

Chapter 2

Coordination of Distributed Systems

Abstract This chapter provides an overview of those approaches to coordination of distributed systems (programmable in Sect. 2.1, probabilistic in Sect. 2.2) which directly motivated, inspired, and influenced the approach to coordination in self-organising systems proposed in Chap. 3. A brief argumentation on why the approaches are all tuple-based, and on which benefits this brings [24], is provided as a preparatory background, describing the seminal LINDA model [11] which almost all the described approaches are built on top of.

2.1 Tuple-Based Coordination

LINDA has been proposed by David Gelernter in [11] as a distributed programming language exploiting the notion of *generative communication*—introduced in the same paper. Nevertheless, it is suddenly recognised by Gelernter himself as a full-fledged computational model whose implications go beyond distributed communication. In fact, LINDA is nowadays mostly considered a *coordination model* [6], rather than merely a distributed programming language.

The LINDA language consists of three primitives: `out`, `in`, `rd`. Respectively, they allow processes to produce, consume, and observe *tuples* stored in a *tuple space*. Tuples are named, ordered collections of heterogeneous, typed values and/or variables—usually, tuples containing some variables are called *templates* (or, anti-tuples). A tuple space is the abstract computational environment in which LINDA programs are executed, thus, it includes tuples as well as processes using LINDA primitives.

The semantics of LINDA primitives is what makes the LINDA language particularly suitable to be used as a coordination model: whereas the `out` primitive always puts a tuple in the tuple space, `in` and `rd` *attempt* to get one—respectively, withdrawing it or not. In fact, if a tuple *matching* the variables (possibly) used in the `in` or `rd` primitive is found that tuple is returned and the caller process may continue execution; otherwise, the caller process is *suspended* until a matching tuple becomes available. This execution semantics is often referred to as LINDA *suspensive semantics*, and is

one of its distinguishing features enabling coordination policies to be programmed solely on top of the LINDA language.

In his paper Gelernter does not further specifies `out` semantics. Much more recently, [4] distinguishes two admissible semantics regarding actual tuple insertion in the tuple space w.r.t. to the producer process, deeply affecting the computational expressiveness of the LINDA model. In the following, it is always assumed the *ordered semantics* of `out` primitive—as well as of any other insertion primitive which could be mentioned.

What enables the suspensive semantics is the primary contribution of Gelernter's paper, that is, *generative communication*. In generative communication, the data items communicated—that is, the tuples—live independently w.r.t. their producers, since they are (possibly, persistently) stored by the tuple space. This also enables uncoupling in space and time: in fact, sender (receiver) processes need not to know when and where receivers (senders) will be executing in order to successfully communicate.

The last distinguishing feature of the LINDA model is the *associative access* to data. In fact, interacting processes need not to know the address where a tuple is stored to access it: they simply have to know their name and content—even partially thanks to variables. This enables a third form of uncoupling: reference uncoupling (called communication orthogonality in [11]), w.r.t. both tuples and senders/receivers.

LINDA appears well suited for supporting coordination in systems featuring, e.g.: (i) distribution, by relying on multiple tuple spaces installed on networked hosts; (ii) openness, thanks to its uncoupling facilities; (iii) incomplete information, handled by associative access. Knowledge-intensive sociotechnical systems (STS) enjoy all these features, nevertheless have many more which cannot be dealt with effectively by LINDA as it is; in particular, *uncertainty* and *unpredictability*, and the need for *adaptiveness*.

For this reason, the following sections review some coordination models and infrastructures, either inspired to LINDA or implementing LINDA in interesting ways, all having in common the goal of dealing with the aforementioned shortcomings—some by looking at natural metaphors for implementation, some by extending the model.

2.1.1 On Distribution

As described in previous section, LINDA has been originally conceived as a language for decorating concurrent/parallel programs running on a *shared memory* model. Thus, a *single* global tuple space was sufficient for the task of coordinating interacting *co-located* processes. As soon as LINDA potential to become a full-fledged

coordination for *distributed* processes was recognised, the need for working with *multiple* tuple spaces, possibly spread among networked hosts arose.

Many research works since then proposed either LINDA extensions, implementations, or brand new models featuring multiple tuple spaces distributed over a network: Gelernter himself [12], Ciancarini with the PoliS model [5], KLAIM [7] and LIME [23] also adding mobility-related aspects, are just a few examples. In the following, another contribution is described for it is exploited in Chap. 4 as the infrastructure supporting situated coordination: TuCSoN [20].

2.1.1.1 TuCSoN

TuCSoN [20] is a model for the coordination of *distributed* processes, as well as of autonomous agents, extending in a number of ways the basic LINDA model. In TuCSoN, an extension of tuple spaces called tuple centres [19] can be spread (i) over *multiple machines* connected in a network, or (ii) on the same machine provided they belong to different TuCSoN nodes, which are, essentially, containers of locally available tuple centres—each TuCSoN nodes sharing a host must be reachable on a different network address.

Focussing on the distribution perspective, TuCSoN extends the basic LINDA model by:

- assigning *globally unique* IDs to tuple centres
- allowing processes to invoke TuCSoN primitives—LINDA primitives plus many others—*remotely*, that is, from network reachable hosts
- allowing processes to exploit *any number* of local or remote tuple centres for coordinating with other processes, simultaneously
- allowing tuple centres themselves to invoke TuCSoN primitives on each other, wherever they are—locally available or remotely reachable
- supporting role-based access control policies [27] for dealing with privacy and security issues

It is worth to note that, in any distributed setting, it is fundamental that coordination primitives are designed and implemented to be *asynchronous by default*, that is, to avoid *coupling* control flow of the invoker process with network waits. Differently from LINDA in fact, where the network is not considered thus absence of a matching tuple when a getter primitive is invoked is the only cause of suspension for processes, in TuCSoN network level issues may cause unexpected suspension at the application level, e.g., because a network link breaks during invocation of a getter primitives, which is then bound to wait *indefinitely* for a result.

For these reasons, TuCSoN adopts a *two-steps semantics* w.r.t. primitives execution:

Invocation —The request of the operation is sent to the target tuple centre, wherever it is—remote or local

Completion—The response of the operation is sent back to the invoker process, wherever it is—remote or local

Any TuCSoN primitive undergoes these two steps, so as to (i) detect as early as possible network issues, and (ii) decouple synchronism of invocation as chosen by the invoker process from synchronism of the underlying implementation as designed in TuCSoN.

2.2 Programmable Coordination

As far as adaptiveness is a main concern, a natural solution to support it is to allow some degree of *programmability* of the coordination machinery, e.g., enabling interacting agents to change the coordination laws, or even the coordination medium to change them itself, at run-time. Among the many existing approaches to programmable coordination, here follows description of those which proven to be particularly influential for the model described in Chap. 3: LGI [16], GAMMA [2], ReSpecT [18], and TOTA [15].

It should be noted that LGI is not tuple based, but message-based, and that GAMMA is a model for specification of programs based on multiset rewriting, not a coordination model. Nevertheless, a few reasons motivate their inclusion in the section:

- LGI is among the first examples of programmable coordination machinery
- LGI can be easily implemented in a tuple space-based setting, by mapping messages to tuples
- GAMMA embeds a notion of programmability in the reaction abstraction, allowing arbitrary manipulation of the multiset data structure
- GAMMA too can be easily implemented in a tuple space-based setting, being tuple spaces themselves multisets
- GAMMA reaction model may be interpreted as a coordination model where transformation rules locally enact some coordination policies based on the contextual information provided by data items in the multiset

2.2.1 LGI

LGI (Law Governed Interactions) [16] is a message-based coordination model for open distributed systems providing system designers with the ability to program and

enforce *coordination policies* on agents interactions, called *laws*. LGI design is based on four fundamental software engineering principles:

Enforcement—A coordination policy for an open group needs to be *enforced*

Decentralisation—The enforcement mechanism should *not* require central control

Separation—Coordination policies should be *explicitly* reified, and enforced by a *single* mechanism flexible enough to implement a wide range of policies

Incrementality—It should be possible to deploy and enforce a policy *incrementally*, at run-time, *without* affecting agents and activities not subject to it

An LGI system is called an $\mathcal{L}$ -group $\mathcal{G}$, which is a four-tuple $\mathcal{G} = \langle \mathcal{L}, \mathcal{A}, \mathcal{CS}, \mathcal{M} \rangle$ where:

$\mathcal{L}$ —Is the *law of the group*, that is, an explicit and enforced set of “rules of engagement” between members of the group

$\mathcal{A}$ —Is the set of *agents* belonging to $\mathcal{G}$

$\mathcal{CS}$ —Is a set of *control states*, mutable, and subject to $\mathcal{L}$, one per member of the group

$\mathcal{M}$ —Is the set of messages—called $\mathcal{L}$ -messages—that can be exchanged between members of $\mathcal{G}$ under law $\mathcal{L}$

Laws are defined on precise *events*—such as sending and arrival of $\mathcal{L}$ -messages—and meant to enforce some effect—the *ruling* of the event. Events subject of laws are called *regulated events*. Laws are global because all members of group $\mathcal{G}$ are subject to them, but defined locally for each member, so that:

- each law regulates only events happening locally
- ruling of events depend on the event itself and on the local control state of the agent
- ruling of events may trigger only local operations

A control state $\mathcal{CS}$ is maintained by LGI for each agent in the group, whose semantics is defined by the laws of the $\mathcal{L}$ -group. The control state is meant to track any information relevant for the purpose of law enforcement, thus coordination—such as the role of the agent, communication tokens, etc. The control state is *not* accessible to agents, but only to LGI *controllers* $\mathcal{C}$, whom any agent is assigned to as soon as it participates in a LGI-governed system, which are LGI components in charge of enforcing laws—thus, of the coordination process.

Interaction and coordination in LGI are based on message passing and mediated by controllers, thus among the possible regulated events there are:

$\text{sent}(h, m, y)$ —Occurring whenever a $\mathcal{L}$ -message m sent by agent h to agent y arrives at $\mathcal{C}_h$

$\text{arrived}(x, m, h)$ —Occurring whenever a $\mathcal{L}$ -message m sent by agent x arrives at $\mathcal{C}_h$

Given such events, laws may exploit a few *primitive operations* to affect the state of interaction:

Operations on $\mathcal{CS}$ —Updating the control state of the agent involved in the event
 Operations on $\mathcal{L}$ -messages—Forwarding and delivering messages on behalf of the
 interacting agents

The role of enforcement belongs to controllers and amounts to ensure that (i) any exchange of $\mathcal{L}$ -messages *conforms* to law $\mathcal{L}$, and (ii) the *effects* of events regulation are carried out.

Controllers evaluate laws for each compliant event, and carry out the corresponding event ruling *atomically*, so that the sequence of primitive operations implementing the ruling *does not* interleave with those of any other event occurring locally. LGI laws are expressed by means of *event-condition-action* rules, implemented in Prolog.

Summing up, LGI represents a first successful attempt at injecting programmability in a coordination model, and provides design principles which are widely recognised in subsequent models.

2.2.2 GAMMA

GAMMA [2] is a formal model for specifying programs as *multiset transformers*, ensuring correctness of the programs despite *sequentiality* or *parallelism* of the computational model executing them.

GAMMA rationale is that a program should be derived in a high-level language with no *accidental sequentiality*—that is, sequentiality not related to the logic of the program, but to the underlying computational model eventually adopted for actual execution. This way, such a high-level language can be implemented on either sequential, concurrent, distributed, or parallel machines.

GAMMA relies on a single data structure, the *multiset*, and provides a single operator, the Γ operator, which may be intuitively defined by resorting to a *chemical metaphor*: a computation is a succession of *reactions* which consume elements of the multiset and produce new ones according to specific rules. The computation ends when no more reactions can occur. If one or several reactions may trigger on several subsets at the same time, the choice which is made among them is not deterministic. If the reactions hold for several *disjoint* subsets, the reactions can also be carried out simultaneously. Thus, GAMMA programs may contain implicit parallelism

The Γ operator relies on the following basic operations on multisets:

union—the number of occurrences of an element in $M_1 + M_2$ is the sum of its number of occurrences in M_1 and M_2

difference—the number of occurrences of an element in $M_1 - M_2$ is the difference of its number of occurrences in M_1 and M_2

oneof(M)—yields one arbitrarily selected element of M

Reactions are pairs of closed functions of the form (R, A) enacting the following effect on multiset M : *replace* in M a subset of elements $\{x_1, \dots, x_n\}$ such that

$\mathbb{R}\{x_1, \dots, x_n\}$ by the elements of $\mathbb{A}\{x_1, \dots, x_n\}$. Namely, $\mathbb{R}$ is the set of action *pre-conditions*, whereas $\mathbb{A}$ is the set of action *post-conditions*, in terms of data items present in the multiset.

Being reactions defined in terms of closed functions, they have a purely *local* effect: they replace in the multiset the consumed elements by the produced elements, *independently* of the rest of the multiset. So execution of a GAMMA program can be interpreted as a chaotic process involving several subsets of the multiset in several reactions at any given time.

Summing up, GAMMA represents a first successful attempt at injecting non-determinism and parallelism in computational processes execution.

2.2.3 Tuple Centres and ReSpecT

ReSpecT (Reaction Specification Tuples) is a *logic-based* language for the coordination of complex software systems [18]. ReSpecT promotes a coordination model providing *tuple centres* [19] as programmable, general-purpose coordination media [6]. The behaviour of ReSpecT tuple centres is programmed through the ReSpecT first-order logic language.

A tuple centre is a tuple space enhanced with the possibility to program its behaviour in response to interactions. First of all, coordinated entities (ReSpecT agents, henceforth, or simply agents) can operate on a ReSpecT tuple centre in the same way as on a standard LINDA tuple space: by exchanging tuples—which are first-order logic terms, in the case of ReSpecT—through a simple set of coordination primitives. Accordingly, a tuple centre enjoys all the many features of a tuple space mentioned in Sect. 2.1, that is, generative communication, associative access, and suspensive semantics.

Then, while the basic tuple centre model is independent of the type of tuple, ReSpecT tuple centres adopt logic tuples—both tuples and tuple templates are essentially Prolog facts—and *logic unification* is used as the tuple-matching mechanism. Since the overall content of a tuple centre is a multiset of logic facts, it has a twofold interpretation as either a collection of messages, or a (logic) *theory of communication* among agents, thus promoting in principle forms of reasoning about communication.

Finally, a tuple centre is a *programmable* tuple space, so as to add programmability of the coordination medium as a new dimension of coordination. While the behaviour of a tuple space in response to interaction events is *fixed*—so, the effects of coordination primitives are fixed—the behaviour of a tuple centre can be *tailored* to the system needs by defining a set of *specification tuples*, or *reactions*, which determine how a tuple centre should react to incoming/outgoing events. While the basic tuple centre model is not bound to any specific language to define reactions, ReSpecT tuple centres are programmed through the ReSpecT logic-based specification language.

The **ReSpecT** coordination language is a logic-based language for the specification of the behaviour of tuple centres. As a behaviour specification language, **ReSpecT** (i) enables the definition of computations within a tuple centre, called *reactions*, and (ii) makes it possible to associate reactions to *events* occurring in a tuple centre. So, **ReSpecT** has both a *declarative* and a *procedural* part. As a specification language, it allows events to be declaratively associated to reactions by means of specific logic tuples, called *specification tuples*, whose form is $\text{reaction}(E, R)$. In short, given an event E_v , a specification tuple $\text{reaction}(E, R)$ associates a reaction $R\theta$ to E_v if $\theta = \text{mgu}(E, E_v)$ —where *mgu* is the most general unifier, in Prolog terminology. As a reaction language, **ReSpecT** enables reactions to be procedurally defined in terms of sequences of logic reaction *goals*, each one either succeeding or failing. A reaction as a whole succeeds if all its reaction goals succeed, and fails otherwise. Each reaction is executed sequentially with a *transactional semantics*: so, a failed reaction has no effect on the state of a tuple centre.

All the reactions triggered by an event are executed before serving any other event: so, agents perceive the result of serving the event and executing all the associated reactions altogether as a single transition of the tuple centre state. As a result, the effect of a coordination primitive on a tuple centre can be made as complex as needed by the coordination requirements of a system.

Generally speaking, since **ReSpecT** has been shown to be Turing-equivalent [9], any computable coordination law could be encapsulated into a **ReSpecT** tuple centre. This is why **ReSpecT** can be assumed as a general-purpose *core* language for coordination: a language that could be used to represent and enact policies and rules (laws) for coordination in systems of any sort.

Summing up, **ReSpecT** is a coordination language supporting programmability of coordination primitives and laws, allowing run-time modifications of the semantics of primitives, as well as run-time adjustment of coordination laws, thus enabling and promoting *adaptiveness* of the overall coordination logic.

2.2.4 TOTA

Tuples On The Air (TOTA) [15] is a programming model and middleware for supporting *adaptive context-aware activities* in pervasive and mobile computing scenarios—in particular, the development of those *nature-inspired, self-organising coordination* schemes (e.g., ant foraging, flocking) that are increasingly finding useful applications in modern distributed systems [1, 3, 21].

The key idea in TOTA is to rely on *spatially distributed tuples*, propagated across a network on the basis of application-specific rules. As will be apparent at the end of the TOTA model description, it is quite a strong departure from the original LINDA model, although being still tuple-based—e.g., no suspensive semantics. In particular, rather than to *enforce* coordination, TOTA is meant to *enable* (context-aware) coordination, to be actually programmed and supported at the application level.

In the TOTA middleware, all interactions between agents take place in a *fully uncoupled* (in space, time, reference) way via tuple exchanges. However, there is not any notion like a centralized shared tuple space as in original LINDA. Rather, a tuple can be injected into the network from any node and, after cloning itself, can diffuse across the network according to tuple-specific *propagation rules*. Once a tuple is spread over the network, it can be perceived as a single distributed data structure called *tuple field*—made up by all the tuples created during the propagation of the injected tuple—to draw an analogy with physical fields (e.g., gravitational), which have different values (in TOTA, tuples) at different points in space (in TOTA, network nodes).

On the one hand, the middleware takes care of propagating the tuples and adapting their values in response to the dynamic changes that can (possibly) occur in the network topology. On the other hand, agents can exploit a simple API to define and inject new tuples in the network and to *locally sense* nearby tuples as well as associated *events* (e.g., arrival and dismissing of tuples). This ultimately enables agents to perform *context-aware coordinated activities*.

To support this idea, TOTA assumes the presence of a peer-to-peer network of possibly mobile nodes, each running a local instance of the TOTA middleware. Each TOTA node holds references to a limited set of *neighbour nodes* and can *communicate directly* only with them. The structure of the network, as determined by the neighbourhood relations, is automatically maintained and updated by the nodes to support dynamic changes, either due to nodes' mobility or to their birth/death.

TOTA tuples T can be defined at the application level and are characterised by a *content* C , a *propagation rule* P and a *maintenance rule* M , hence $T = \langle C, P, M \rangle$:

Content C —An ordered set of typed elements representing the information conveyed by the tuple.

Propagation rule P —Determines how the tuple should be distributed across the network—ultimately determining the “shape” of the field. Propagation typically consists of a tuple (i) cloning itself, (ii) being stored in the local tuple space, then (iii) moving to neighbour nodes—recursively. However, different kinds of propagation rules can determine the “scope” of the tuple—e.g., the distance at which such tuple should be propagated, the spatial direction of propagation, etc.—and how such propagation can be affected by the presence or the absence of other tuples in the system. In addition, the propagation rule can determine how the tuple's content C should change during propagation—thus tuples are not necessarily distributed replicas.

Maintenance rule M —Determines how a tuple should react to *events* occurring in the environment—including flow of time. On the one hand, maintenance rules can preserve the proper spatial structure of tuple fields despite network dynamics—thanks to TOTA middleware constantly monitoring network topology and the income of new tuples, eventually re-propagating tuples. On the other hand, tuples can be made *time-aware*, e.g., to support temporary tuples or tuples that slowly “evaporate”—in the spirit of pheromones [21].

From the architectural viewpoint, the TOTA middleware supporting the above model is constituted by three main parts:

TOTA API —The main interface between the application agents and the middleware. It provides functionalities to let application agents inject new tuples in the system, retrieve tuples, and place subscriptions to tuple-related and network-related events to the *event interface*.

Event Interface —The component in charge of *asynchronously* notifying the application agents about subscribed events, like the income of a new tuple or the fact that a new node has been connected/disconnected to the node's neighbourhood.

TOTA Engine —In charge of receiving tuples injected from the application level, sending them to neighbour nodes according to their propagation rule, and updating/re-propagating them according to their maintenance rule. To this end, this component continuously monitors network reconfiguration, the income of new tuples, and possibly other events.

As regards TOTA API, here follows a deeper explanation of three TOTA primitives—for the others, in particular regarding reading specific instances of field tuples, (un)subscription to events, and tuple deletion, refer to [15]:

inject —Used to inject a tuple in the TOTA network.

read —Accesses the *local* tuple space and returns a collection of the tuples present in the tuple space which match the tuple *template* passed as parameter. A template is a TOTA tuple in which some of the content elements can be left uninitialised (*null*). These null elements are the formal parameters of traditional LINDA models. Shifting to OO pattern-matching, a template tuple *Tmpl* and a tuple *T* match if and only if the following holds:

- *Tmpl* is an instance of either the class of *T* or one of its superclasses; this extends the LINDA model [13] according to object orientation by supporting match also between tuples of different types, provided they belong to the same class hierarchy
- the non-null elements of *Tmpl* that represent primitive types (*int*, *char*, *boolean*, etc.) have the same value of the corresponding elements in *T* and the non-null, non-primitive elements (objects) of *Tmpl* are equal—in their serialised form—to the corresponding ones of *T*

readOneHop —Returns a collection of the tuples present in the tuple spaces of the node's one-hop neighbourhood that match the template tuple.

Notice both *read* and *readOneHop* operations are *synchronous* and *nonblocking*—no suspensive semantics *by default*¹: they return either all the tuples matching the given template or the empty set if no matching tuples are found.

¹It is still possible to realize blocking operations using the event-based interface: an agent may simply subscribe to a specific tuple and wait until the corresponding reaction is triggered to resume its execution—more on this in [15].

Summing up, TOTA is a novel programming model loosely inspired by LINDA, extending OO tuples with embedded “executable” code system designers may exploit to build *distributed computational fields* with the aim of supporting context-awareness—mostly, *spatial-awareness*—at the application level, so as to enable adaptive, spatial-sensitive, self-organising coordination among application agents. To support this vision, [15] elaborates on a couple of examples regarding crowd steering toward POIs and routing to meet-up in a museum scenario.

Some of the mechanisms programmed by means of TOTA tuples in such case studies are very similar to those bio-inspired mechanisms already known in the literature: the meeting task, involving agents attracted towards each other, is similar to those chemotaxis mechanisms allowing bacteria to move in a coordinated way [17, 21], whereas the routing mechanism involved in POIs discovery is instead similar to ant-based routing [3], where ants follow pheromone trails to reach food. This effectively demonstrates the effectiveness of the TOTA middleware in supporting self-organising—mostly field-based—coordination.

2.3 Probabilistic Coordination

A natural solution to account for *unpredictability* and *uncertainty* is to embrace *stochasticity* by tolerating probabilistic rather than deterministic computations and decision making. Among the many existing stochastic approaches to coordination—e.g., digital pheromones [22], biochemical tuple spaces [26], probabilistic pi-calculus [14], probabilistic and stochastic KLAIM [8, 10]—here follows description of the two which proven to be particularly influential for the model proposed in Chap. 3: pKLAIM [10] and SwarmLinda [25].

2.3.1 pKLAIM

pKLAIM [10] is a probabilistic extension to KLAIM [7], a *kernel language* for formal modelling of interactions among *mobile agents* distributed in a network—and coordinating by putting and consuming tuples in nodes.

Understanding the details of KLAIM and pKLAIM is outside the scope of this section, thus only the way pKLAIM exploits probability is discussed here.

pKLAIM introduces probabilities in KLAIM at different levels:

- at the *local* level, that is, considering network nodes in isolation:
 - probabilistic *parallel* and *choice* operators allow processes to express probabilistic behaviours

- probabilistic *allocation environments* associate distributions on physical sites (network nodes) to logical localities—which means, logical names used in processes specifications are probabilistically assigned to network nodes at run-time
- at the *global* level, that is, considering the network of distributed agents and nodes as a whole:
 - a *discrete time* variant of pKLAIM considers a *probability* associated with each node, indicating the *chance* that the process at that node will be selected for execution
 - a *continuous* time variant, instead, associates a *rate* to each node, determining how *often* the node is active—that is, able to perform action transitions

As far as the discrete time execution model is considered, the update principle simulates a *global scheduler*: at every time step one node is scheduled for execution according to probabilities the nodes' probabilities. In case of continuous time semantics, state transitions do not occur regularly like in the discrete case, but the time between state transitions is *exponentially distributed*—that is, driven by a Poisson process. This way, each node can trigger a network update *independently*, at any given time, with a certain probability.

Summing up, pKLAIM provides a probabilistic coordination mobile for mobile agents, where their actions are executed probabilistically according to a number of dimensions, such as intrinsic probability of action, local schedulers (choice and parallel composition), allocation environments.

2.3.2 *SwarmLinda*

SwarmLinda [25] is the proposal of a scalable implementation of the LINDA model based on *swarm intelligence* techniques, drawing inspiration from ant colonies [21].

One can understand the world of SwarmLinda as a terrain (network of tuple spaces) in which ants (tuple templates) search for food (tuples), leaving pheromone trails upon successful matches, indicating the *likelihood* that further matches for that template are available. Ants look for food in the proximity of the anthill (the caller process); when found, the food is brought to the anthill² and a trail is left so that other ants can know where the food is.

Digital ants behave according to the following rules:

1. spread the scent of the caller process in the node it is interacting with and its neighbourhood, to represent the anthill
2. check for a matching tuple: if a match is found, return to the anthill leaving scent for the template matched at each step (tuple space traversed); if no match is found, check the 1-hop neighbourhood for traces of the desired scent

²The ants know the way back to the anthill because (i) they have a short memory of the last few steps they took and (ii) the anthill has a distinctive scent.

3. if no desired scent is found, *randomly* choose a neighbour space to continue search
4. if a desired scent is found, move one step towards that scent and start over

The key to scalability lies in the *local* nature of ants perceptions: they carry out local searches solely, and inquire direct neighbours only. Furthermore, *probabilistic non-determinism*—necessary for supporting adaptiveness and fault-tolerance [25]—is achieved by adding a small random factor to each scent: this enables paths other than the one with the strongest scent to be chosen.

This results in the *emergence* of application-specific paths between tuple producers and consumers. Moreover, given that scents are *volatile* thanks to an evaporation mechanism, the paths found can dynamically adapt to changes in the system—e.g., when consumers or producers join, leave or move, or in case of failures in nodes hosting tuple spaces.

Besides tuples searching, ant colonies inspiration is also used to partition tuple spaces dynamically, that is, in brood sorting [21]. In nature, ants are able to sort different kinds of things kept in the anthill, such as food, larvae, eggs, etc., and do so in spite of the amount of each type, thus being very scalable. In SwarmLinda, tuples are the things to sort and the ant is the active process representing an `out`, thus:

1. upon execution of an `out`, start visiting the nodes
2. observe the *kind* of tuples the nodes store, that is, the template they match
3. store the tuple in the node if nearby nodes store tuples matching the same template, considering a small random factor
4. if nearby nodes do not contain similar tuples, randomly choose whether to drop or continue to carry the tuple to another node

To guarantee convergence to meaningful partitions, certain conditions must be satisfied: (i) for each time the process decides not to store the tuple, the random factor will tend to $\varepsilon \approx 0$ so as to increase the chance of storing the tuple in next steps; (ii) the likelihood of locally storing the tuple is calculated probabilistically, based on the kinds of objects in the ant's memory, that is, if most of them are similar to the one being carried, the likelihood to deposit the tuple increases.

Summing up, SwarmLinda is a nature-inspired implementation of the LINDA model, accounting for *uncertainty* and *unpredictability*—e.g., of tuples' location and of agents' interactions—by embracing probability in the mechanisms supporting tuples searching and storage. *Adaptiveness* is enabled in turn, by leveraging *stochasticity* in both resource-to-consumer paths formation and tuples partitioning.

References

1. Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., et al.: Design patterns from biology for distributed computing. *ACM Trans. Auton. Adapt. Syst. (TAAS)* **1**(1), 26–66 (2006)
2. Banâtre, J.P., Métayer, D.L.: The gamma model and its discipline of programming. *Sci. Comput. Program.* **15**(1), 55–77 (1990). doi:[10.1016/0167-6423\(90\)90044-E](https://doi.org/10.1016/0167-6423(90)90044-E)

3. Bonabeau, E., Dorigo, M., Theraulaz, G.: *Swarm Intelligence*. Oxford (1999)
4. Busi, N., Gorrieri, R., Zavattaro, G.: On the expressiveness of linda coordination primitives. *Inf. Comput.* **156**(1), 90–121 (2000)
5. Ciancarini, P.: Polis: a programming model for multiple tuple spaces. In: *Software Specification and Design, 1991., Proceedings of the Sixth International Workshop on*, pp. 44–51 (1991). doi:[10.1109/IWSSD.1991.213078](https://doi.org/10.1109/IWSSD.1991.213078)
6. Ciancarini, P.: Coordination models and languages as software integrators. *ACM Comput. Surv.* **28**(2), 300–302 (1996). doi:[10.1145/234528.234732](https://doi.org/10.1145/234528.234732)
7. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: A kernel language for agent interaction and mobility. *IEEE Trans. Softw. Eng.* **24**(5), 315–330 (1998). doi:[10.1109/32.685256](https://doi.org/10.1109/32.685256)
8. De Nicola, R., Latella, D., Katoen, J.P., Massink, M.: StoKlaim: A stochastic extension of Klaim. Tech. Rep. 2006-TR-01, Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo" (ISTI) (2006). <http://www1.isti.cnr.it/~Latella/StoKlaim.pdf>
9. Denti, E., Natali, A., Omicini, A.: On the expressive power of a language for programming coordination media. In: *1998 ACM Symposium on Applied Computing (SAC'98)*, pp. 169–177. ACM, Atlanta, GA, USA (1998)
10. Di Pierro, A., Hankin, C., Wiklicky, H.: Probabilistic KLAIM. In: R. De Nicola, G.L. Ferrari, G. Meredith (eds.) *Coordination Models and Languages, LNCS*, vol. 2949, pp. 119–134. Springer, Berlin, Heidelberg (2004). doi:[10.1007/978-3-540-24634-3](https://doi.org/10.1007/978-3-540-24634-3)
11. Gelernter, D.: Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* **7**(1), 80–112 (1985). doi:[10.1145/2363.2433](https://doi.org/10.1145/2363.2433)
12. Gelernter, D.: Multiple tuple spaces in LINDA. In: *Parallel Architectures and Languages Europe (PARLE'89)*, LNCS, vol. 366, pp. 20–27. London, UK (1989)
13. Gelernter, D., Carriero, N.: Coordination languages and their significance. *Commun. ACM* **35**(2), 97–107 (1992). doi:[10.1145/129630.129635](https://doi.org/10.1145/129630.129635)
14. Herescu, O.M., Palamidessi, C.: Probabilistic asynchronous pi-calculus. *CoRR* **cs.PL/0109002** (2001)
15. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **18**(4), 15:1–15:56 (2009). doi:[10.1145/1538942.1538945](https://doi.org/10.1145/1538942.1538945)
16. Minsky, N.H., Ungureanu, V.: Law-Governed interaction: A coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **9**(3), 273–305 (2000). doi:[10.1145/352591.352592](https://doi.org/10.1145/352591.352592)
17. Nagpal, R.: Programmable self-assembly using biologically-inspired multiagent control. In: *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: part 1*, pp. 418–425. ACM (2002)
18. Omicini, A.: Formal ReSpecT in the A&A perspective. *Electron. Notes Theor. Comput. Sci.* **175**(2), 97–117 (2007). doi:[10.1016/j.entcs.2007.03.006](https://doi.org/10.1016/j.entcs.2007.03.006)
19. Omicini, A., Denti, E.: From tuple spaces to tuple centres. *Sci. Comput. Program.* **41**(3), 277–294 (2001). doi:[10.1016/S0167-6423\(01\)00011-9](https://doi.org/10.1016/S0167-6423(01)00011-9)
20. Omicini, A., Zambonelli, F.: Coordination for Internet application development. *Auton. Agent. Multi-Agent Syst.* **2**(3), 251–269 (1999). doi:[10.1023/A:1010060322135](https://doi.org/10.1023/A:1010060322135)
21. Parunak, H.V.D.: “Go to the ant”: Engineering principles from natural agent systems. *Ann. Oper. Res.* **75**(0), 69–101 (1997). doi:[10.1023/A:1018980001403](https://doi.org/10.1023/A:1018980001403). Special Issue on Artificial Intelligence and Management Science
22. Parunak, H.V.D., Brueckner, S., Sauter, J.: Digital pheromone mechanisms for coordination of unmanned vehicles. In: Castelfranchi, C., Johnson, W.L. (eds.) *1st International Joint Conference on Autonomous Agents and Multiagent Systems*, vol. 1, pp. 449–450. ACM, New York, NY, USA (2002). <http://dx.doi.org/10.1145/544741.544843>
23. Picco, G.P., Murphy, A.L., Roman, G.C.: LIME: Linda meets mobility. In: *21st International Conference on Software Engineering (ICSE'99)*, pp. 368–377. ACM Press, New York, NY, USA (1999). doi:[10.1145/302405.302659](https://doi.org/10.1145/302405.302659)
24. Rossi, D., Cabri, G., Denti, E.: Tuple-based technologies for coordination. In: Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R. (eds.) *Coordination of Internet Agents: Models, Technologies, and Applications*, Chap. 4, pp. 83–109. Springer (2001)

25. Tolksdorf, R., Menezes, R.: Using Swarm Intelligence in Linda Systems. In: Omicini, A., Petta, P., Pitt, J. (eds.) *Engineering Societies in the Agents World IV*, LNCS, vol. 3071, pp. 49–65. Springer (2004). doi:[10.1007/b98212](https://doi.org/10.1007/b98212). 4th International Workshops (ESAW 2003), London, UK, 29-31 Oct. 2003. Revised Selected and Invited Papers
26. Viroli, M., Casadei, M.: Biochemical tuple spaces for self-organising coordination. In: Fiel, J., d, Vasconcelos, V.T. (eds.) *Coordination Languages and Models*, LNCS, vol. 5521, pp. 143–162. Springer, Lisbon, Portugal (2009). doi:[10.1007/978-3-642-02053-7](https://doi.org/10.1007/978-3-642-02053-7)
27. Viroli, M., Omicini, A., Ricci, A.: Infrastructure for RBAC-MAS: An approach based on Agent Coordination Contexts. *Appl. Artif. Intell. Int. J.* **21**(4–5), 443–467 (2007). doi:[10.1080/08839510701253674](https://doi.org/10.1080/08839510701253674). Special Issue: State of Applications in AI Research from AI*IA 2005

Coordination of Complex Sociotechnical Systems

Self-organisation of Knowledge in MoK

Mariani, S.

2016, XVIII, 234 p. 66 illus., 58 illus. in color., Hardcover

ISBN: 978-3-319-47108-2