Science of Computer Programming 74 (2009) 702–722

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

On the collective sort problem for distributed tuple spaces

Matteo Casadei *, Mirko Viroli, Luca Gardelli

Alma Mater Studiorum–Università di Bologna, via Venezia 52, 47023 Cesena, Italy

ARTICLE INFO

Article history:
Received 29 June 2007
Received in revised form 15 May 2008
Accepted 15 September 2008
Available online 24 February 2009

Keywords:
Self-organizing systems
Tuple spaces
Stochastic simulations
Collective sort

ABSTRACT

In systems coordinated with a distributed set of tuple spaces, it is crucial to assist agents in retrieving the tuples they are interested in. This can be achieved by sorting techniques that group similar tuples together in the same tuple space, so that the position of a tuple can be inferred by similarity. Accordingly, we formulate the *collective sort* problem for distributed tuple spaces, where a set of agents is in charge of moving tuples up to a complete sort has been reached, namely, each of the N tuple spaces aggregate tuples belonging to one of the N kinds available. After pointing out the requirements for effectively tackling this problem, we propose a self-organizing solution resembling *brood sorting* performed by ants. This is based on simple agents that perform partial observations and accordingly take decisions on tuple movement. Convergence is addressed by a fully adaptive method for simulated annealing, based on *noise* tuples inserted and removed by agents on a need basis so as to avoid sub-optimal sorting. Emergence of sorting properties and scalability are evaluated through stochastic simulations.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Among many scenarios relying on coordination models and languages, the most popular is based on the idea that agents in a distributed system can interact with each other through tuple spaces spread over the network, where tuples can be inserted and retrieved relying on so-called generative communication [17,18,32]. This approach has been shown to support time and space decoupling, as well as to promote a clear separation between the computational part of the system, which should stay inside agents, and the coordination part of the system, implemented through tuple spaces.

In open systems, however, due to the unpredictability of agents' behavior, it is often difficult to know in which tuple space a certain tuple may occur. As a consequence, when an agent needs to retrieve tuples matching a given pattern, the only strategy would be to randomly select one tuple space among the available ones, and try another one in the case the tuple is not found—leading to obvious performance issues. Accordingly, a strategy is required to assure that agents have some knowledge about the location of the tuples of interest, so that such tuples can be more quickly retrieved. A general solution to this problem is to devise approaches for moving tuples to the most proper tuple space, and locate them accordingly. Works like the TOTA middleware [24], SwarmLinda [29], and stochastic KLAIM [30], though starting from different perspectives, all develop the idea of extending the basic tuple space model of LINDA with features related to tuples' repositioning by moving or copying.

Along this idea, in this article we envision a sorting technique for tuple spaces, where a set of *sorting agents* is in charge of moving tuples until each space holds only tuples of the same *kind*. To support this behavior, sorting agents – which are part of the infrastructure providing the coordination service – must be designed so as to agree on what to sort (the set of N tuple spaces subject to ordering), and how to sort (a clustering relation that groups the tuples of interest into N kinds, so that 1 kind can be exactly associated to 1 space). Similarly to sorting in standard data structures like e.g. arrays, such

* Corresponding author. Tel.: +39 0547 6 34710; fax: +39 0547 3 39219.
E-mail addresses: m.casadei@unibo.it (M. Casadei), mirko.viroli@unibo.it (M. Viroli), luca.gardelli@unibo.it (L. Gardelli).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.09.018

an aggregation technique – a case of *segregation* in the context of collective robotics [26] – can be regarded as an approach with the ultimate goal of simplifying the process of finding tuples: if a certain tuple is eventually found in a tuple space, then any tuple of the same kind can be found in the same space. Note that even if full sorting is not completely reached, partial sorting may still improve performance of tuple retrieval since the probability of finding the tuple at the first attempt becomes higher than $1/N$. This technique is hereafter referred to as *collective sort* for distributed tuple spaces.

Such a sorting service is meant to work in the "background" to the standard activity of tuple spaces, that is, ordering of tuples proceeds while *user agents* coordinate their activity by inserting and retrieving tuples. In other words, it is an online service. Unlike standard sorting techniques, hence, here sorting should effectively work in dynamic and unpredictable scenarios where user agents keep moving, inserting, and dropping tuples. Therefore, the tuple space that will eventually aggregate a certain kind of tuples is not known statically: it is chosen implicitly and probabilistically as tuples start aggregating in a space rather than another as a consequence of multiple tuple movements. Hence, we are concerned with robustness and reactiveness to changes other than moving rate: we need sorting to be a property emerging in spite of external interactions—of course, the more the external environment keeps altering a tuple configuration, the more resources must be devoted to sorting if convergence is to be achieved.

By looking at existing systems, we see that interesting related behaviors already manifest in Nature. Ants use a self-organizing technique called *brood sorting* to solve a similar problem [5]: they move items (brood or larvae) based on local and partial criteria, and sorting emerges as a global system property. Inspired by brood sorting, we develop a solution to the collective sort problem. Interestingly, we show that full sorting can be achieved by requiring sorting agents with neither computational ability (intelligence or memory), nor complete observations of tuple spaces—though, such aspects of course impact on the performance of ordering. The proposed solution is based on the following ingredients: (i) probabilistic access to tuples in tuple spaces; (ii) local decision on tuple movement based on a couple of observations (two read operations, on two spaces); and (iii) avoidance of non-optimal sorting by a fully adaptive approach in the style of *simulated annealing* [20].

As for any self-organizing technique, probability is a key aspect. Not only small variations on tuple configuration can lead to completely different system behaviors, but the same can also happen in different system "runs" from the same initial conditions—since tuples are retrieved probabilistically. Therefore, we shall rely on stochastic simulation tools in order to check whether the proposed solution meets our expectations in terms of quality. To this end, we adopt the stochastic simulation library we developed in the MAUDE term rewriting system [9]—of course other tools like e.g. SPIM [33], SWARM [2] and REPAST [1], could be used as well.

The remainder of this article is organized as follows: Section 2 motivates and formulates the collective sort problem, Section 3 describes the proposed solution and its design choices, Section 4 evaluates the approach by stochastic simulations, Section 5 discusses related work, and Section 6 concludes providing final remarks. This article is an extended version of previous material appeared in [9,38], providing a new and refined solution to the collective sort problem, along with an extensive evaluation concerning convergence, scalability and reactiveness to external interactions.

2. Collective sort

In this section we formulate the collective sort problem, analyze motivations, define architectural and behavioral constraints, provide requirements for an effective solution, and finally outline an example scenario of application.

2.1. Motivation

We assume a network composed of tuple spaces and agents, where agents address tuple spaces by identity. Agent interaction relies on so-called generative communication, namely, agents put tuples (records of primitive values) in tuple spaces and later retrieve such tuples by content, that is, using a partial specification known as *tuple template*.

The presence of a tuple in a particular space may affect the behavior of the overall system since it reifies the occurrence of an event related to system coordination. Such events may include: (i) agents requesting services of some kind provided by another agent; (ii) agents providing the outcome of the execution of a service; (iii) agents depositing data values which are part of the overall system state; (iv) agents publishing part of their internal state and knowledge; and (v) agents updating some shared-variable upon which other agents synchronize their activity. When storing a tuple, the choice of the tuple space to be used is critical. If an agent knows the identifier of the tuple space where a specific tuple is stored, the tuple can be retrieved by only one read operation; however, if this is not the case, the agent may end up trying different tuple spaces until finding the right one. Therefore, an agent needs to be aware of the location of the tuples it is interested in; if it is not possible to know their location, even some kind of awareness could be helpful.

In standard data structures, like e.g. arrays, access to data is simplified – i.e. made quicker – by keeping information sorted: typically, a sorting algorithm is exploited along with insertions and removals that preserve sorting, and fast searching operations are conceived so as to leverage sorting. Sorting itself is based on a total order relation over data that is predefined and static.

Our idea is to take a collection of tuple spaces, initially hosting tuples stored in a completely random manner, and apply a similar approach. Since a tuple space is an unstructured and unbounded bag (i.e., a multiset), the only relevant information for an agent is in which tuple space a tuple is located—there is no notion of the "position" of a tuple within a tuple space.

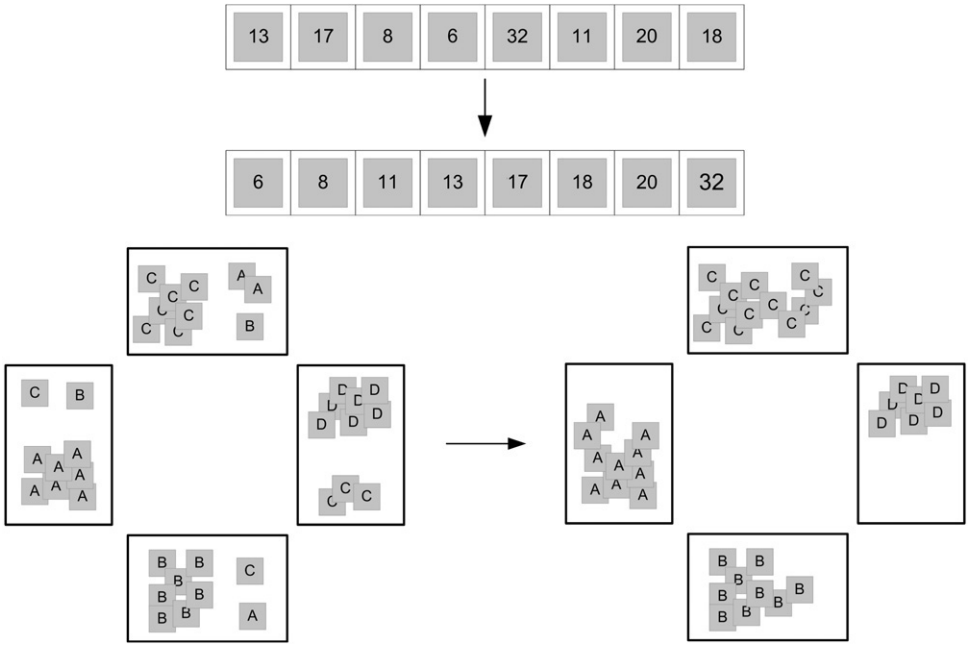


Fig. 1. Array sorting vs. collective sorting.

Accordingly, our goal is to devise a sorting procedure that moves tuples from one space to another until each space holds only tuples of the same *kind*. If a tuple of kind *k* is known to reside in space *s*, all tuples of the same kind can be expected to be located in *s*, thus simplifying searches.

Given the goal of ordering *N* tuple spaces, we hence assume that the tuples to be sorted (which might be a subset of all the tuples in the system) are clustered into *N* kinds, and that sorting agents are aware of such clustering. As for the total order relation used in standard structures like arrays, our order relation may be predefined, that is, conceived at design-time and before actually powering on the sorting service. On the other hand, it would be possible to also change the sorting configuration dynamically – due to changes in the set of spaces, or if a more symmetric clustering relation is to be used – provided sorting agents are all aware of this change. Hence, we do not focus on how kinds are formed: in principle, tuples which are "similar" according to some metric, depending on the coordinated system at hand, should belong to the same kind—however, as far as tuple retrieval is concerned, a kind is better formed by all tuples that match one or more template. See Fig. 1 for a pictorial comparison between standard array sorting, and collective sorting in tuple spaces.

We call the problem of gathering tuples of the same kind over a set of *N* tuple spaces as *collective sort*. It should be clear from now on that, unlike standard algorithms, collective sort is meant to work *at runtime*. While it is typically agreed that the content of an array is frozen during sorting, tuple spaces work in unpredictable dynamic scenarios, so that collective sort needs to be conceived as a background activity aimed at moving the system towards sorting at the same time as user agents keep changing the state of tuple spaces.[1] Depending on the ratio between rate of changes and resources devoted to sorting, the system might evolve according to one of three behaviors: (i) full sorting is achieved (modulo a small noise due to mutations operated by user agents), (ii) a certain level of (partial) sorting can be maintained, and (iii) the system becomes more and more unsorted as time passes, until becoming chaotic—in next section we will quantify the degree of order in terms of entropy. It is worth noting that agents can take advance of a partially sorted system as well, since the probability of finding a tuple of kind *k*, where a previous one was found, is indeed higher than in other tuple spaces. Accordingly, the average retrieving cost is lower than in fully unsorted cases.

2.2. Seeking for a self-organizing solution

Collective sort in distributed tuple spaces is reminiscent of a classical problem in robotics known as *segregation*, where robots roam the ground with the goal of finding, grouping, and separating items—for further details refer to Section 5.

In that context, solutions are typically searched in Nature, which is a rich source of simple but robust strategies. The segregation behavior has already been observed on social insects an referred to as *brood sorting* [6]. When organizing brood and larvae, ants tend to group and keep such items separated from an initial situation where they are randomly situated

[1] One could think of an analogy with concurrent garbage collection, where users processes keep mutating data while the collector tries to manage memory in parallel.

in the ground. Although ants actual "behavior" is still not fully understood, there are several models that are able to mimic the dynamics of the system. Ants wander randomly on the ground and their behavior is modeled by two probabilities, respectively, the probability of picking up P_p and dropping P_d an item, which are evaluated with respect to the recently encountered items. The idea is that an ant (i) picks up an item if its concentration is low with respect to previous experience, (ii) starts wander randomly, and (iii) drops the item where its concentration is higher with respect to where it was picked.

The ant-based solution to brood sorting is intrinsically *self-organizing* [8], that is: (i) it is a process in which a pattern at the global level of the system *emerges* solely as the result of the numerous interactions among the low-level components of the system, and (ii) where the rules specifying interactions among a system's components are executed by using only local information, without any reference to a global pattern. Namely, ants are guided by spatially local observations and motivated by the only need of picking items up where concentration is low, and dropping them where concentration is higher: numerous such interactions make full sorting (i.e. the segregation pattern) emerge at the global level!

The above solution to brood sorting (and any self-organization approach in general) manifests interesting features. First, it is intrinsically robust, since it does not require global information: it promptly reacts to changes in the environment (e.g. new brood, larvae, or ants are dynamically added or removed), to faults like environment splits (e.g. a barrier splitting the ground into two parts), and to local malfunctioning (e.g. some ant behaving in a completely different way). Second, it is intrinsically probabilistic since in the real world small fluctuations always happen which, due to bifurcation effects, might cause the system behavior to globally change—this is indeed a source of robustness. As for all self-organization approaches, performance is of course lower than solutions based on global observations, due to the overhead caused by the need to continuously perform pointwise (local) observations. Hence, self-organizing systems stress the tradeoff between the performance of global approaches and the robustness of local approaches—and considering that global approaches are not always available, as for distributed systems in general. Such solutions are therefore considered interesting for developing robust online services that should function in unpredictable environments, and where performance and robustness live together as key factors.

As a result, it is interesting to seek a solution to collective sort inspired by ants' brood sorting. However, it should be noted that the two application scenarios have key differences, that might require a significant adaptation:

- Topological space — Instead of being a continuous environment, or a network of connected subparts of the environment, our scenario features a flat set of N tuple spaces, each being a conceptually unbounded bag of tuples.
- Agent Mobility — Instead of being performed by mobile agents carrying items with them and wandering randomly in the environment, the sorting service is executed by an infrastructure composed of software agents, each associated to a tuple space. Since sending tuples is typically less expensive than moving software agents, such agents should not be likely or necessary to move.
- Actions and Perceptions – Instead of perceiving items based on a range of locality, such software agents should be able to look for tuples in either the tuple space they are assigned to (called local tuple space), or a different tuple space (called remote tuple space) – the latter operation is necessarily more expensive. Similarly, actions correspond to removing tuples and inserting them elsewhere.

2.3. Architectural and behavioral constraints

Based on the above considerations, a working solution for the collective sort problem is developed that could achieve the robustness properties sought by self-organization approaches. First of all, it is important to characterize all the architectural and behavioral constraints for any candidate solution:

- Service architecture. In a system with N tuple spaces, and multiple *user agents* coordinating their activity through those spaces, collective sort needs to be considered as an online service provided by one or more *sorting agents*, each assigned to exactly one of the N spaces. Each agent works at a certain rate, that is, it executes a number of instances of an interaction protocol per time unit. Of course, it is understood that sorting performance is directly dependent on the sorting rate of such agents.
- Sorting agent behavior. The behavior of sorting agents, and ultimately of the proposed algorithm, is hence described in terms of a (possibly probabilistic) protocol to be executed multiple times by each agent—most likely, taking an observation and accordingly performing some actions, inspired by brood sorting. This protocol is to be the composition of primitive operations over tuple spaces, that is, reading, removal, and insertion of tuples. Note that tuple counting is not allowed by standard tuple space systems, hence the only means for observing a space is to repeatedly read single tuples on it. Some memory and limited symbolic ability might be assumed but this is not mandatory—e.g., the agent may remember what is the last tuple moved, or have the ability to check whether two tuples belong to the same kind or not.
- Data modeling. We assume there is a strict connection between the notion of kind and tuple template, so that it is easy for an agent to get a tuple of a certain kind from a tuple space by asking for a tuple matching a template, or get a tuple of *any* kind by asking for a more generic template. Whereas LINDA does not allow e.g. to look for tuples matching one of n templates, this is not a conceptual or technological problem per se, and can in fact be implemented over existing tuple-based infrastructures such as TuCSoN [32,31]—hence, in the collective sort solution we identify the concept of kind with that of tuple template. Moreover, we assume that reading or removal of tuples matching a given template is a *uniform*

operation that yields a probabilistically fair result [38], namely, among the many tuples matching the template all have the same probability of being chosen.

2.4. Quality attributes

Other than architectural and behavioral requirements, which are meant to shape the structure of a solution, it is also interesting to point out the quality attributes expected from a successful solution, expressed in terms of qualitative and quantitative aspects.

As a way of measuring the degree of sorting of a certain configuration of tuples, we rely on Shannon *entropy* [37], which represents the uncertainty in the observation of a random variable—this is also called information entropy, or simply entropy from now. This is expressed as $K * \sum_{i=1}^{n} p_i \log(p_i)$, where K is any constant value and p_i is the probability for the variable to assume the ith of n possible values. In our case, the ordering of a tuple space can be associated with the variable that represents the kind of a tuple randomly drawn in the space. Given N kinds $k_1, \ldots, k_N$, if we denote with q_k the amount of tuples of kind k, and with n the total number of tuples in the space, then the probability of a tuple to be of kind k_i is q_{k_i}/n, which is basically the concentration c_{k_i} of tuples of kind k_i in the space. Accordingly, the entropy associated with tuple space s can then be computed as:

$$H_s = K * \sum_{i=1}^{N} c_{k_i} \log(c_{k_i}) \tag{1}$$

while the global system entropy is simply the sum of each space entropy, namely $H = \sum_{i=1}^{N} H_{s_i}$, which ranges from 0 to $K * N \log_2 N$. K is then set to $1/(N \log_2 N)$ in order to bound the globally system entropy between 0 and 1, where 0 means complete sorting (each space holds tuples of only one kind), while 1 means complete unsorting (each space has an equal number of tuples per kind).

Concerning the outcome of ordering, the quality attribute we seek for the collective sort solution can hence be listed as:

- Full sorting. In the case of a quiescent system – one in which the *mutation rate* of user agents is zero – complete sorting ($H = 0$) must be reached from any initial configuration. Namely, system evolution should never get stuck in unordered states (where $H > 0$).
- Reactiveness. Given a certain non-zero mutation rate, and a desired level of sorting $H_D < 1$, there should be a sorting rate that leads the system to an entropy value constantly lower than H_D.

Another key issue concerns the performance which can be achieved in sorting. Being an online service, we note that the sorting time strictly depends on the resources devoted to sorting. Supposing as fixed the number of network operations per time unit devoted to sorting, performance can be characterized in terms of the amount of network operations (reading, removal or insertion of tuples) required to achieve sorting.

As a starting reference, we can consider an initial situation with N spaces and kinds, and T tuples per kind chaotically inserted in the spaces ($H = 1$). The optimal algorithm would consider only tuples that are out of place, and accordingly move such tuples directly to the proper destination space: since each space holds T tuples and $T * (N - 1)/N$ of them are out of place, we would have a total of $T * (N - 1)$ tuples moved. However, this result would be accomplished only if mutation rate is zero and global information is available, but this is an ideal situation that does not fit collective sort.

First of all, we cannot suppose that mutation rate is zero. In general, the "reactiveness" seen above also highlights the need of promptly reacting to an unpredictable, possibly significant change to the tuple configuration, e.g. a user agent inserting a significant number of tuples into a single space in a few time units. This means that the sorting service should devote some network resource to observing spaces (intercepting changing situations) and some other to transferring tuples—as mentioned in the previous section. Different policies can be evaluated to trade off the observation rate and transfer rate, so as to either promote convergence time in the static case or prompt reactiveness to changes. Secondly, in our framework we do not have global information readily available. As already mentioned, this is because tuple spaces do not allow one to count the number of tuples matching a given template. Accordingly, multiple probabilistic read operations are used to observe a space.

As a result, collective sort is to be run by continuously observing tuple spaces in order to be reactive to changes in tuple configuration, and by emergently selecting the tuple space that should gather a certain kind of tuple. Hence, this causes an unavoidable overhead with respect to the ideal case above. Moreover, it should be noted that, as in array sorting, the relative overhead obviously increases with the size of the problem, namely with the number of tuples and tuple spaces. As a general rule, we seek the following result:

- Convergence cost. The average "cost" (network operations) for sorting starting from a chaotic configuration where $H = 1$ has to be at most one order of magnitude greater than the solution based on static, global information. Moreover, the sorting cost should be expected to scale at most polynomially with the number of tuples and tuple spaces.

2.5. An example scenario

A key scenario of large-scale distributed systems nowadays is based on wireless sensor networks, namely, systems with a high number of small, wireless devices deployed in the physical world to monitor environmental properties [3].

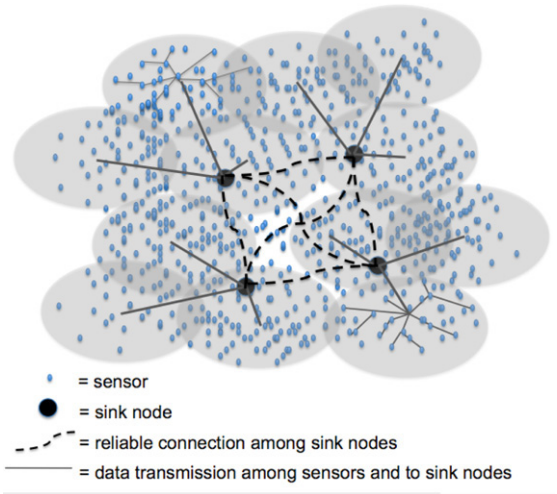


Fig. 2. Applicative scenario with a wireless sensor network and multiple sink nodes.

A fundamental issue in this context is to devise effective strategies to gather all the generated data [21]. Recent works like [10,41] focus on building suitable algorithms for gathering such data into a small set of *sink nodes*.

Based on this context, we show an application scenario for collective sort, as a way to clarify its main motivations and applicability. We suppose that sensors are spread over a wide area to monitor a set of environmental properties (temperature, pressure, humidity, light) and accordingly generate events reporting changes in such properties and/or faults. Such sensors form a highly dynamic set, for they may fail, or move – e.g. since they are deployed on board of a vehicle or a mobile device – or unpredictably hibernate to save energy. A proper infrastructure like that envisioned in [10] is exploited to gather all the generated events into a set of N sink nodes, reliably connected to a stable network—e.g. they can be thought of as environmental stations connected through a WAN. Each sink node is associated with a tuple space that unpredictably receives from sensors tuples of the kind:

```
event(sensor_id,sensor_position, time,property_name,property_value).
```

Hence, sensors will play the role of user agents. Additionally, other user agents – called manager agents – are connected to sink nodes, and are in charge of properly managing all the generated data, executing the following tasks: generating a global map of a certain property, looking for peak values of a certain property, gathering general statistics of faults, erasing old events when updated data arrives, erasing duplicated events if the same one arrives to different sinks, and so on. See Fig. 2 for a pictorial representation of this application scenario.

Whereas having multiple sinks enhances the efficiency of data gathering [10,41], this clearly introduces implementation issues since manager agents are forced to look at the tuples they are interested in throughout the set of spaces: the manager agent in charge of gathering the global map of temperature would look for the template `event(?,?,?,temperature,?)`, the agent looking for faults would look for `event(?,?,fault,fault,fault)`, and so on. In order to avoid the overhead induced by the need of looking for tuples in all the N tuple spaces, it is helpful to set up a collective sort service that at runtime keeps all the generated data sorted in an emergent way. Note that the particular settings of this application are such that it may be hard to assign tuples to spaces statically, basically because the space where certain tuples will be deposited cannot be known at design time—the routing algorithm for sinks may be probabilistic, sensors can move, faults can occur in unpredictable places, certain properties may rapidly evolve only in certain places of the network, and so on.

Accordingly, after setting up a finite set of tuple templates of interest and deciding how they should be clustered into N kinds, N sorting agents can be deployed in the environmental stations, working at a certain sorting rate. As an example with $N = 4$, tuples modeling faults can go to space 1, tuples modeling peak values to space 2, tuples with updates on temperature and pressure on space 3, and finally tuples with updates on humidity and light to space 4. If during sorting, it became clear that one space is gathering many more tuples than others, kinds might be redesigned and sorting agents can be accordingly updated—in the context of this article we treat this aspect as orthogonal to the sorting behavior, though.

As the sorting service is powered on and reaches the desired level of sorting, each manager agent will shortly find the tuple space gathering the tuple it is interested in, thus improving the execution performance of its task.

3. A solution to the problem

In this article we present a solution that satisfies the quality attributes outlined in the previous section. In particular, we aim at showing that this can be obtained without relying on agent's intelligence – as many self-organization approaches highlight – but by devising an agent's behavior based on very simple protocols of basic tuple operations.

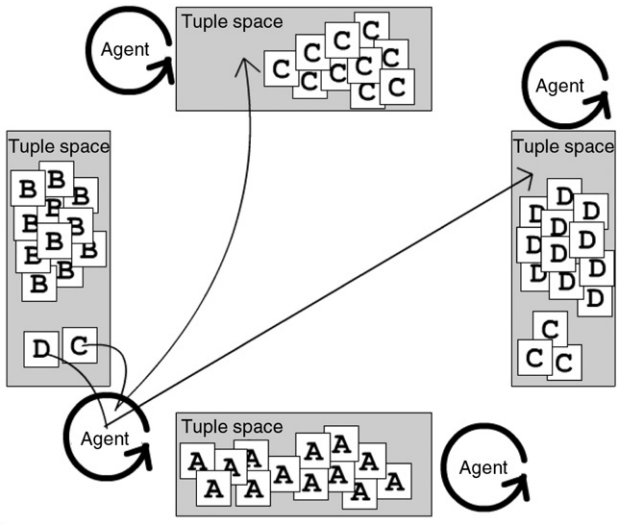


Fig. 3. Architecture for collective sort—The agent on left-bottom should eventually relocate tuples C and D to different tuple spaces.

For presentation purposes, and to make our design choices clearer, this section incrementally introduces the solution conceived for collective sort. An intermediate approach is discussed, along with some preliminary results that required a design improvement, ultimately leading to the solution fully evaluated in Section 4.

3.1. Basic strategy

Given the general architecture for collective sort, we start describing a basic strategy for the solution inspired by ants' brood sorting.

Similarly to ants, each sorting agent performs a partial system observation, namely, an observation on the local tuple space (where an item is possibly picked up) and an observation on some remote tuple space randomly chosen (where the item may be dropped). According to such observations, if it can be inferred that some tuple is better sent to a remote tuple space s, then the agent locally removes the tuple and inserts it in s. This observation–action cycle is executed by each agent with a fixed rate r, so that the global sorting rate is $N * r$—which is the number of moving attempts per time unit. This scenario is depicted in Fig. 3.

Therefore, each agent has the general goal of moving away tuples from its local tuple space whenever they are not forming a collection. In particular, the agent protocol we consider is as follows:

- `FIRE:` a remote tuple space R is drawn randomly;
- `LOCAL-OBS:` a uniform `read` operation is performed on local space L, yielding a tuple of kind K_L;
- `REMOTE-OBS:` a uniform `read` operation is performed on R, yielding a tuple of kind K_R;
- `MOVE:` if $K_L \neq K_R$ a tuple of kind K_R is moved from L (if any exists there) to R.

Uniform `read` operation, also called `urd`, is the operation explode by sorting agents to read any tuple from the tuple space—remember that any tuple has the same probability of being retrieved. If `urd` operation on a tuple space yields a tuple of kind K, it means that – probabilistically – tuples of kind K are those mostly occurring and hence, K becomes the best candidate for finally aggregating on that space. Accordingly, once task `REMOTE-OBS` is executed, the agent knows that space L is aggregating K_L while space R is aggregating K_R. The rationale of task `MOVE` is hence that if K_R and K_L are different, then the agent can fruitfully send a tuple of kind K_R from L to R, so that both K_R in R and K_L in L will correspondingly aggregate more.

The observation and then the decision taken by the agent are however affected by probability, hence the correctness of this distributed algorithm is to be checked by simulation, in order to verify, first of all, whether complete ordering is reached starting from any initial situation, and then, to evaluate its quality attributes.

3.2. Simulation method

As far as collective sort evaluation is concerned, system evolution can be modeled as a Continuous-Time Markov Chain (CTMC), namely a stochastic transition system where transitions are labeled with rates, representing the average frequency at which the transition can occur [9].[2] Let r be the rate of each sorting agent, the basic CTMC model of a collective sort

[2] If modeling time passing is not of interest, but one only cares about counting events, Discrete-Time Markov Chains could be used instead.

run initially selects the next sorting agent that fires, through N transitions labeled with rate r: this gives each sorting agent the same probability of being selected, while keeping the global sorting rate fixed to $N * r$. Then, the steps of the sorting agent's protocol are executed through transitions with very high rates (namely, those transitions should ideally occur instantaneously), but taking into account probability when executing `urd` operations—each matching tuple has same probability of being returned. This process is either executed t times, where t characterizes the duration of the simulation, or until convergence is reached ($H = 0$). The state of system configuration is basically a matrix $(q)_{ik}$ of natural numbers, where q_{ik} is the amount of tuples of kind k into space i.

There are many simulation tools that could be used to experiment with collective sort, all providing different language expressiveness, but of course yielding the same simulation results—examples include SPIM [33], SWARM [2] and REPAST [1]. Following the work in [9] we adopted a simulation engine written in the MAUDE term-rewriting system [11]. The main reasons for this choice are that MAUDE (i) allows – thanks to term-rewriting paradigm – to flexibly structure a system behavior as a typed operational semantics in Plotkin's style [34], (ii) executes transitions and computations with high performance thanks to advanced matching algorithms, (iii) is equipped with a full-fledged library for mathematical computations; and (iv) supports interaction with external tools—which could be built to control simulations and draw results. MAUDE allows one to set up the syntax of a system configuration in a flexible way by means of sorts and constructors (i.e., functions). For instance, the following code

```
 sort Tuple TupleMSet Space
DataSpace . op _[_] : Qid Nat -> Tuple [ctor] . subsort Tuple <
TupleMSet . op _|_ : TupleMSet TupleMSet -> TupleMSet [ctor assoc
comm] . op <_@_> : Nat TupleMSet -> Space [ctor] . subsort Space <
DataSpace . op _|_ : DataSpace DataSpace -> DataSpace [ctor assoc
comm]
```

defines the sort `Tuple` (`a[100]` represents 100 copies of tuple `a`), `TupleMSet` (multisets of tuples separated by associative and commutative composition operator "`|`"), `Space` (`<1@M>` is space `1` with multiset of tuples `M`), and `DataSpace` (a composition of spaces, again by operator "`|`"). As a result, an initial configuration where each tuple space has the same number of tuples of any kind, for instance $T = 100$ and $N = 4$, is described as:

```
< T1 @ (K1[25])|(K2[25])|(K3[25])|(K4[25]) > |
< T2 @ (K1[25])|(K2[25])|(K3[25])|(K4[25]) > |
< T3 @ (K1[25])|(K2[25])|(K3[25])|(K4[25]) > |
< T4 @ (K1[25])|(K2[25])|(K3[25])|(K4[25]) >
```

Transition rules are written in our framework as a unary postfixed function ==> associating a system state with the set of all possible target states, each with its own rate. As an example, equation

```
eq (init | DS)==> =       (0.25->[[0]|DS]); (0.25->[[1]|DS]);
                          (0.25->[[2]|DS]); (0.25->[[3]|DS]).
```

associates an initial state including the `init` term with four possible states (meaning a sorting agent has yet to be designated), where `init` is substituted with term `[i]` (`i` is the selected sorting agent), each labeled by rate 0.25 (global sorting rate is set to 1).[3] Once the entire transition system is defined, the simulation is executed by a MAUDE command of the kind:

```
rewrite < [ 5000 : ( SS ) @ 0.0 ] > .
```

which produces on standard output a trace of 5000 system states, starting from configuration SS and from time `0.0`: such a trace is then used to draw a chart showing the system evolution.

An example of a simulation trace starting from the above initial state is pictorially represented in Fig. 4(a), reporting the dynamics of the "winning" tuple in each tuple space—namely, the tuple that eventually aggregates there. Note that tuples reach their full aggregation level at different points in time, in an unpredictable way. The chart in Fig. 4(b) displays instead the evolution of tuple space `T1` taken as a reference: notice that only tuples of kind `K1` aggregate there despite the initial concentration of `K1` in `T1` being the same as other tuples. In particular, e.g., at some point around step 1000 there is a bifurcation which promotes aggregation of `K1` tuples instead of `K2`.

It is interesting to also analyze the trend of the entropy of each tuple space as a way to estimate the degree of order in the system through a single value: since the simulated strategy is to increase the inner order of the system, entropy is expected to decrease to zero, as actually shown in Fig. 4(c). Each chart reports the number of protocol instances (moving attempts) executed by agents: the chart shows that in this simulation full sorting is reached after around 3000 time units—i.e., 3000 executions of an agent protocol.

[3] The above code works only with $N = 4$ just to simplify presentation, but our specification deals with the general case.

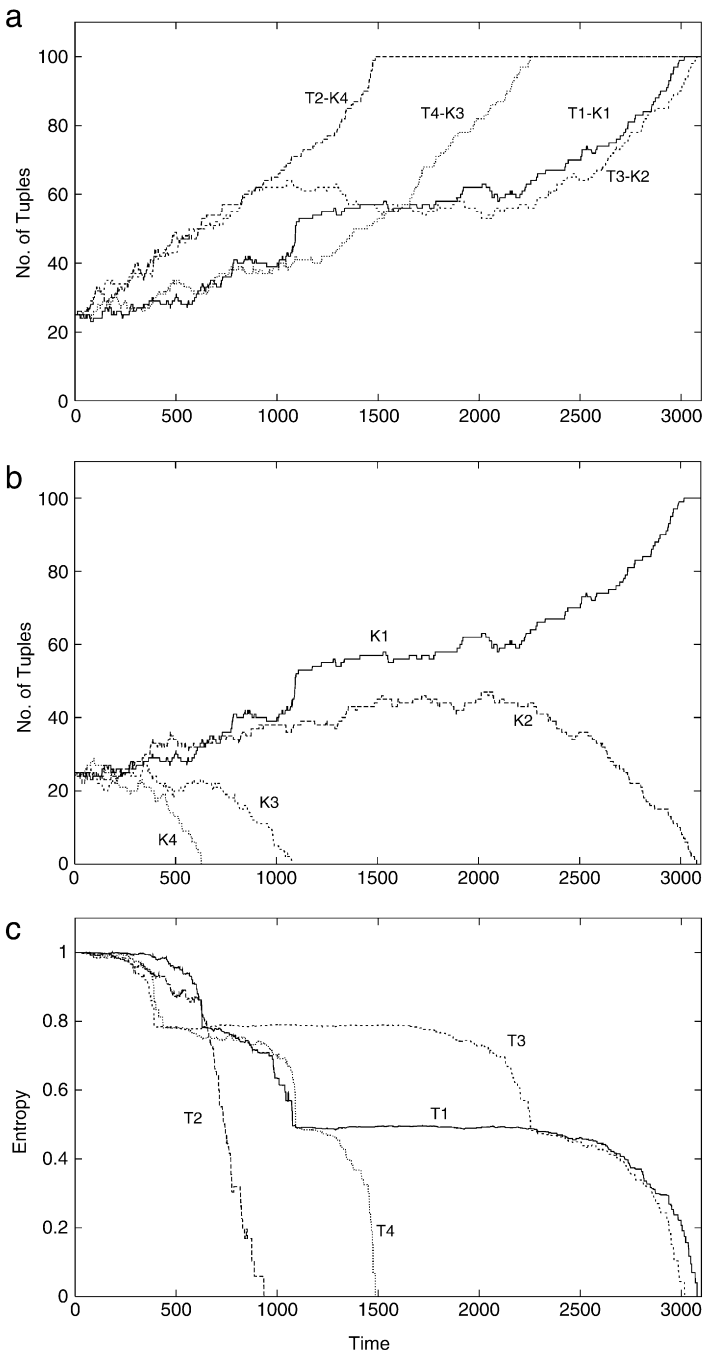


Fig. 4. Charts of a simulation trace: (a) Winning Tuple; (b) Tuple space T1; (c) Entropy in each tuple space (normalized).

3.3. On convergence

At a first glance, the solution developed so far appears to converge to complete sorting from any initial configuration of tuples. However, it is easily detectable that there are some stable states attracting the system trajectory and having positive entropy, that is, characterized by an incomplete degree of sorting. A state of this kind is called a *local minimum* (from the standpoint of entropy). An example of such a minimum is the following state, obtained by the traces shown in Fig. 5:

```
< T1 @ (K1[100])|(K2[0]) |(K3[0])  |(K4[0]) > |
< T2 @ (K1[0])  |(K2[69])|(K3[0])  |(K4[0]) > |
```

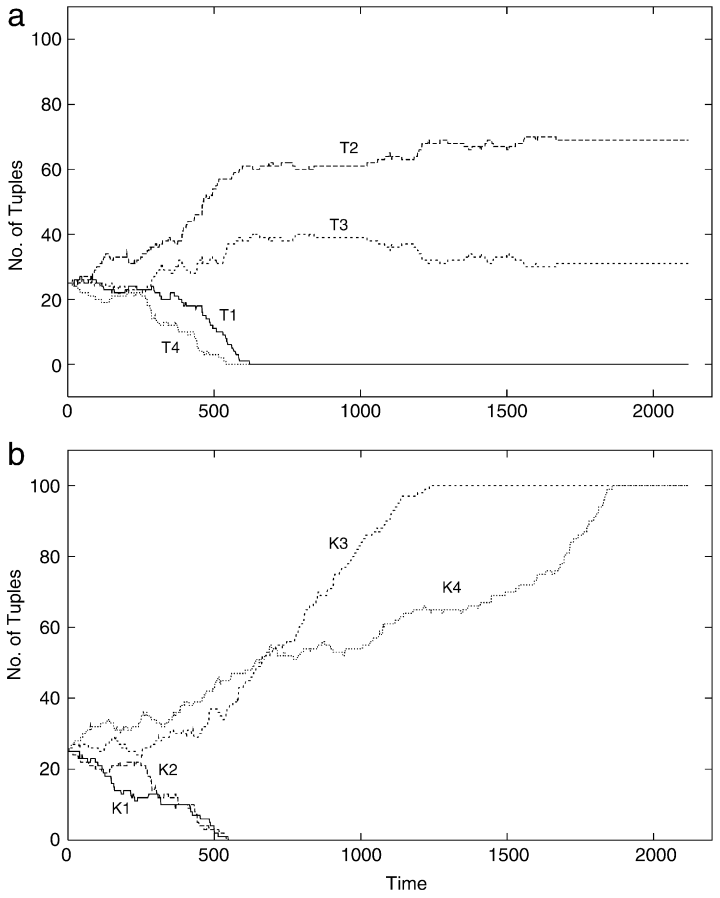


Fig. 5. Charts of a simulation trace to a local minimum: (a) Tuple kind K2 aggregating in spaces T2 and T3; (b) Both kinds K3 and K4 aggregating in space T4.

```
< T3 @ (K1[0])   |(K2[31])|(K3[0])   |(K4[0]) > |
< T4 @ (K1[0])   |(K2[0]) |(K3[100])|(K4[100]) >
```

Tuple kind K2 is the only one aggregating in both T2 and T3, and at the same time, both kinds K3 and K4 aggregate in space T4. It is easy to recognize that once this state is reached, no agent will ever move a tuple, since in no space a tuple is found that aggregates less than elsewhere. Our simulations show that: (i) about 5% of runs from the initial chaotic configuration with $N = 4$ ends in a local minimum; (ii) this probability increases with N, e.g., it is more than 15% when $N = 7$, since many more local minima exist; (iii) simulations from states that are sufficiently near to a local minimum always end up in it—local minima are attractors. This makes the approach discussed so far inadequate with respect to the quality attributes defined in previous section, and thus requires a suitable solution before proceeding with any further evaluation. The attempt of solving this problem is what led us to the solution actually proposed in this article, as discussed in the following.

The main reason that the local minimum analyzed above cannot be escaped lies in the fact that the strategy we developed does not explicitly avoid the case where the same tuple aggregates in two different tuple spaces. In fact, due to task MOVE in sorting agent protocol, nothing is done when $K_L = K_R$! Hence, it can happen that the same tuple kind fully aggregates on two different tuple spaces, and dually, two remaining tuple kinds aggregate in the same space as shown in the local minimum above.

These two issues can actually find a common solution by a more careful analysis of brood sorting in social insects. There, an ant picks an item and releases it where a new place is found with greater concentration, expressed as quantity of brood over a unit of space. That is, an ant is implicitly able to compare the amount of brood with a standard quantity, which in that specific case is represented by the amount of empty space. If a similar notion were defined in collective sort, that could in principle allow one to solve the two issues above. On the one hand, if space T3 could be recognized as having "less" tuples K2 than space T2, then movements from space T3 to T2 could be promoted more. Hence, as one of the two spaces stops aggregating tuples, some tuples K3 or K4 could be moved there from space T4.

3.4. System annealing by noise tuples

To implement a mechanism supporting this idea, we add to tuple spaces another kind of tuple `noise`, which – for simplicity – we initially suppose to have constant concentration (i.e. amount) in all spaces throughout sorting. Such tuples are not inserted and retrieved by user agents, and are not subject to sorting, but are managed (inserted/retrieved) by sorting agents only. Now, when a sorting agent performs a uniform read to randomly select a tuple, such an observation gets "perturbed", since there is a probability that the result is `noise`. Simply, if the tuple space holds, say, 95 regular tuples (those to be sorted) and 5 noise tuples, there is a 5% of probability of retrieving `noise`. Following the previous version of the algorithm, the new interaction protocol is such that a tuple is moved from the local space to the remote space if and only if the two observations are different—but now one of them could be `noise`. As an example, if the remote observation is `noise` and the local one provides kind `k`, then the locally observed tuple is moved anyway, even though this does not necessarily decrease entropy. That is, the role of `noise` tuples is to alter probabilistically the correctness of agent actions, which now may increase disorder when observations were perturbed with some noise. As a result, this mechanism causes tuples to be moved even though the system is in a local minimum, hopefully making the system trajectory escape from it. This technique actually resembles the concept of *simulated annealing* used in optimization algorithms [20]. There, a perturbation is added in order to avoid the risk of finding non-optimal solutions: such a perturbation is initially high and is made to fade continuously as the system searches solutions, until completely disappearing.

In our case, the occurrence of `noise` tuples models such a perturbation: what should be the dynamics of noise through time, then? A feasible approach would be to set an initial amount of noise equal in all tuple spaces, and either leave it unaltered during system life-cycle or decrease it at a fixed rate. However, this choice would require to set the noise amount at design-time, but then optimality of this amount would depend on the average occupation of tuple spaces during system execution [38]: this situation is not appealing since we want our approach to work independently of the number of tuples in the system. What we actually look for is a fully adaptive noise mechanism, where noise is initially very low, and it increases as the system approaches to a local minimum, and decreases when such a minimum is escaped. In this way, we could expect the system performance to be only slightly affected if the system stays sufficiently far from local minima; on the other hand, noise production may become significant only in unfortunate cases where local minima are approached.

To achieve this result, we will manage noise as follows:

- As already described, noise alters observations performed by uniform read operations, and hence the pertinence of tuple movement, since we still move a tuple if the local and remote observations are different—though one of them could be noise.
- Initially only one noise tuple occurs in each tuple space, hence, perturbation is very low; as a result, sorting performance is not significantly affected.
- Each time two tuple spaces seem to aggregate the same tuple – two equivalent non-noise observations are made – noise is increased; in fact, that would mean we are likely approaching a local minimum, hence we have to increase system annealing.
- When some tuple is transferred due to pertinent observations – two different non-noise observations are made – noise is decreased; in fact, that would mean we are likely escaping a local minimum, hence we have to decrease system annealing.

Accordingly, the agent protocol is changed as follows:

- `FIRE:` a remote tuple space R is drawn randomly;
- `LOCAL-OBS:` a uniform `read` operation is performed on local space L, yielding a tuple of kind K_L;
- `REMOTE-OBS:` a uniform `read` operation is performed on R, yielding a tuple of kind K_R;
- `MOVE:` if $K_L \neq K_R$ a tuple is moved from L to R, i.e.:
  - if $K_R =$ `noise`, such a tuple has to be of kind K_L;
  - otherwise, such a tuple has to be of kind K_R (if existing in L);
- `NOISE:` if $K_L \neq$ `noise` and $K_R \neq$ `noise` local noise amount is changed, i.e.:
  - if $K_R = K_L$ then noise is increased by one in L;
  - if $K_R \neq K_L$ then noise is decreased by one in L.

Now both K_L and K_R could be noise. Task `MOVE` says that differences in observations in L and R should always cause a transfer: if K_R is not noise, a K_R tuple is moved to R, otherwise, a K_L tuple is moved to R. Task `NOISE` increases noise when L and R are aggregating the same (non-noise) tuple $K_R = K_L$, and decreases noise whenever a non-perturbed transfer is actually executed.

Considering now the worst case of a symmetric local minimum:

```
< T1 @ (K1[100])|(K2[100])|(K3[0]) |(K4[0]) > |
< T2 @ (K1[0])  |(K2[0])  |(K3[50])|(K4[0]) > |
< T3 @ (K1[0])  |(K2[01]) |(K3[50])|(K4[0]) > |
< T4 @ (K1[0])  |(K2[0])  |(K3[0]) |(K4[100]) >
```

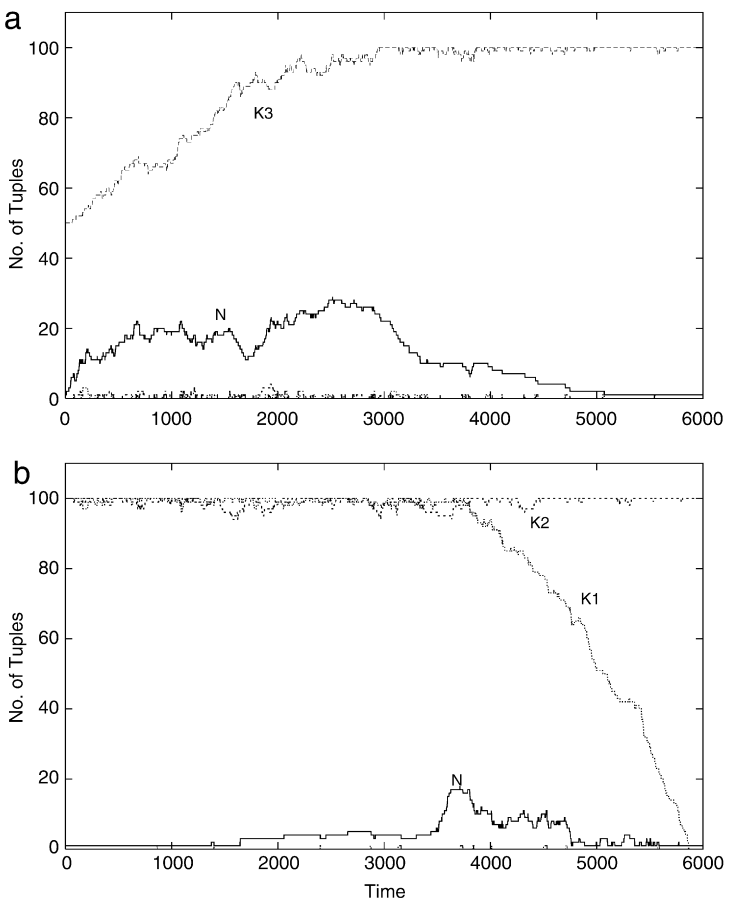


Fig. 6. Charts of a simulation trace escaping from a local minimum: (a) Situation in space T2: winning tuple K3 and noise in evidence; (b) Situation in space T1: kind K1 leaves the space.

we expect that noise starts increasing in both tuple spaces T2 and T3 (NOISE). At some point, a movement of tuples K3 will occur between T2 and T3 since some noise is observed (MOVE). Because of a bifurcation effect, if either space T2 or T3 has a greater concentration of tuples K3 with respect to noise, that would cause more tuples to be transferred there, and that space will eventually fully aggregate tuples K3. Accordingly, the other tuple space will be emptied, it will lose noise tuples, and it will finally become target of tuples of kind K1 and/or K2. This is actually what can be observed from the traces in Fig. 6(a) and (b), showing how the local minimum is escaped in spaces T2 and T1: in both cases we see that as noise tuples increase, the system escapes the local minimum configuration, and after that, noise tuples fade.

More simulations performed to evaluate this solution actually show that: (i) using noise slightly affects performance, for typically systems stay away from local minima and generate little noise; (ii) starting from a local minimum, the system is always able to escape it; and (iii) full sorting is always eventually reached. This solution hence appears to be a promising one, so we proceed in next section in more fully evaluating it.

4. Evaluation

According to the quality attributes described in Section 2.4, concerning convergence, scalability, and reactiveness, we here evaluate the proposed approach. To this end, standard analysis techniques for distributed systems usually include automatic machine checking or hand-written proofs.

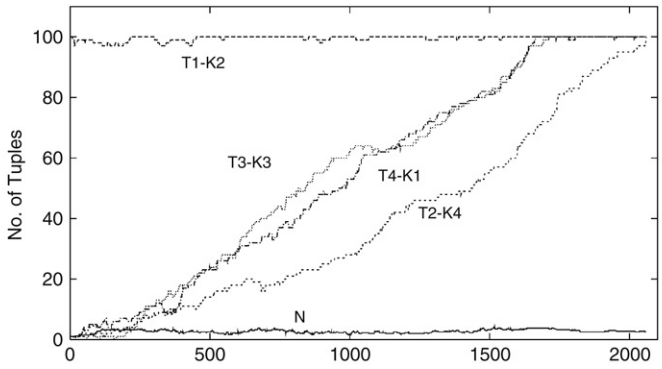
In the first case, analysis would be obtained e.g. by a probabilistic model checking technique [22,36]. There, the graph of all possible states and transitions is constructed and annotated with probabilities and rates, so that queries in a probabilistic logic like PTL can be checked by navigating the entire graph: e.g., for collective sort we could ask what is the probability that from an initial state with $N = 4$, $T = 100$, $H = 1$ we reach an ordered state ($H = 0$) within 3000 time units. However, this technique suffers from state-space explosion problem, so that even the construction of the entire graph is a very expensive operation—finding proper optimizations and developing this idea are however interesting subjects for future work.

In the second case, we should try to find proofs that the proposed algorithm converges, and in general, that the quality requirements are met. However, this is particularly complex, and very few examples exist in the literature that apply this

a

i	T1	T2	T3	T4
K1	100	0	0	0
K2	100	0	0	0
K3	100	0	0	0
K4	100	0	0	0

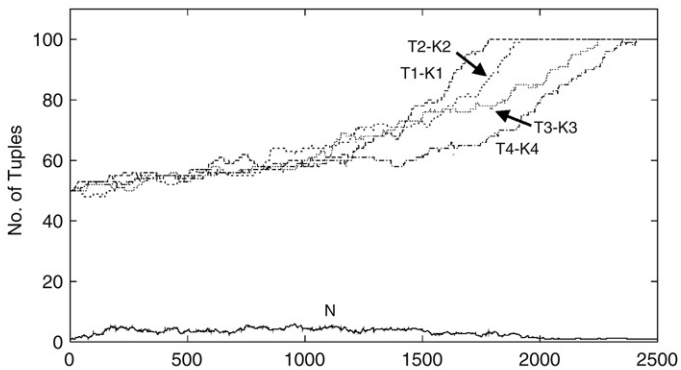
f	T1	T2	T3	T4
K1	0	0	0	100
K2	100	0	0	0
K3	0	0	100	0
K4	0	100	0	0



b

i	T1	T2	T3	T4
K1	50	50	0	0
K2	50	50	0	0
K3	0	0	50	50
K4	0	0	50	50

f	T1	T2	T3	T4
K1	100	0	0	0
K2	0	100	0	0
K3	0	0	100	0
K4	0	0	0	100



c

i	T1	T2	T3	T4
K1	25	25	25	25
K2	25	25	25	25
K3	25	25	25	25
K4	25	25	25	25

f	T1	T2	T3	T4
K1	0	0	0	100
K2	100	0	0	0
K3	0	0	100	0
K4	0	100	0	0

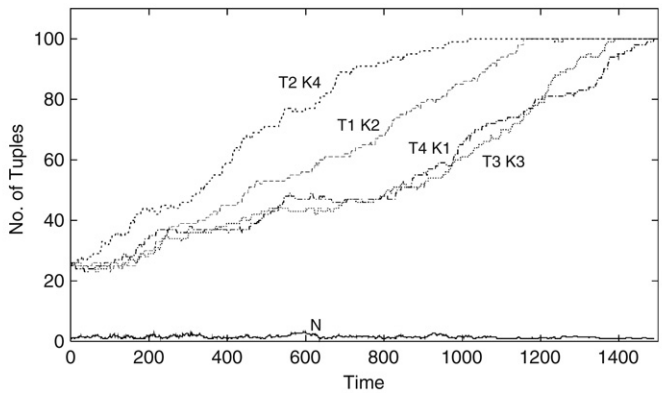


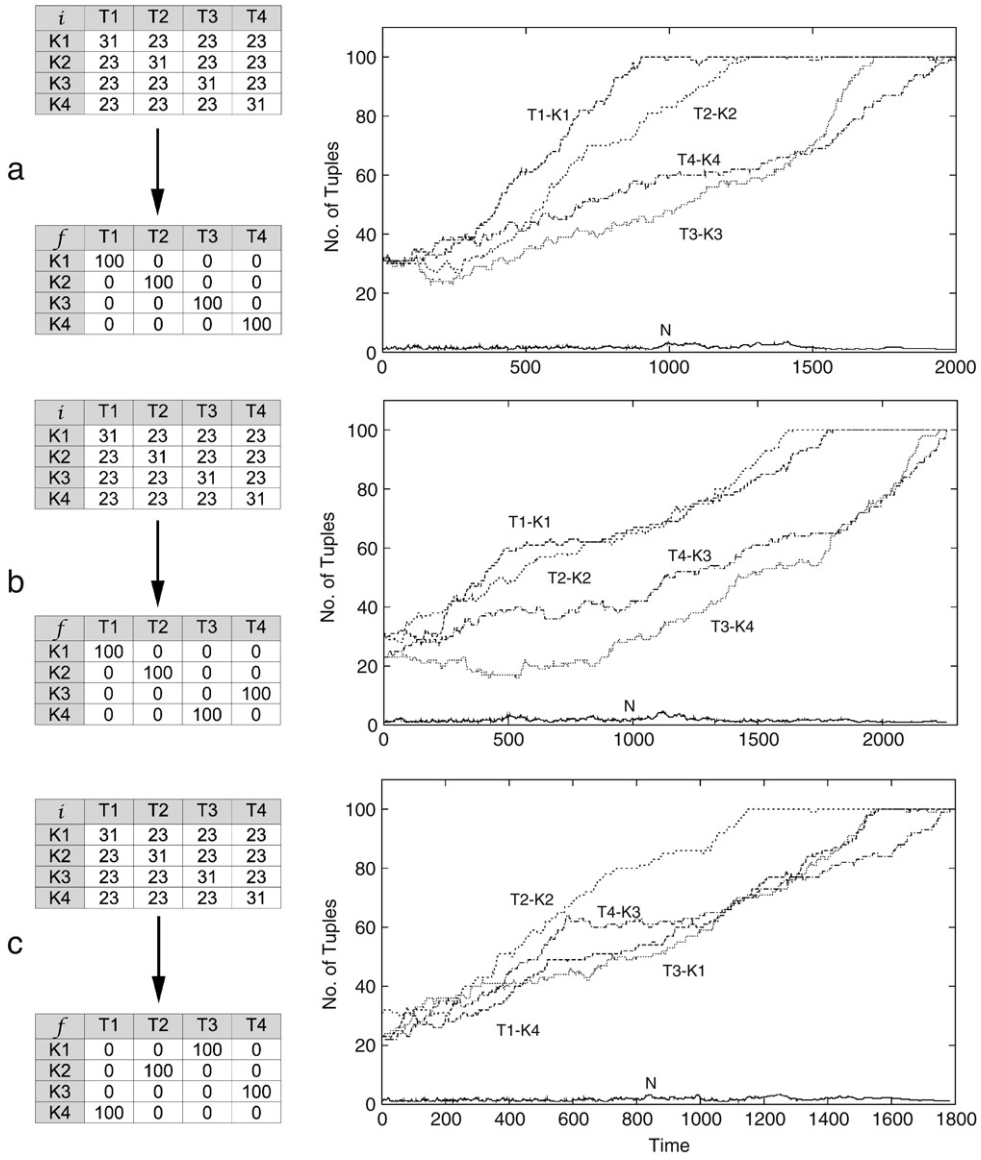
Fig. 7. Sorting from different configurations: winning tuples and noise.

approach to self-organizing systems. A recent exception is the work in [39], which focusses on a very straightforward self-organization scenario, in which a chain of cellular-like components is subject to a repeated applications of local transformation rules. Necessary and sufficient conditions are provided for the emergence of a certain global pattern of cellular states—similarly to what happens in certain stages of embryogenesis. This research direction, although very interesting, is still in its infancy and cannot be applied to generally analyze systems like ours.

On the other hand, computer-based simulation techniques have gained growing attention over the past years as an important tool for studying complex systems [40,4], especially when previous analytic results are not available [40]—e.g. due to analytical intractability of the target system. Accordingly, simulation is the approach we adopt for evaluating collective sort.

4.1. Full convergence

In our solution, on the one hand, non-perturbed tuple movements cause entropy to decrease; on the other hand, the noise mechanism successfully perturbs those configurations that approach local minima. Fig. 7 shows examples of how the system can come to full sorting starting from different initial configurations—these are taken as samples of several simulations we ran, all of which reached full sorting. Interestingly, we can observe that in these cases – differently from the one shown in



i	T1	T2	T3	T4
K1	31	23	23	23
K2	23	31	23	23
K3	23	23	31	23
K4	23	23	23	31

f	T1	T2	T3	T4
K1	100	0	0	0
K2	0	100	0	0
K3	0	0	100	0
K4	0	0	0	100

i	T1	T2	T3	T4
K1	31	23	23	23
K2	23	31	23	23
K3	23	23	31	23
K4	23	23	23	31

f	T1	T2	T3	T4
K1	100	0	0	0
K2	0	100	0	0
K3	0	0	0	100
K4	0	0	100	0

i	T1	T2	T3	T4
K1	31	23	23	23
K2	23	31	23	23
K3	23	23	31	23
K4	23	23	23	31

f	T1	T2	T3	T4
K1	0	0	100	0
K2	0	100	0	0
K3	0	0	0	100
K4	100	0	0	0

Fig. 8. Different sorting result from the same configuration.

Fig. 6 – total noise is typically very low: this is mainly because here the system stays sufficiently far from local minima. In all the three cases, the system dynamics is highly probabilistic, and hardly predictable.

Moreover, it is typically the case that the same system shows different dynamics in different runs. As another illustrative example, we consider the starting configuration shown in Fig. 8, which apparently seems to always lead to the full sorted configuration in chart (a). Many cases – about 20% of the times – show that the system actually converges to a different final state, where one or more kinds actually aggregate in spaces with an initial smaller concentration than other kinds. This, again, highlights the true unpredictable character of our self-organizing solution to collective sort.

As a further measure for evaluating collective sort convergence, we considered the variability of sorting time throughout a large series of 1000 simulations with $N = 4$ and $T = 400$ tuples per kind. Fig. 9 shows how sorting time distributes over the executed simulations. It is easy to recognize a peak centered around 2500 time units, meaning that most of the simulation runs ($\sim$220) led to a sorting time value close to 2500 time units, while 92.2% of the simulation runs led to a sorting time $\leq$ 5000 time units. In addition, a few percentage of simulations led to a high value of convergence time (up to 40 000 time units): this is due to evolutions of collective sort that approached one or iteratively more states of a local minimum, which usually require a much higher amount of time to achieve full sorting due to the perturbation effect generated by noise tuples. This chart shows the influence of noise on convergence: most of the times it does not affect convergence time, while in other cases where sorting approaches a local minimum, noise starts working ultimately leading to convergence.

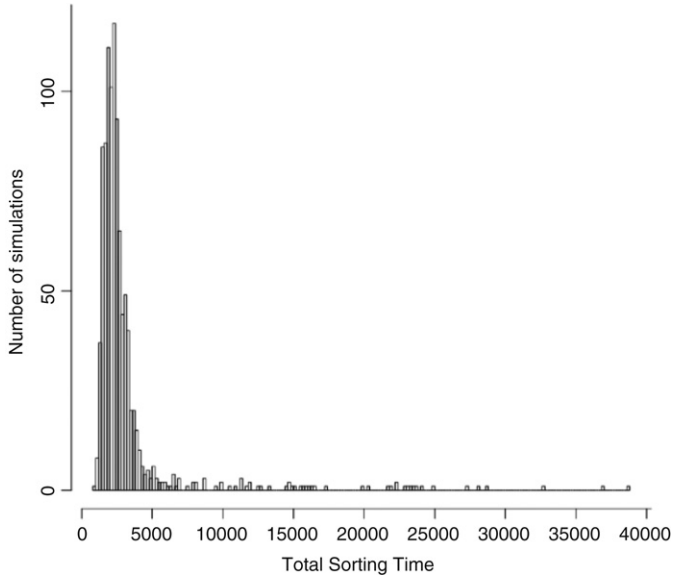


Fig. 9. Total-sorting-time distribution over 1000 simulation runs executed with $N = 4$ and 400 tuple per kind.

4.2. Sorting cost and scalability

Concerning performance, it should be noted from previous charts that the average time for reaching full sorting is around 2500 time units, considering the basic case with 4 tuple spaces, 4 tuple kinds, and an initial set of 400 tuples. The global sorting rate considered is 1.0, that is, there is an average of one transfer attempt per time unit, and accordingly a 0.25 sorting agent rate.

The optimal solution to the problem – in which a snapshot of the system is taken and agents are accordingly programmed as how to move tuples – would require instead around 300 time units (see Section 2.4), which is the time necessary to move the tuples that are not already in the right space. Namely, in a basic case of the problem, the system performance is around one order of magnitude worse than the optimal solution: this is still within the expected range of the quality requirements sought for the proposed solution.

It is now interesting to see how much our solution scales with the dimension of the problem. First of all, we consider scalability in the number of tuples, that is, how the size of the set of tuples affects the system behavior. Fig. 10 shows average performance values (taken from a set of 20 runs), on a system with $N = 4$, an initially chaotic configuration, and an increasing number of tuples from 80 to 8000. Two values are depicted: tuple transfers and sorting time. The first one is simply increased each time a tuple is moved, the second one is instead the elapsed time units—or moving attempts. Another interesting value is network usage (or number of remote interactions), which can be directly computed as the sum of tuple transfers and elapsed time—since other than tuple transfers, we have a remote observation per time unit. As a main result of these charts (i) the proposed approach scales linearly with the number of tuples, and (ii) the number of transfers is around 1/3 of the number of moving attempts—this parameter somehow representing the pertinence of observations.

A simulation trace showing the evolution of collective sort with $N = 4$ and 4000 tuples per kind is reported in Fig. 11. It is easy to see that the evolution of the simulation is not greatly affected by the higher tuple concentration, the only difference is that system evolution appears much more deterministic than e.g. in Fig. 7(c). In addition, total noise concentration keeps values which are 2 orders of magnitude lower than the total tuple concentration as in experiments with lower tuple concentrations. This simulation shows that non-determinism, as well as the need of relying on noise, actually decreases as the system's size becomes larger and larger: hence, unexpected situations are more likely to occur on smaller-scale systems.

Scalability in N, the number of tuple spaces (and kinds), is a more critical issue. Fig. 12 shows the average performance values (taken from a set of 20 runs), on a system with a fixed number of tuples equal to 400, an initially chaotic configuration, a fixed sorting agent rate equal to 0.25, and an increasing number of tuple spaces, from 3 to 15—the global sorting rate is $0.25 * N$. As in the previous case, we measured the number of tuple transfers and elapsed time units. Here we notice that the proposed solution scales with more difficulty, though the result appears reasonable anyway: while the average time to full sorting is 2500 with $N = 4$, it is about 30 000 with $N = 10$—the latter being indeed a more complex problem!

The sorting cost actually appears to be quadratic in the number of tuple spaces. This result seems however a key characteristic of collective sort, rather than depending on the solution we propose in this article—this is a problem even in array sorting, which does not scale linearly. In general, if some tuple remains far from where its kind is aggregating, the time (and network operations) needed by its agent to find the proper remote tuple space is linear in N, and this applies to all sorting agents/kinds. This appears to be an intrinsic consequence of the fact that a global status is not available to sorting agents, and observations are necessarily pointwise.

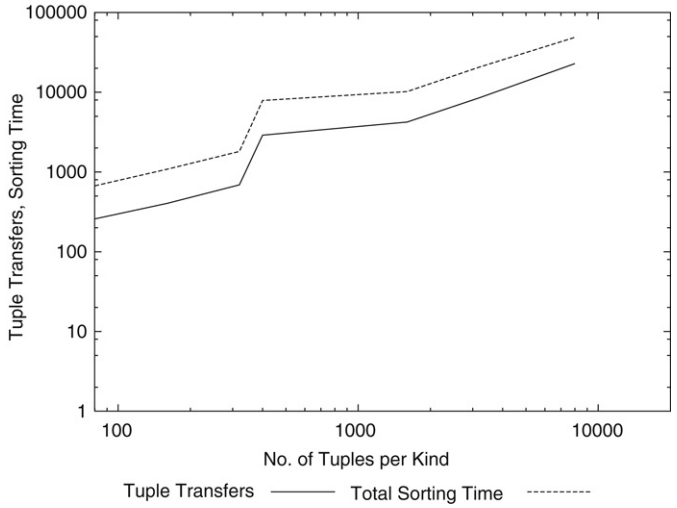


Fig. 10. Scalability in the number of tuples ($N = 4$): transfers and elapsed time units.

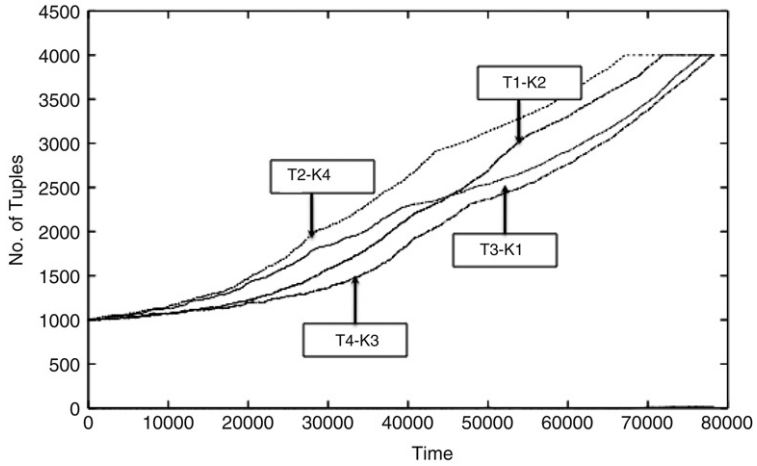


Fig. 11. Winning tuple (a) and noise (b) evolution for a simulation with $N = 4$ and 4000 tuples per kind initially distributed in a uniform way.

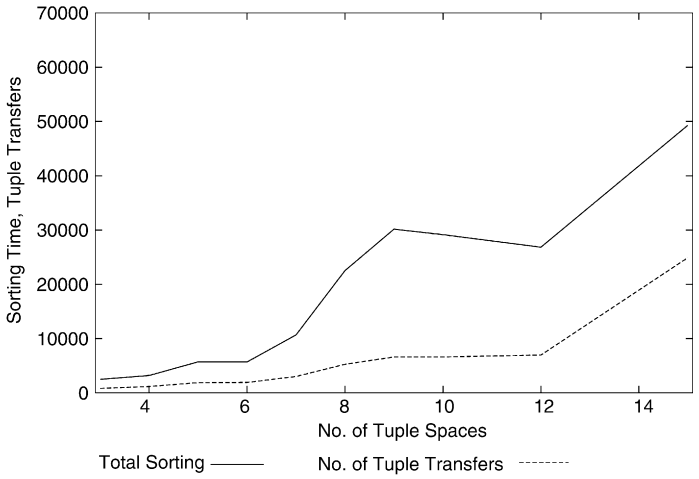


Fig. 12. Scalability in N (n. of tuple spaces/kinds): transfers and elapsed time units.

In addition to the results shown in Figs. 12 and 13 reports a simulation trace executed with $N = 8$ and 400 tuples uniformly distributed among the N available kinds. Again, noise keeps values which are 2 orders of magnitude lower than tuple concentration even though collective sort operates on a larger set of tuple spaces.

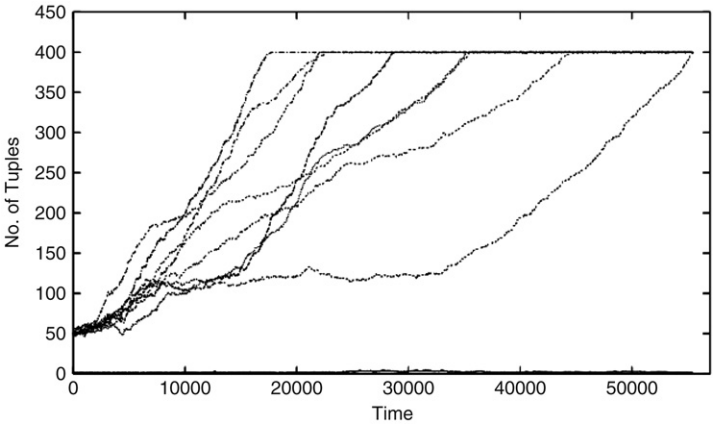


Fig. 13. Winning tuple (a) and noise (b) evolution for a simulation with $N = 8$ and 400 tuples per kind initially distributed in a uniform way.

To summarize, even considering scalability, the proposed approach appears to successfully meet the desired requirements.

4.3. Reactiveness

The main reason that collective sort has been solved by using a self-organizing approach is to tackle unpredictable interactions with the environment. A typical usage scenario includes user agents that exploit the coordination service provided by the tuple spaces, that is, they keep inserting and removing tuples. The details of this behavior cannot be fully known a priori, hence, sorting should be able to react to changes of the surrounding conditions, in an adaptive way. Providing a comprehensive and general set of simulations in this context is not easy, since user agents can manifest an extremely wide range of different interactive behaviors. This section shows how the ratio between user agent rate and sorting agent rate, called *mutation/sorting ratio*, influences the result of sorting. To this end, we keep the global sorting rate fixed to 1.0 and include in the simulation a *mutation rate* for user agents, that is, at that rate a user agent randomly moves a tuple from one space to another. Starting from an initially sorted configuration of tuples (400 tuples and $N = 4$, with $H = 0$), depending on mutation rate we easily expect that (i) full sorting is almost always maintained, (ii) a certain level of (partial) sorting can be maintained, or (iii) the system becomes more and more unsorted as time passes. Evolution through such situations is reported in Fig. 14, where each chart provides the evolution of entropy over time for different mutation rates.

As shown in the summary Fig. 15, the key factor is the mutation/sorting ratio, which gives a clear indication of the adequacy of sorting resources, in terms of the maximum level of entropy they can guarantee—a mutation/sorting ratio smaller than 0.5 seems to be a reasonable trade-off between required sorting resources and achieved sorting level. It is clear that a form of load-balancing is required to be sure that sorting resources are adequate with respect to the current degree of disorder, and can self-adapt to it—increasing on a by-need basis and then decrease. Techniques related to prey–predator approach as studied e.g. in [16,15] could be evaluated in future research.

5. Related works

5.1. Self-organization in social insects

Initially developed in chemistry and physics, *self-organization theory* describes the *emergence* of macroscopic patterns generated from microscopic interactions of a system's components. In this theory, individual complexity is not excluded, but at some level of description it is possible to provide an explanation of a complex behavior in terms of simple entities [5]. This point is also stressed in our work, due to the choice of showing that collective sort can be solved by relying on very simple agents as well.

Self-organization theory encompasses biological systems, in particular, it is suitable to describe the dynamics observed in insect colonies. Social insects are well known for their ability to collectively solve problems despite the individual limited perception and action capabilities. The wide repertoire of behaviors exhibited by insect colonies ranges from cooperative transport to complex structure building [5,8]. In particular, sorting and clustering phenomena are observed in various forms and across several insect species. Clustering involves gathering items scattered into the environment and organizing them in piles [12,5], while sorting involves more complex patterns and shapes like concentric rings [14]. As an example of clustering, consider pile formation in termite colonies: termites wander in the environment, pick up scattered wood chips and arrange them in clusters. Resnick proposed a simple model able to recreate this collective dynamics [35]: individual termite behavior is specified by two simple rules: (i) if nothing is carried, pick up an item as one is encountered; (ii) if carrying an item, drop

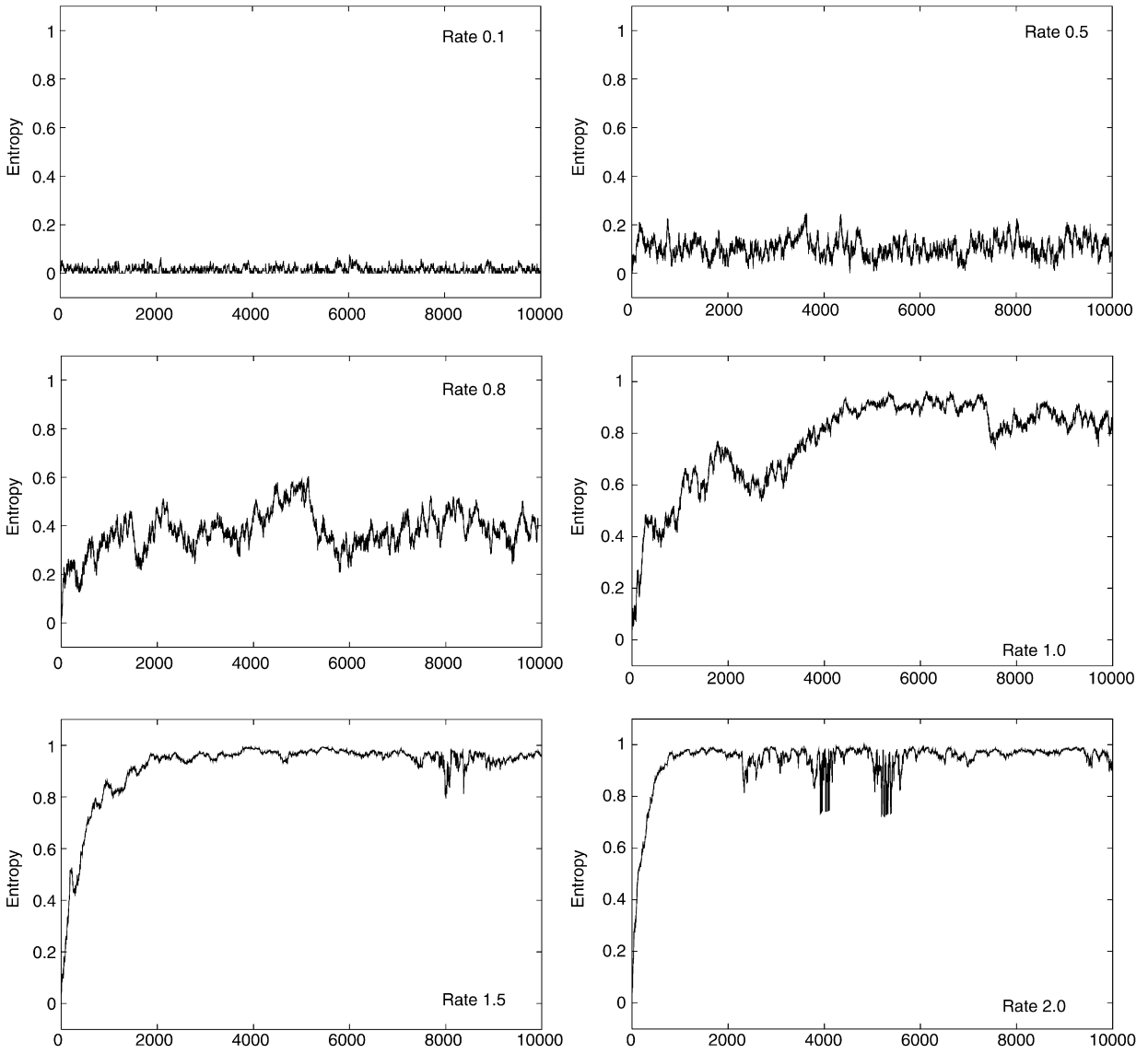


Fig. 14. Evolution of entropy with different perturbation/sorting ratio.

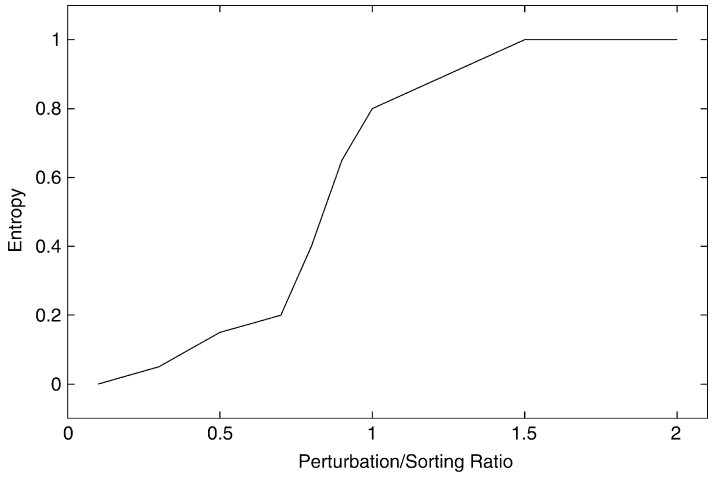


Fig. 15. Maximum entropy depending on perturbation/sorting ratio.

it when encountering another one. Observations across several ant species reported that corpses are clustered in small cemeteries in order to clean up nests [5]. Similarly, broods of the same size are clustered while broods of different size are sorted into concentric annuli. Concentric patterns have been observed in hive organization by honeybees: the central area is occupied by broods and surrounded by pollen and honey placed in concentric rings [7,8].

Although the actual mechanisms regulating these behaviors are not fully understood yet, researchers agree on the fundamental role of local density perception. In the literature there are several models able to replicate the observed dynamics: specifically, collective robotics have produced several notable results through experimentation with physical robots.

5.2. Collective robotics

Collective robotics is characterized by scenarios involving multiple autonomous robots coordinating with one another by using local sensing and actions as well as limited communication. In the last decade, collective robotics has probably been the most active field involved in developing artificial systems inspired by social insects' behavior. For instance, the main goal of the European project SWARM-BOTS has been to study novel approaches for developing self-organizing and self-assembling swarms of robots [13]. However, among the several scenarios investigated in that specific context, none is strictly related to sorting behaviors.

Instead, other works in collective robotics deal with the problem called *spatial sorting* which is closely related to collective sort but solved by mechanisms more similar to what observed in social insects. A taxonomy of spatial sorting problems reflecting actual insect behaviors is provided in [26]:

clustering—the only available type of item is grouped in a small fraction of the available space;
segregation—each type of items is clustered but contiguous to other clusters;
patch sorting—each type of items is clustered but distant from other clusters;
annular sorting—clusters of different types of items are arranged in concentric rings.

Sorting tuple spaces in the general case, where the number of tuples nt and of tuple kinds nk may be different, spans different problems:

- when $nk = 1$ we have clustering;
- $nk > 1$ and $nk < nt$ resembles patch sorting;
- $nk > 1$ and $nk \geq nt$ resembles segregation;

Annular sorting requires a specific notion of topology we do not currently handle, but would be an interesting approach to consider as well. The choice of focussing on the case $nk = nt$ in collective sort is motivated by the need of optimizing tuple searching by user agents, which is in fact very easy in this case due to the fact that entropy can decrease to zero.

In [12], Deneubourg et al. proposed a probabilistic model for brood clustering. Probabilities P_p and P_d as described in Section 2.2 depend on estimating the local density f of items: in [12] f is evaluated as the number of items encountered in time interval T. While researchers agree on the fact that ants actually perceive local density when picking up an item, whether they do so when dropping an item is still matter of debate [27]. In the struggle to find a minimal set of rules, as pointed out in [26], it is possible to achieve spatial sorting by exploiting techniques similar to self-sorting, i.e. sorting that occurs under environmental forces like gravity. In [23] a generalization based on a similarity function for exploratory data analysis has been proposed [5]. The Lumer Faieta's function replaces f: basically, such a function minimizes intra-cluster distance with respect to inter-cluster distance. Results coming from this research could be applied to evaluate an improved version of our solution to collective sort.

As an aside, it should be noted that collective sort is only apparently related to *data clustering* as defined in [19], which is indeed a different problem than spatial clustering of robotics. Data clustering defines techniques for associating patterns to group of related data items. As mentioned in Section 3 instead, data clustering could be applied at design-time to the set of tuples an application has to handle, in order to identify N clusters that will be used as kinds to be grouped.

5.3. Swarm based coordination

In nature, swarms are famous due to their ability to coordinate movements and tasks: typical examples include schools of fish, flocks of birds, and social insects in general [5,8]. Since it is possible to capture their behavior and encode it by simple rules, swarm principles are currently being investigated that could be applied to the coordination of artificial large-scale systems to tackle robustness and scalability issues. Notable examples of coordination media and infrastructures following swarm principles include SwarmLinda [29] and TOTA [24].

SwarmLinda applies algorithms inspired by ant colonies to LINDA-like distributed tuple spaces in order to tackle scalability and adaptability. On the one hand, a specific algorithm for searching tuples is included. This algorithm mimics the behavior of food foraging [8] where tuple templates resemble ants and each template is associated to a scent, i.e., a pheromone. Then, an algorithm for tuple distribution is defined that resembles brood sorting. When a tuple is inserted, it

starts visiting tuple spaces and is dropped when location hosting an aggregation of similar tuples is visited. Furthermore, SwarmLinda envisions the use of other techniques e.g. for migrating clusters of tuples and optimize the movement of tuples across tuple spaces [28]. SwarmLinda is an approach tightly related to collective sort: the main difference is that we move tuples by using a service external to tuple spaces, and seek complete sorting, whereas SwarmLinda is mainly targeted at diffusing tuples as much as possible, avoiding e.g. over-clustering.

TOTA (Tuples On The Air) is a middleware that allows coordination of agents in mobile ad-hoc networks. TOTA provides an active environment that allows one to quickly develop applications based on swarm-like coordination strategies, and in particular, following the principles of stigmergy. The environment is articulated in a dynamic network of peer-to-peer TOTA nodes partially connected. The key features [25] of TOTA include: (i) storage, propagation and maintenance of tuples; (ii) definition of tuples and their maintenance rules at the application level; and (iii) a simple API for injecting and retrieving tuples. In particular, when injected into a space, a copy of a tuple is spread in neighboring nodes recursively, and then eventually fades, creating a so-called computational field – like e.g. gravity in physics – which mobile agents can navigate to retrieve each other, or data of interest.

The collective sort problem for tuple spaces, along with some prototype solution, has been presented in some previous work of ours [9,38]: this article consolidates that research, proposes a new and improved solution, and provides a comprehensive evaluation and discussing a wide set of simulation results. Differently from [9], here sorting agents do not draw a candidate tuple kind to move before observing tuple spaces, but rather infer it from local and remote observations: this allows one to sensibly speed up convergence to full sorting, as tuples that are not aggregating are now more easily found. The mechanism of `noise` tuples – actually called `vacuum` tuples in [38], due to a different interpretation – has been accordingly adapted with respect to the one presented in [38]: this resulted into a mechanism that efficiently solves the local minimum issue, though better approaches could be investigated in future work.

6. Conclusion and future work

In order to keep tuples organized in a distributed system, we conceived the collective sort problem for distributed tuple spaces, aimed at grouping similar tuples in the same tuple space, for fast retrieval based on previous experience. A self-organizing solution to the problem has been proposed, inspired from ants' brood sorting, which stresses the possibility of providing full convergence to sorting even with very simple agents. The approach has been validated through a number of stochastic simulations of system evolution, providing evidence of full convergence, scalability, and reactiveness to unpredictable changes in the environment.

Future work of this research includes evaluating new solutions, which could be able to stress different aspects of the problem: First of all, the current solution can be finely tuned in several ways to improve performance, by: (i) dynamically increasing the number of tuples moved at each step by sorting agents (i.e. reducing observation rate), (ii) improving the noise mechanism in order to increase the current pertinence of moving, and (iii) considering load balancing issues, so that the rate of sorting agents can be dynamically set so as to adapt to the degree of sorting in the local space. Then, the collective sort scenario can be extended to a more general case where the number of tuples is not equal to the number of kinds. In particular, when the number of spaces increases, it is more reasonable to shift to non-flat networks of spaces – e.g., large scale-free networks – and accordingly consider a smaller set of kinds. There, an interesting sorting pattern would be to aggregate tuple kinds in one or more clusters spread over the network. Finally, it would be interesting to consider sorting based on a continuous similarity function between tuples, as considered e.g. in SwarmLinda [29].

References

[1] Recursive porous agent simulation toolkit (repast), 2006. Available online at: http://repast.sourceforge.net/.
[2] Swarm, 2006. Available online at: http://www.swarm.org/.
[3] M. Balazinska, A. Deshpande, M. Franklin, P. Gibbons, J. Gray, S. Nath, M. Hansen, M. Liebhold, A. Szalay, V. Tao, Data management in the worldwide sensor web, Pervasive Computing, IEEE 6 (2) (2007) 30–40.
[4] S. Balqies, Applied system simulation: A review study, Information Sciences 124 (1–4) (2000) 173–192.
[5] E. Bonabeau, M. Dorigo, G. Theraulaz, Swarm Intelligence: From Natural to Artificial Systems, Santa Fe Institute Studies in the Sciences of Complexity, Oxford University Press, 198 Madison Avenue, New York, NY 10016, USA, 1999.
[6] E. Bonabeau, M. Dorigo, G. Theraulaz, Swarm Intelligence: From Natural to Artificial Systems, Santa Fe Institute Studies in the Sciences of Complexity, Oxford University Press, Inc., 1999.
[7] S. Camazine, Self-organizing pattern formation on the combs of honey bee colonies, Behavioral Ecology and Sociobiology 28 (1) (1991) 61–76.
[8] S. Camazine, J.-L. Deneubourg, N.R. Franks, J. Sneyd, G. Theraulaz, E. Bonabeau, Self-Organization in Biological Systems, Princeton Studies in Complexity, Princeton University Press, 41 William Street, Princeton, NJ 08540, USA, 2001.
[9] M. Casadei, L. Gardelli, M. Viroli, Simulating emergent properties of coordination in Maude: The collective sort case, in: C. Carlos, M. Viroli (Eds.), Proceedings of the 5th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2006, in: Electronic Notes in Theoretical Computer Science, vol. 175(2), Elsevier Science B.V., 2007.
[10] P. Ciciriello, L. Mottola, G.P. Picco, Efficient routing from multiple sources to multiple sinks in wireless sensor networks, in: Wireless Sensor Networks, in: LNCS (Tutorial Volume), vol. 4486, Springer, 2007, pp. 34–50. URL: http://dx.doi.org/10.1007/978-3-540-69830-2_3.
[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, Maude Manual, Department of Computer Science University of Illinois at Urbana-Champaign, version 2.2 ed., version 2.2 is available online at: http://maude.cs.uiuc.edu, December 2005.
[12] J. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, L. Chrétien, The dynamics of collective sorting: Robot-like ants and ant-like robots, in: J.-A. Meyer, S.W. Wilson (Eds.), From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, MIT Press, Cambridge, MA 02142, USA, 1991, pp. 356–363.

[13] M. Dorigo, E. Tuci, F. Mondada, S. Nolfi, J.-L. Deneubourg, D. Floreano, L.M. Gambardella, The SWARM-BOTS project, Künstliche Intelligenz 4/05 (2005) 32–35. Also available at: http://www.swarm-bots.org as IRIDIA Technical Report No. TR/IRIDIA/2005-018. URL: http://www.swarm-bots.org.
[14] N.R. Franks, A.B. Sendova-Franks, Brood sorting by ants: Distributing the workload over the work-surface, Behavioral Ecology and Sociobiology 30 (2) (1992) 109–123.
[15] L. Gardelli, M. Viroli, M. Casadei, On engineering self-organizing environments: Stochastic methods for dynamic resource allocation, in: D. Weyns, H.V.D. Parunak, F. Michel (Eds.), 3rd International Workshop Environments for Multi-Agent Systems, E4MAS 2006, AAMAS 2006, Hakodate, Japan, 2006. URL: http://www.cs.kuleuven.ac.be/~distrinet/events/e4mas/2006/contents/papers/gardelli.pdf.
[16] L. Gardelli, M. Viroli, A. Omicini, On the role of simulations in engineering self-organising MAS: The case of an intrusion detection system in TuCSoN, in: S.A. Brueckner, G. Di Marzo Serugendo, D. Hales, F. Zambonelli (Eds.), Engineering Self-Organising Systems (3rd International Workshop (ESOA 2005), Utrecht, The Netherlands, 26 Jul. 2005), in: LNAI, vol. 3910, Springer, 2006, pp. 153–168. Revised Selected Papers.
[17] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1) (1985) 80–112.
[18] D. Gelernter, Multiple tuple spaces in Linda, in: Parallel Architectures and Languages Europe (PARLE'89), vol. II: Parallel Languages, Springer-Verlag, 1989, pp. 20–27.
[19] A. Jain, M. Murty, P. Flynn, Data clustering: A review, ACM Computing Surveys 31 (3) (1999) 264–323.
[20] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.
[21] B. Krishnamachari, D. Estrin, S.B. Wicker, The impact of data aggregation in wireless sensor networks, in: Proceedings of the 22nd International Conference on Distributed Computing Systems, ICDCSW'02, IEEE Computer Society, Washington, DC, USA, 2002.
[22] M. Kwiatkowska, G. Norman, D. Parker, Stochastic model checking, in: M. Bernardo, J. Hillston (Eds.), Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation, SFM'07, in: LNCS (Tutorial Volume), vol. 4486, Springer, 2007.
[23] E.D. Lumer, B. Faieta, Diversity and adaptation in populations of clustering ants, in: From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adpative Behavior, MIT Press, Cambridge, MA 02142, USA, 1994, pp. 501–508.
[24] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications with the tota middleware, in: Pervasive Computing and Communications, 2004. PerCom 2004 (Proceedings of the Second IEEE Annual Conference on), IEEE, 2004.
[25] M. Mamei, F. Zambonelli, Programming stigmergic coordination with the TOTA middleware, in: 4th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS'05, ACM Press, New York, NY, USA, 2005.
[26] C. Melhuish, O. Holland, S. Hoddell, Collective sorting and segregation in robots with minimal sensing, in: R. Pfeifer, B. Blumberg, J.-A. Meyer, S.W. Wilson (Eds.), From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior, MIT Press, Cambridge, MA 02142, USA, 1998, pp. 465–470.
[27] C. Melhuish, A.B. Sendova-Franks, S. Scholes, I. Horsfield, F. Welsby, Ant-inspired sorting by robots: The importance of initial clustering, Journal of the Royal Society Interface 3 (7) (2006) 235–242.
[28] R. Menezes, R. Tolksdorf, A new approach to scalable Linda-systems based on swarms, in: Symposium on Applied Computing, SAC'03, ACM Press, New York, NY, USA, 2003.
[29] R. Menezes, R. Tolksdorf, Adaptiveness in Linda-based coordination models, in: G.D.M. Serugendo, A. Karageorgos, O.F. Rana, F. Zambonelli (Eds.), Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering, in: LNAI, vol. 2977, Springer, Berlin, Heidelberg, 2004, pp. 212–232.
[30] R.D. Nicola, D. Latella, M. Massink, Formal modeling and quantitative analysis of klaim-based mobile systems, in: Proceedings of the 2005 ACM Symposium on Applied Computing, SAC'05, ACM Press, New York, NY, USA, 2005.
[31] A. Omicini, E. Denti, From tuple spaces to tuple centres, Science of Computer Programming 41 (3) (2001) 277–294.
[32] A. Omicini, F. Zambonelli, Coordination for Internet application development, Journal of Autonomous Agents and Multi-Agent Systems 2 (3) (1999) 251–269.
[33] A. Phillips, The Stochastic Pi Machine (SPiM), version 0.042, 2006. Available online at: http://www.doc.ic.ac.uk/~anp/spim/. URL: http://www.doc.ic.ac.uk/~anp/spim/.
[34] G. Plotkin, A structural approach to operational semantics, Tech. Rep. DAIMI FN-19, Department of Computer Science, AArhus University, Denmark, 1991.
[35] M. Resnick, Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds, MIT Press, Cambridge, MA 02142, USA, 1997.
[36] J. Rutten, M. Kwiatkowska, G. Norman, D. Parker, Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems, in: CRM Monograph, vol. 23, American Mathematical Society, 201 Charles Street, Providence, RI 02904-2294, United States of America, 2004.
[37] C.E. Shannon, A mathematical theory of communication, Bell System Technical Journal 27.
[38] M. Viroli, M. Casadei, L. Gardelli, A self-organising solution to the collective sort problem in distributed tuple spaces, in: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC 2007, ACM, Seoul, Korea, 2007, special Track on Coordination Models and Languages.
[39] D. Yamins, R. Nagpal, Automated global-to-local programming in 1-d spatial multi-agent systems, in: 7th International Joint Conference on Agents and Multi-Agent Systems, AAMAS-08, IFAAMAS, Estoril, Portugal, 2008.
[40] B.-Y. Yaneer, Dynamics of Complex Systems, Studies in Nonlinearity, Westview Press, 1997.
[41] K. Yuen, B. Liang, L. Baochun, A distributed framework for correlated data gathering in sensor networks, IEEE Transactions on Vehicular Technology 57 (1) (2008) 578–593.