 1$), the time-fluid version accumulates more error on average than the classic version, at least for low values of $\|\vec{v}\|$. Interestingly, when the tolerance to error is high, the system acquires a certain robustness to the network performance as well: when $\epsilon = 1$, the performance between a fast network and a network one order of magnitude slower is barely noticeable. Higher values of $\|\vec{v}\|$ progressively flatten the differences between the algorithms, although the time-fluid version with low tolerance consistently outperforms all other algorithms.

there are circumstances in the moving and channel experiments when the classic version is more efficient than some configurations of the time-fluid ones, in the case of the gradient it is consistently more efficient no matter how the conditions are changed. The reason for
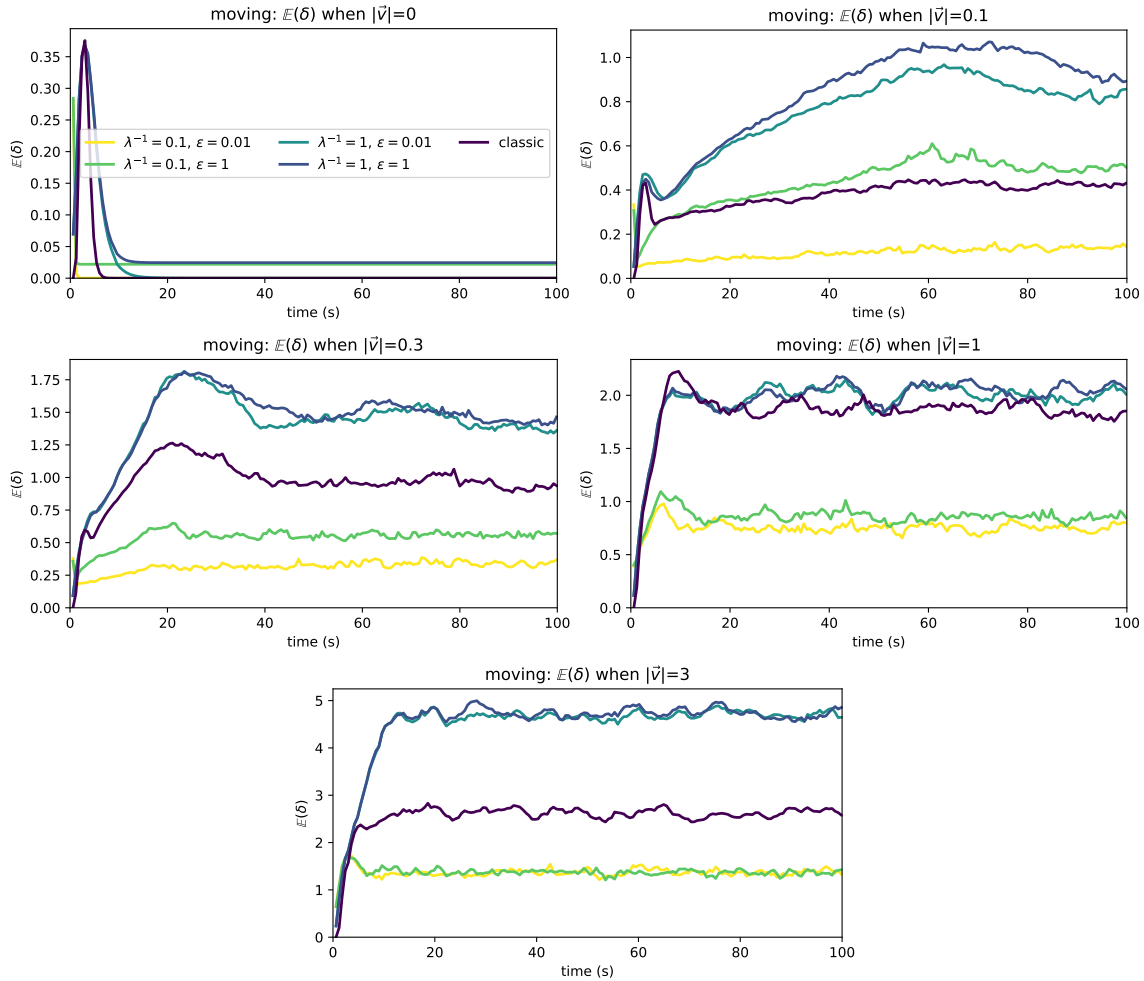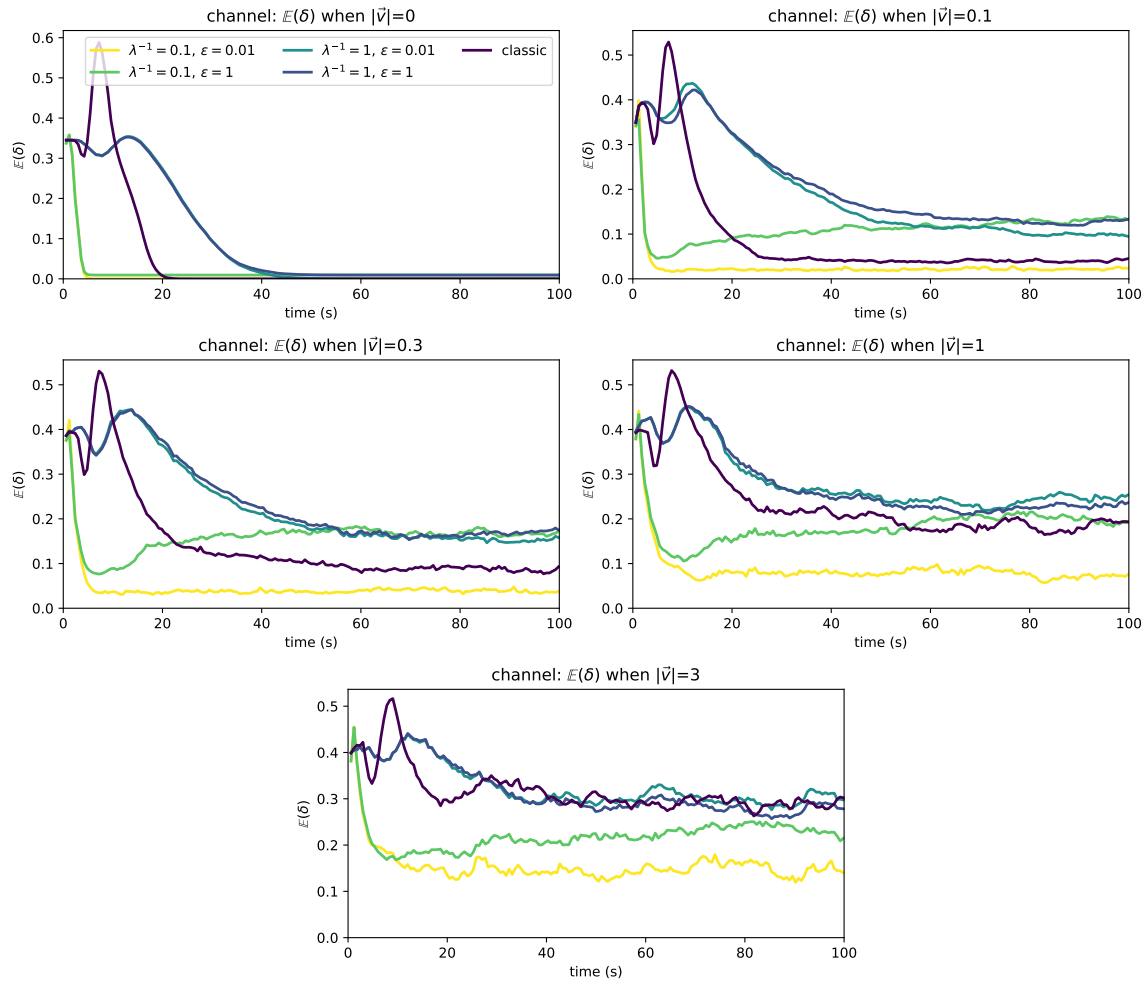
Figure 12: Compared error in the *moving* scenario. The classic version is the darkest (purple in color) line, and is compared with four time-fluid variants, with tolerance of 0.01m and 1m, executed with delays of 0.01s and 1s. Constant movement by all nodes makes the tolerance very often violated (and hence, time-fluid computation scheduled) for most of the time, thus producing similar results for both versions running on a fast network ($\lambda^{-1} = 0.1$).

this behaviour are to be found in the ability of the time-fluid version to shut down the computation where there is no need of updates.

Figure 13: Compared error in the *channel* scenario. The classic version is the darkest (purple in color) line, and is compared with four time-fluid variants, with tolerance of 0.01m and 1m, executed with delays of 0.01s and 1s. The time-fluid versions running on a fast network outperform all the other versions, while those running on a slower network take more time than the classic one to stabilise, but once a reasonable stability is reached, performance is comparable.
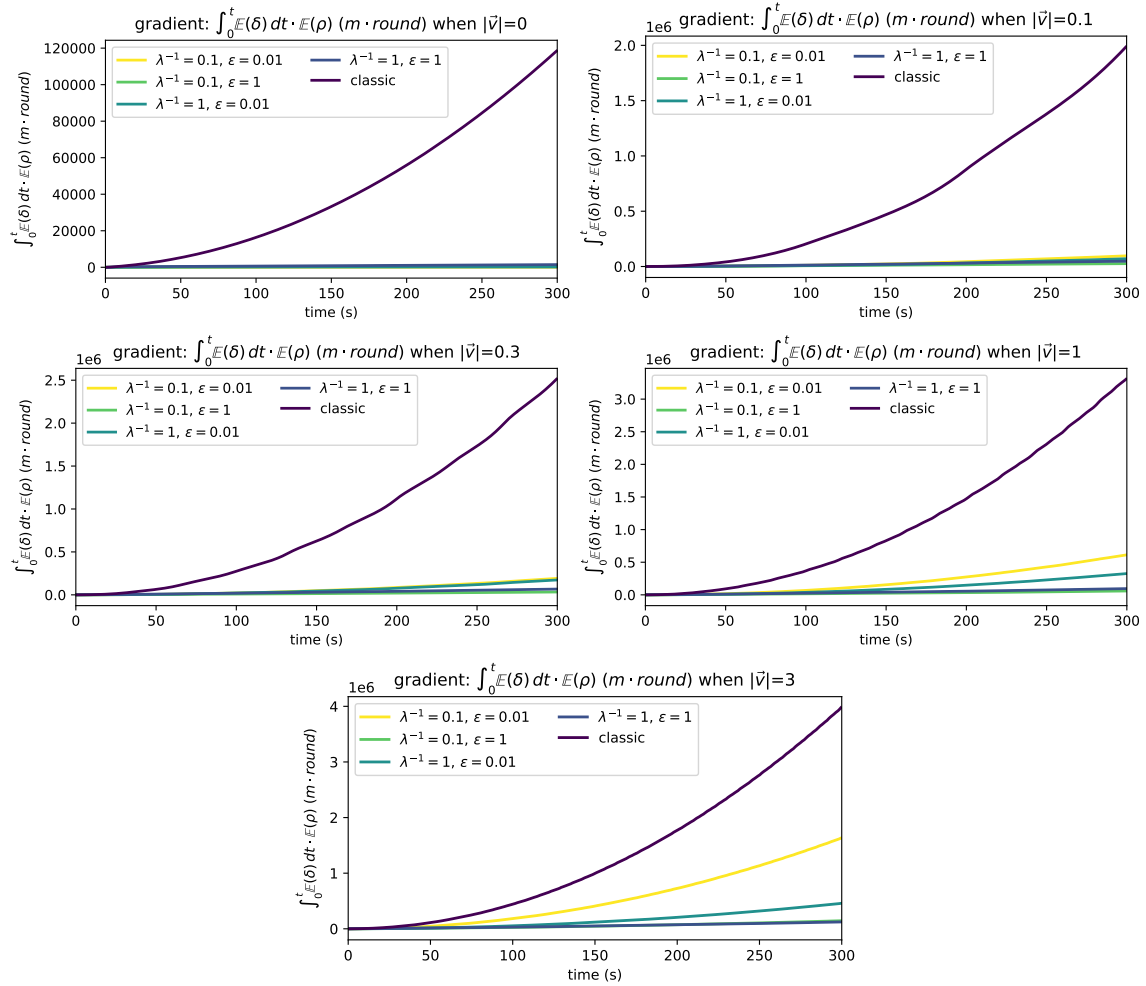
Figure 14: Compared product between cumulative error and cumulative round count in the gradient scenario, as a proxy metric for efficiency. Lower values indicate better efficiency. The classic version is the darkest (purple in color) line, and is compared with four time-fluid variants, with tolerance of 0.01m and 1m, executed with delays of 0.01s and 1s. Time-fluid versions strictly perform better than the classic version, due to their ability to cope with changes happening in a sub-portion of the whole network.
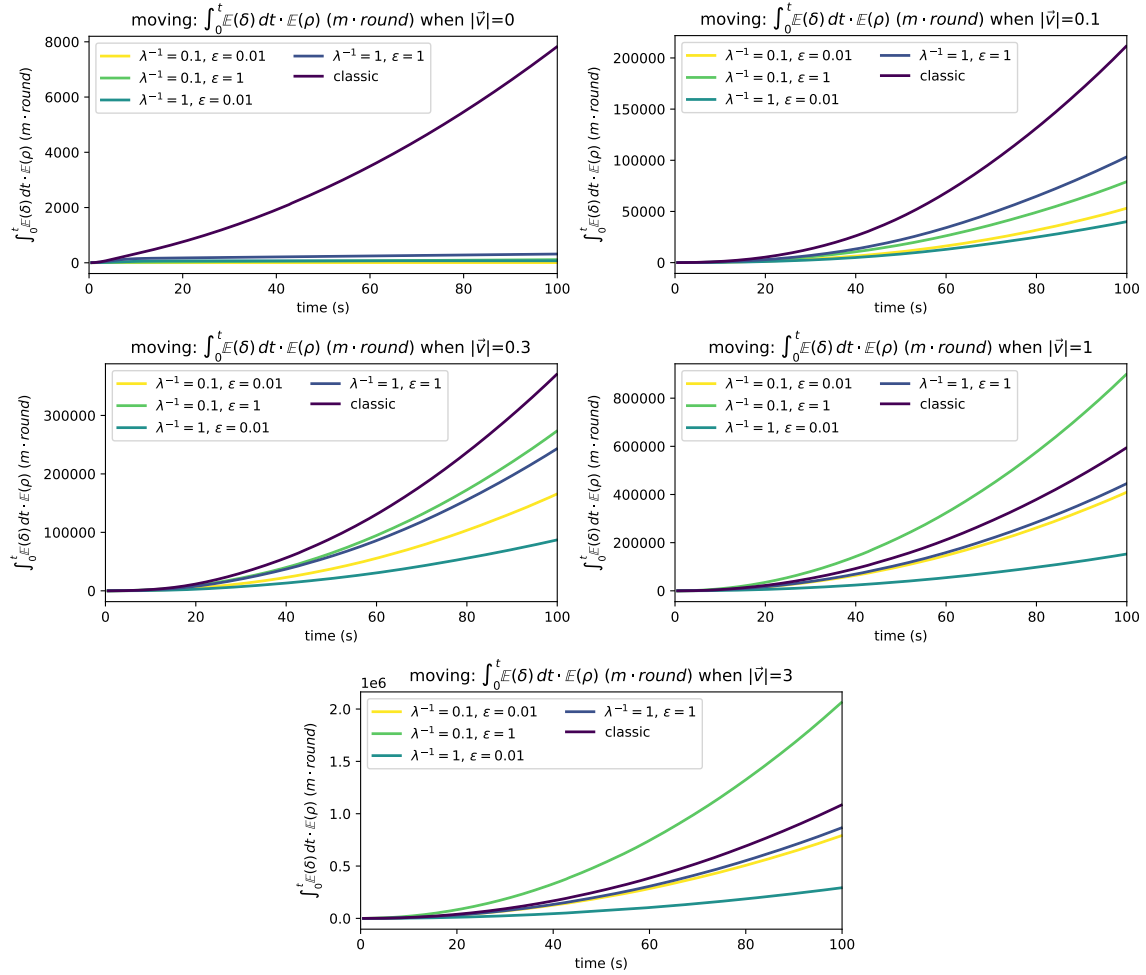
Figure 15: Compared error in the channel scenario. The classic version is the darkest (purple in color) line, and is compared with four time-fluid variants, with tolerance of 0.01m and 1m, executed with delays of 0.01s and 1s. Time-fluid versions of all fashions outperform the baseline for $\|\vec{v}\| \leq 0.3 m/s$. At higher speeds, unfortunate combination of tolerance and network latency may produce values whose error is high if compared to the amount of computational resources required to perform the computation. In our experiments, this happens for instance when $\lambda^{-1} = 0.1s$ and $\epsilon = 1m$: frequent network messages mean frequent re-adjustments, but the high tolerance does not bring the process to the point that error gets low enough to justify the expense. Even though the version with very low tolerance $\epsilon = 0.1m$ runs more rounds (hence, costs more) it reduces the error so much that the final cost metric is lower overall.
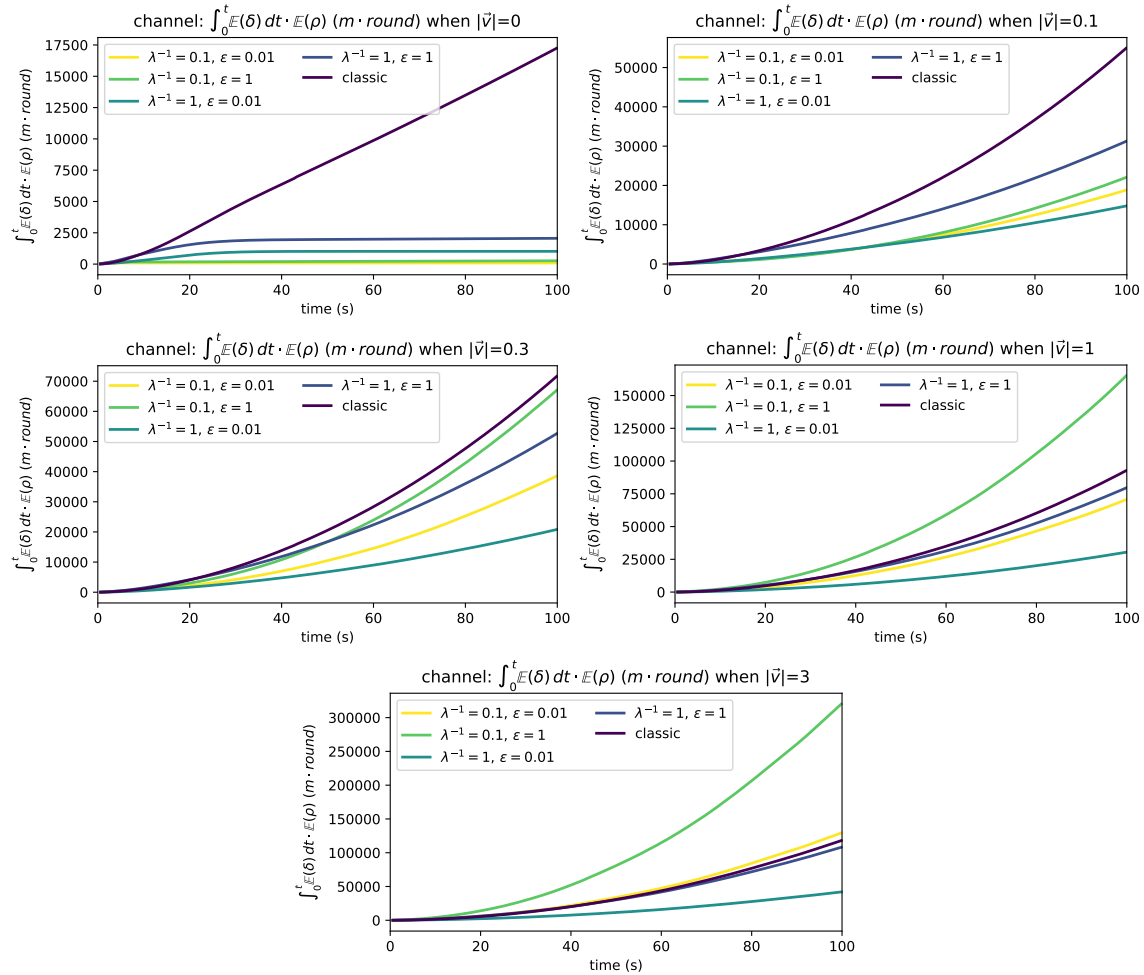
Figure 16: Compared error in the channel scenario. The classic version is the darkest (purple in color) line, and is compared with four time-fluid variants, with tolerance of 0.01m and 1m, executed with delays of 0.01s and 1s. Time-fluid versions of all fashions outperform the baseline for $\|\vec{v}\| \leq 0.3 m/s$. At higher speeds, unfortunate combination of tolerance and network latency may produce values whose error is high if compared to the amount of computational resources required to perform the computation. In our experiments, this happens in particular when $\lambda^{-1} = 1s$ and $\epsilon = 1m$. Other versions achieve a ration between performance and cost which is similar to the one of the classic version. However, we note that there are some circumstances under which the time-fluid version of the channel achieves a much better performance / cost ratio even at high speeds ($\lambda^{-1} = 1s$ and $\epsilon = 0.01m$).
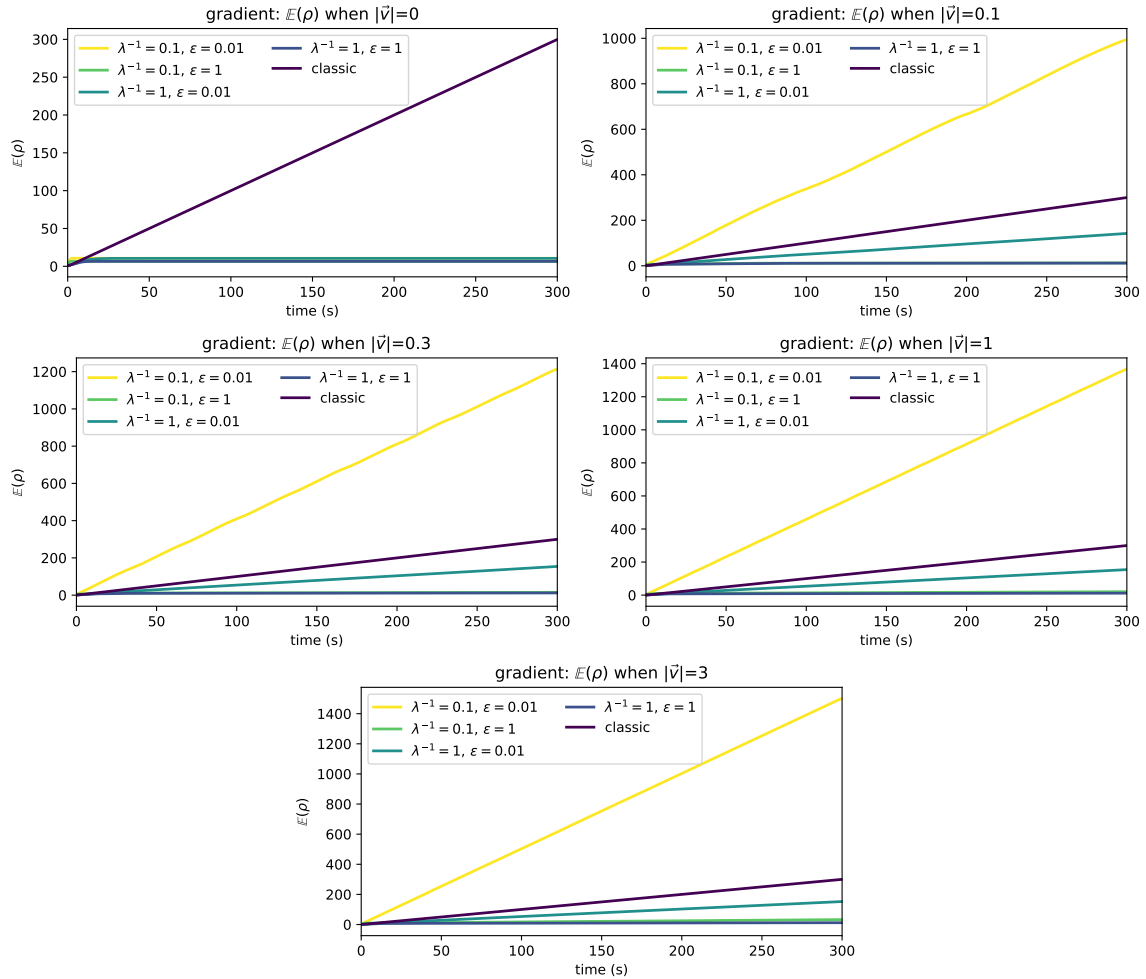
Figure 17: Overall round count in the gradient scenario. Both network performance and tolerance influence consistently the overall count of rounds performed by the time-fluid versions. When $\|\vec{v}\| = 0$, all time-fluid versions stop very quickly compared to the classic version. Even low movement speeds impact noticeably on the count of rounds executed by the time-fluid version with low tolerance and a low-latency network ($\lambda^{-1} = 0.01s$, $\epsilon = 0.01m$), while both a larger tolerance and higher network latencies are very effective at trimming down the overall count of executed rounds. Note that the scales on the y-axes differ.
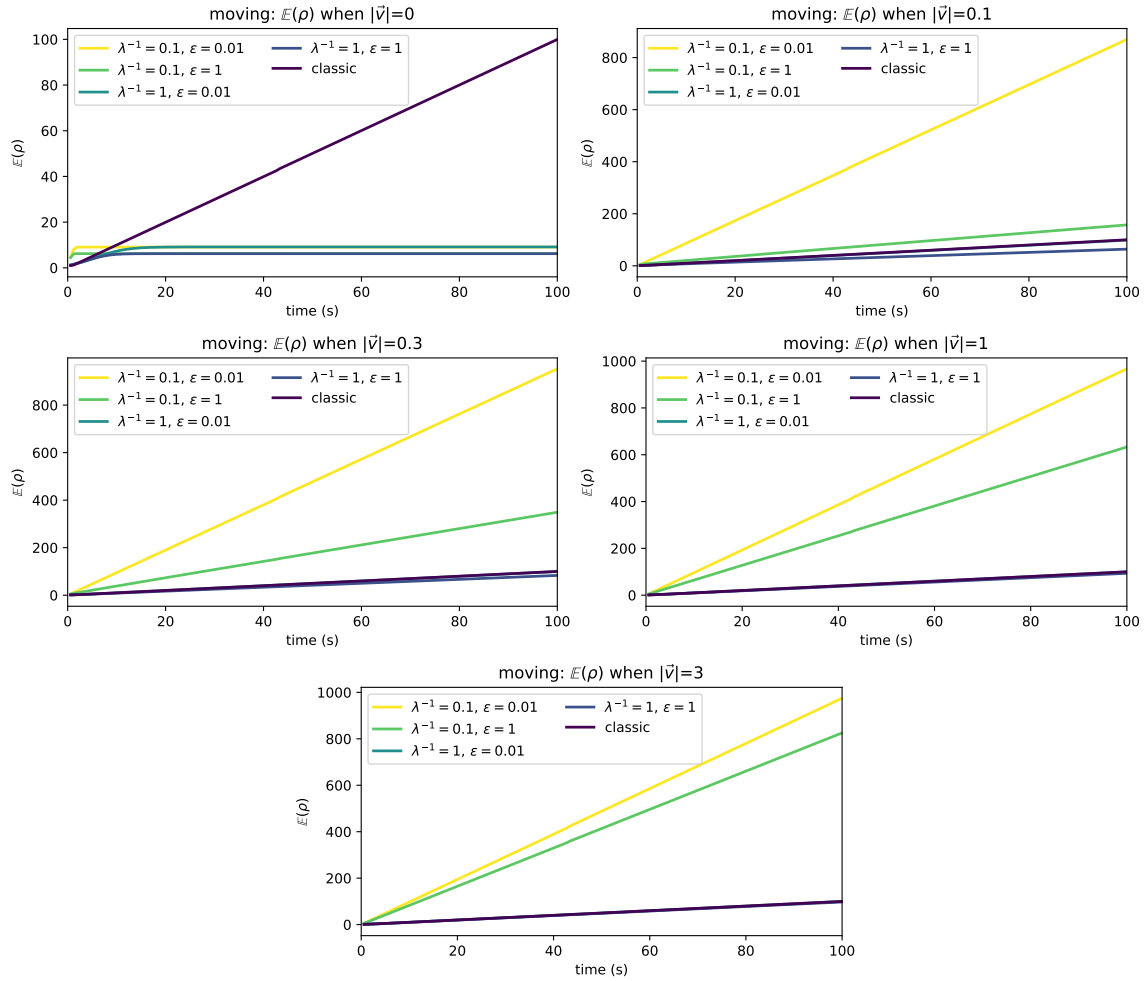
Figure 18: Overall round count in the moving scenario. Both network performance and tolerance influence consistently the overall count of rounds performed by the time-fluid versions. When $\|\vec{v}\| = 0$, all time-fluid versions stop after a transient whose length is shorter for low-latency networks (lower values of λ^{-1}) and whose overall round count depends on the accepted error (tolerance ϵ). In this scenario, network latency is the factor that mostly influences the overall round count for time-fluid versions, as global widespread movement with higher speed quickly makes devices get past the fixed tolerance threshold. Note that the scales on the y-axes differ.
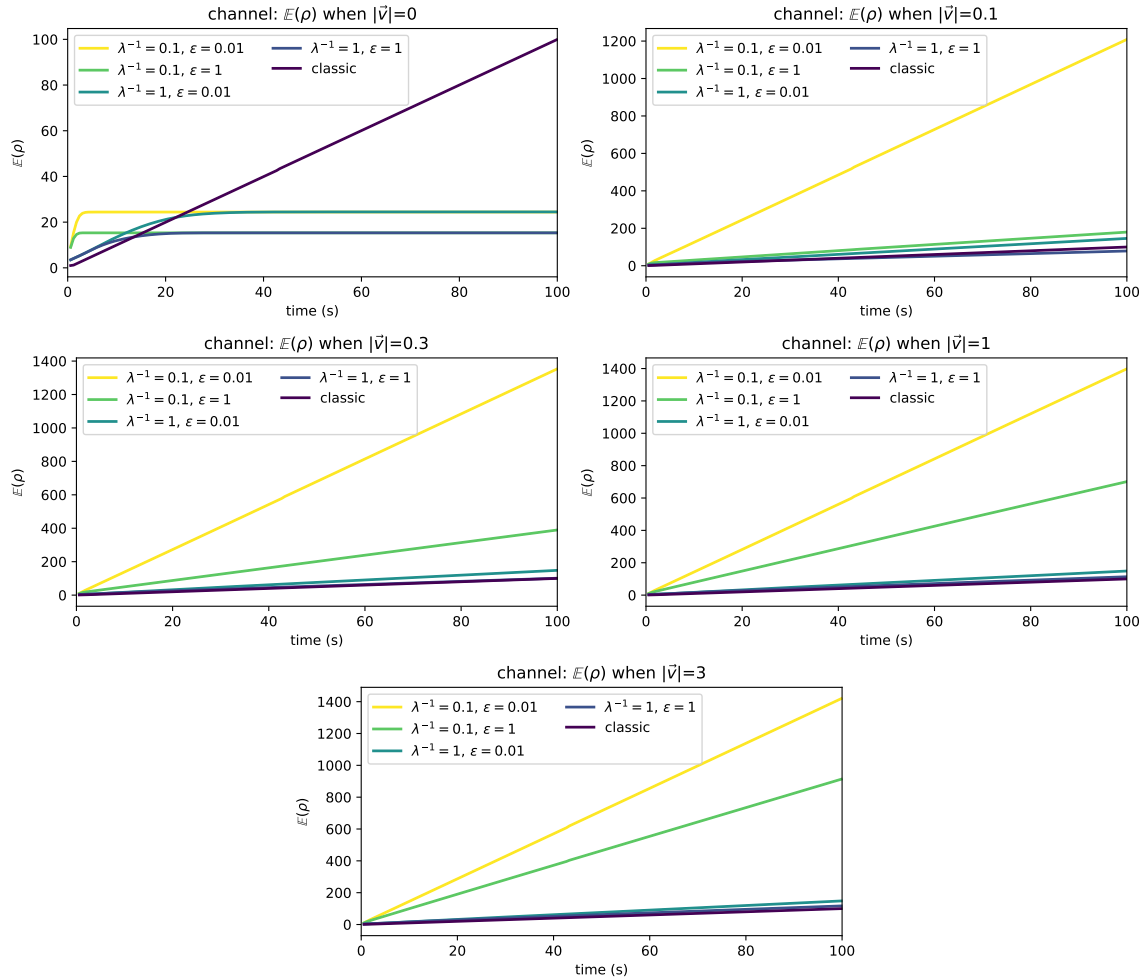
Figure 19: Overall round count in the channel scenario. Both network performance and tolerance influence consistently the overall count of rounds performed by the time-fluid versions. When $\|\vec{v}\| = 0$, all time-fluid versions stop after a transient whose length is shorter for low-latency networks (lower values of λ^{-1}) and whose overall round count depends on the accepted error (tolerance ϵ). Note that the scales on the y-axes differ.
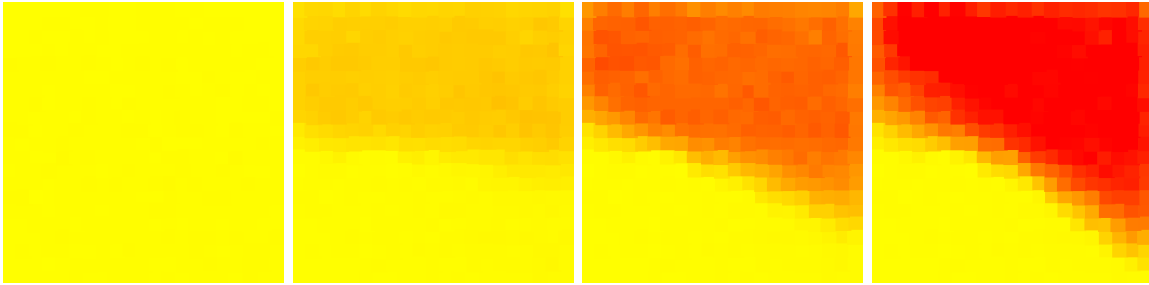
Figure 20: Heat-map representation of the number of executed rounds along time. Each device is depicted as a point located on its actual coordinates. Time progresses from left to right. Devices start (left) with no round executed (yellow) and, with the simulation progression (left to right), execute 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.

This is especially clear by looking at Figure 20, which depicts with increasingly red colour the devices in the grid based on the count of rounds they performed: only those devices that come to get closer to the moving source than to the static sources become red, while others, once stabilised, stop their computation.

6.3. **Impact of tolerance to change and network latency.** Data showing the overall impact on the time-fluid versions of the algorithms for the scenarios under test are depicted in Figure 21. Network latency and tolerance interplay in an interesting way: the latter can be used to limit the "reactivity" of a system in which a change may trigger a long chain of reactions. This is especially clear by looking at the error for the gradient scenario: at some point ($\epsilon \geq 1$) the error stabilises for every value of λ: the tolerance is dominating the error. The designer can thus design the scheduling of the field-based coordination processes by considering what is the maximum level of reactivity that should be supported. This level can be tuned using a form of tolerance similar to the one used in this work, in order to obtain a system that follows the correct value as most as fast as necessary in order to keep the error under a threshold defined by the system architect. A similar consideration can be made by looking at the other side of the coin: the ability of a system to promptly react to changes is only limited by the physical slowdowns related to the network communication times (or, from a more philosophically sound perspective, by the time required to events to be perceived). The practical consequences of this fact are that a time-fluid system is able to autonomously adapt to working into a different infrastructure: if there is no form of tolerance to error included by the designer, then the time-fluid system will potentially try to converge as quickly as possible. Moreover, there is a large potential for time-fluid coordination systems to be successful in saving global resources: since the system is able to stop computing where and when unnecessary, it can lead to resource savings. On the other hand, however, time-fluidity introduces *asymmetry* in computation frequency: some devices, located in "hot spots" of the computation, will need more resources than others, and this might potentially complicate the prediction of the resource usage by each device.
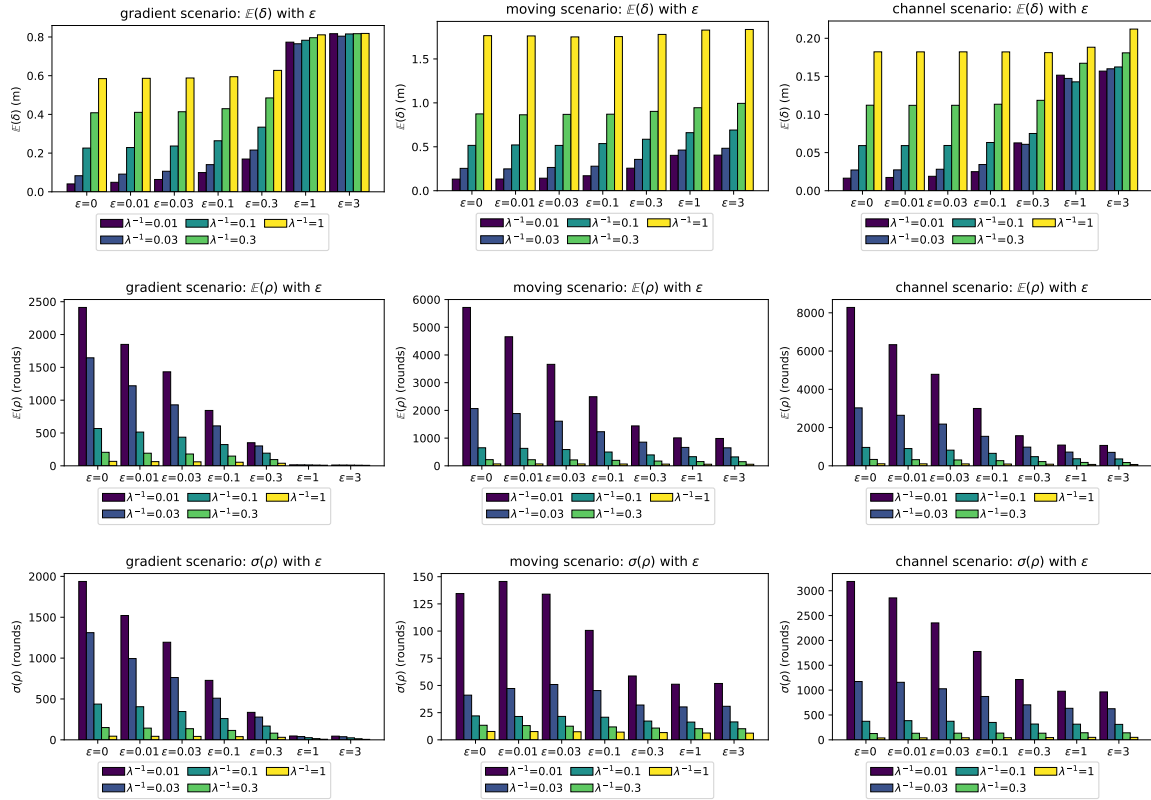
Figure 21: Time-fluid algorithms comparison for different values of tolerance ϵ (abscissa) and mean network delivery time λ^{-1} (colour, the darker the lower). The bar charts show error (top line, lower values are better), cost (middle line, lower values are better), and standard deviation of cost (bottom line, higher standard deviations indicate larger differences between the round frequencies of different devices); gradient, moving, and channel scenarios are depicted respectively in the first, second, and third column. Values are averaged along all the values of $\|\vec{v}\|$ under test.

7. Conclusion and future work

In this work we developed a different way of conceptualising time in field-based coordination systems. Inspired by causal models of space-time in physics, we leverage a novel concept of *causality field* and accordingly introduce a model where field computations may be caused by other field computations or platform-level triggers capturing changes in the computation context. We formalise the approach by extending the field calculus operational framework with a notion of *time-fluid scheduling* over a tree of causally-related field-calculus programs. The model is implemented in the Alchemist-Protelis simulation framework, and an API is proposed to program the scheduling together with the application logic itself. Finally, by means of simulation-based experiments, we show that the time-fluid approach provides significant performance and efficiency benefits with respect to the classical time-driven approach.

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). The relationship between adaptive scheduling and adaptive deployment is also interesting, in particular when considering dynamic architectures and "pulverisation" approaches for application partitioning [BJMS12, CPP+20].

On a more foundational perspective of aggregate computing and field-based coordination, there are several research direction. First, it would be interesting to investigate how time-fluid scheduling approach relates to aggregate processes [CVA+21], a recent construct proposal for modelling dynamic numbers of concurrent field computations which dynamically spread in the system. Second, it is interesting to consider the possibility of supporting time-fluidness purely at the linguistic level, by a scheduling construct properly defining the domain of events of another field computation. Finally, other than time-fluidness one shall consider space-fluidness as well, that is, the possibility of defining computations that affect the shape of perceived space (which devices are to be considered neighbours), with the goal of addressing performance of communication but also to opportunistically cover space in an effective way.

References

[ABD+20] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. Field-based coordination with the share operator. *Logical Methods in Computer Science*, 16(4), 2020.

[ACDV17] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. Compositional blocks for optimal self-healing gradients. In *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18-22, 2017*, pages 91–100. IEEE Computer Society, 2017.

[ADNL+15] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. A calculus for attribute-based communication. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 1840–1845, New York, NY, USA, 2015. Association for Computing Machinery.

[AMF08] Anton Ageev, David Macii, and Alessandra Flammini. 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.

[APMW13] Muhammad Asad Arfeen, K. Pawlikowski, D. McNickle, and A. Willig. The role of the weibull distribution in internet traffic modeling. In *Proceedings of the 2013 25th International Teletraffic Congress (ITC)*. IEEE, September 2013.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[AVD+19] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Transactions on Computational Logic*, 20(1):1–55, jan 2019.

[BGZ00] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Process calculi for coordination: From linda to javaspaces. In *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, pages 198–212. 2000.

[BJMS12]   Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using dy-bip. In *SC@TOOLS*, volume 7306 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[BM93]   Özalp Babaoğlu and Keith Marzullo. *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*, page 55–96. ACM Press/Addison-Wesley Publishing Co., USA, 1993.

[BMZ12]   Nicola Bicocchi, Marco Mamei, and Franco Zambonelli. Self-organizing virtual macro sensors. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):2:1–2:28, 2012.

[BPV15]   Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–30, 2015.

[BVPD17]   Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution in the internet of things. *ACM Transactions on Autonomous and Adaptive Systems*, 12(3):12:1–12:29, 2017.

[CFP+18]   Roberto Casadei, Giancarlo Fortino, Danilo Pianini, Wilma Russo, Claudio Savaglio, and Mirko Viroli. Modelling and simulation of opportunistic IoT services with aggregate computing. *Future Generation Computer Systems*, sep 2018.

[CPP+20]   Roberto Casadei, Danilo Pianini, Andrea Placuzzi, Mirko Viroli, and Danny Weyns. Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment. *Future Internet*, 12(11):203, 2020.

[CPVN19]   Roberto Casadei, Danilo Pianini, Mirko Viroli, and Antonio Natali. Self-organising coordination regions: A pattern for edge computing. In *Lecture Notes in Computer Science*, pages 182–199. Springer International Publishing, 2019.

[CROM19]   Giovanni Ciatto, Lorenzo Rizzato, Andrea Omicini, and Stefano Mariani. Tusow: Tuple spaces for edge computing. In *28th International Conference on Computer Communication and Networks, ICCCN 2019, Valencia, Spain, July 29 - August 1, 2019*, pages 1–6. IEEE, 2019.

[CVA+21]   Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. Engineering collective intelligence at the edge with aggregate processes. *Engineering Applications of Artificial Intelligence*, 97:104081, 2021.

[CVAD20]   Roberto Casadei, Mirko Viroli, Giorgio Audrito, and Ferruccio Damiani. Fscafi : A core calculus for collective adaptive systems programming. In *ISoLA (2)*, volume 12477 of *Lecture Notes in Computer Science*, pages 344–360. Springer, 2020.

[DB16]   Soura Dasgupta and Jacob Beal. 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, NV, USA, December 12-14, 2016*, pages 7282–7287. IEEE, 2016.

[dLGG+13] Rogério de Lemos, David Garlan, Carlo Ghezzi, Holger Giese, Jesper Andersson, Marin Litoiu, Bradley R. Schmerl, Danny Weyns, Luciano Baresi, Nelly Bencomo, Yuriy Brun, Javier Cámara, Radu Calinescu, Myra B. Cohen, Alessandra Gorla, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jézéquel, Sam Malek, Raffaela Mirandola, Marco Mori, Hausi A. Müller, Romain Rouvoy, Cecília M. F. Rubira, Éric Rutten, Mary Shaw, Giordano Tamburrelli, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, and Franco Zambonelli. Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*, pages 3–30, 2013.

[FHA99]   Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[Fis83]   Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International conference on fundamentals of computation theory*, pages 127–140. Springer, 1983.

[FPBV17]   Matteo Francia, Danilo Pianini, Jacob Beal, and Mirko Viroli. 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, September 18-22, 2017*, pages 27–32, 2017.

[FSM+13]   Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluís Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1):43–67, 2013.

[HH17]   S. Hoyer and J. Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1), 2017.

[HHCC18]   Yao-Hua Ho, Yu-Te Huang, Hao-Hua Chu, and Ling-Jyh Chen. Adaptive sensing scheme using naive bayes classification for environment monitoring with drone. *International Journal of Distributed Sensor Networks*, 14(1), 2018.

[HHM⁺00]   J-M Hélary, Michel Hurfin, Achour Mostefaoui, Michel Raynal, and Frédéric Tronel. Computing global functions in asynchronous distributed systems with perfect failure detectors. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):897–909, 2000.

[Hun07]   J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007.

[IPW01]   Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[JMB05]   Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, 2005.

[KRJ09]   Johnsen Kho, Alex Rogers, and Nicholas R. Jennings. Decentralized control of adaptive sampling in wireless sensor networks. *ACM Transactions on Sensor Networks*, 5(3):19:1–19:35, 2009.

[Lam78]   Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LJ07]   Isabelle Linden and Jean-Marie Jacquet. On the expressiveness of timed coordination via shared dataspaces. *Electronic Notes in Theoretical Computer Science*, 180(2):71–89, 2007.

[LLM17]   Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. Asynchronous distributed execution of fixpoint-based computational fields. *Logical Methods in Computer Science*, 13(1), 2017.

[Lob08]   Francisco SN Lobo. Nature of time and causality in physics. In *Psychology of time*, pages 395–422. Emerald Group Publishing Limited Bingley, 2008.

[LYC18]   Jeongjin Lee, Gunjae Yoon, and Hoon Choi. 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*, pages 395–397, 2018.

[M⁺88]   Friedemann Mattern et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[MO12]   Stefano Mariani and Andrea Omicini. Self-organising news management: the molecules of knowledge approach. In *Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2012, Lyon, France, September 10-14, 2012*, pages 235–240. IEEE Computer Society, 2012.

[MW06]   Ronaldo Menezes and Alan Wood. The fading concept in tuple-space systems. In Hisham Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 440–444. ACM, 2006.

[MZ06]   Marco Mamei and Franco Zambonelli. *Field-based coordination for pervasive multiagent systems*. Springer series on agent technology. Springer, 2006.

[MZ09]   Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Transactions on Software Engineering and Methodology*, 18(4):15:1–15:56, 2009.

[OZ98]   Andrea Omicini and Franco Zambonelli. Co-ordination of mobile information agents in tucson. *Internet Research*, 8(5):400–413, 1998.

[PCVN21]   Danilo Pianini, Roberto Casadei, Mirko Viroli, and Antonio Natali. Partitioned integration and coordination via the self-organising coordination regions pattern. *Future Generation Computing Systems*, 114:44–68, 2021.

[Pia21]   Danilo Pianini. Danysk/experiment-2020-lmcs-timefluid: Release 1.0.0, 2021.

[PMV13]   Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *Journal of Simulation*, 7(3):202–215, 2013.

[PMVZ20]   Danilo Pianini, Stefano Mariani, Mirko Viroli, and Franco Zambonelli. Time-fluid field-based coordination. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020,*

Valletta, Malta, June 15-19, 2020, Proceedings, volume 12134 of *Lecture Notes in Computer Science*, pages 193–210. Springer, 2020.

[PVB15]   Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853, 2015.

[RBB+11] Alberto Rosi, Matteo Berti, Nicola Bicocchi, Gabriella Castelli, Alessandro Corsini, Marco Mamei, and Franco Zambonelli. Landslide monitoring with sensor networks: experiences and lessons learnt from a real-world deployment. *International Journal of Sensor Networks*, 10(3):111–122, 2011.

[Rov90]   Carlo Rovelli. Quantum mechanics without time: A model. *Physical Review D*, 42(8):2638–2646, October 1990.

[Rov96]   Carlo Rovelli. Relational quantum mechanics. *International Journal of Theoretical Physics*, 35(8):1637–1678, August 1996.

[Rov98]   Carlo Rovelli. Loop quantum gravity. *Living Reviews in Relativity*, 1(1), January 1998.

[RSH+11] Injong Rhee, Minsu Shin, Seongik Hong, Kyunghan Lee, Seong Joon Kim, and Song Chong. On the levy-walk nature of human mobility. *IEEE/ACM Transactions on Networking*, 19(3):630–643, June 2011.

[SBK05]   Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks*, 3(3):281–323, 2005.

[SRM+13] Daniel Schuster, Alberto Rosi, Marco Mamei, Thomas Springer, Markus Endler, and Franco Zambonelli. Pervasive social context: Taxonomy and survey. *ACM Transactions on Intelligent Systems and Technology*, 4(3):46:1–46:22, 2013.

[TBR+17] Jonas Traub, Sebastian Breß, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 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*, pages 586–597, 2017.

[TPD18]   Demetris Trihinas, George Pallis, and Marios Dikaiakos. Low-cost adaptive monitoring techniques for the internet of things. *IEEE Transactions on Services Computing*, pages 1–1, 2018.

[VAB+18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Transactions on Modeling and Computer Simulation*, 28(2):1–28, mar 2018.

[VBD+18] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From field-based coordination to aggregate computing. In *Lecture Notes in Computer Science*, pages 252–279. Springer International Publishing, 2018.

[VBD+19] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, December 2019.

[VC09]   Mirko Viroli and Matteo Casadei. Biochemical tuple spaces for self-organising coordination. In *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, pages 143–162. 2009.

[VPB12]   Mirko Viroli, Danilo Pianini, and Jacob Beal. Linda in space-time: An adaptive coordination model for mobile ad-hoc environments. In *Coordination Models and Languages - 14th International Conference, COORDINATION 2012, Stockholm, Sweden, June 14-15, 2012. Proceedings*, pages 212–229, 2012.

[ZDK15]   V. Zaburdaev, S. Denisov, and J. Klafter. Lévy walks. *Reviews of Modern Physics*, 87(2):483–530, Jun 2015.