Some Thoughts on Transiently Shared Dataspaces

Nadia Busi
Department of Computer Science
University of Bologna
Mura A. Zamboni 7, 40127 Bologna, Italy
busi@cs.unibo.it

Gianluigi Zavattaro
Department of Computer Science
University of Bologna
Mura A. Zamboni 7, 40127 Bologna, Italy
zavattar@cs.unibo.it

Abstract

Transiently Shared Dataspaces (TSD), recently introduced in the coordination infrastructure Lime, represent an emerging technology to enable the use of dataspaces for the coordination of mobile agents: data-sharing is allowed among agents running on hosts belonging to the same federation (i.e., currently connected). In this paper we present a formalization of the TSD technology in order to provide a framework for reasoning about systems based on this infrastructure. In particular, we concentrate on alternative design choices related to data migration, host connectivity, and reactive programming.

1 Introduction

Typical shared dataspace coordination languages such as Linda [4], JavaSpaces [6], and TSpaces [8] exploit a persistent and globally shared data repository to achieve inter-agent interaction. The diffusion of mobile devices calls for the development of new coordination middlewares suitable for highly reconfigurable networks.

An interesting step in this direction is represented by the introduction of the notion of Transiently Shared Dataspace [7]. The leading idea is to associate a local dataspace to each agent, and to allow for the sharing of data among agents residing on connected hosts. In this context both logical and physical mobility may be taken into account: the former is modeled via an explicit request of agent migration towards a new host, whereas the latter is achieved by means of a dynamically changing connectivity relation, partitioning the set of hosts in federations of communicating components. A necessary step towards a deeper understanding of this new approach is the definition of a formal framework suitable for the comparison of alternative design choices, the investigation of the scalability of the features and properties of the traditional shared dataspace languages, as well as the analysis of aspects emerging in this new setting.

2 Formalizing Agents, Hosts, and Federations

An agent is an active program with its own local dataspace. Agents reside on hosts; logical mobility of agents is modeled by allowing agents to move from a host to another one. A federation is a group of connected hosts which may communicate; the agents belonging to federated hosts share their data through the abstraction of a Transiently Shared Dataspace which comprises all the local dataspaces. Physical mobility of hosts is modeled by supporting the dynamic reconfiguration of the federations.

The calculus we propose extends previous works on process calculi based on the shared dataspace model [3, 2]; an agent is described as a term of an algebra where the basic actions are typical coordination primitives.

We consider a set $Data$, ranged over by $d, e, \ldots$, of data, a set $AName$, ranged over by $a, b, \ldots$, of agent names, and a set $HName$, ranged over by $h, k, \ldots$, of host names.

A program P is a term generated by the following grammar:

$$\begin{array}{rl} P ::= & \mathbf{0} \mid K \mid \mu.P \mid \eta?P\_P \mid new(P).P \mid go(h).P \\ \mu ::= & out(d) \mid in(d) \mid rd(d) \mid \\ & out(d)@a \mid in_L(d) \mid rd_L(d) \\ \eta ::= & inp(d) \mid rdp(d) \mid inp_L(d) \mid rdp_L(d) \end{array}$$

where K stands for a generic element of a set $PName$ of process names; we assume that all process name occurrences are equipped with a corresponding (guarded) defining equation of the form $K = P$. Process names are used to support recursive definitions as, for example, in the term $Ren = in(d).out(e).Ren$, which represents a process able to repeatedly rename data of the kind d in data of the kind e. Let $Prog$ denote the set of all possible programs.

Term $\mathbf{0}$ is a program that can do nothing. Term K is a program defined by an agent name definition $K = P$. Term $\mu.P$ is a program that can perform the coordination action μ and then behaves like P. We consider the typical Linda output, in, and read coordination actions. Concerning output, we consider a local version $out(d)$ which introduces the new datum in the shared dataspace of the agent which performs the operation, and a remote version $out(d)@a$ which directs the emitted datum towards the dataspace of agent a. Regarding input and read, we consider a global version $in(a), rd(a)$ to act on the entire transiently shared dataspace (all the dataspaces of the federated agents are considered), and a local version $in_L(a), rd_L(a)$ to consider the local dataspace of the agent performing the operations. Agent $\eta?P\_Q$ performs a non-blocking input or read coordination operation; if the operation succeds the first program P is activated, otherwise (in the case the required datum is absent) program Q is started. Also in this case we consider the global and the local versions. We consider two further operations: $new(P)$ to create a new agent running on the local host which starts with program P and an empty dataspace, and $go(h)$ to move the agent to the host h.

In the following we use the notation: $\mathcal{P}(Set)$ to represent the power-set of Set and $\mathcal{M}(Set)$ to denote the set of the multisets on Set. Sets and multisets are both represented by the classical notation (using the brackets $\{\,\}$ which we sometime omit); the actual meaning is made clear by the context. Set and multiset union is denoted by $\oplus$ (also in this case the actual use is made clear by the context). Given a multiset m, with $m(d)$ we denote the number of occurrences of d in m.

We are now ready to define the set $Agent$ of the possible agents and the set $Config$ of the possible global system configurations:

$$\begin{array}{lcl} Agent & = & \{a \triangleright h[P, DS] \mid a \in AName, \\ & & P \in Prog, h \in HName, \\ & & DS \in \mathcal{M}(Data)\} \\ Config & = & \{Ag \mid Ag \in \mathcal{P}(Agent) \text{ s.t. } WF(Ag)\} \end{array}$$

where $WF(Ag)$ is true if and only if in a set of named agents the same name is not used twice:

$$\begin{array}{l} WF(Ag) = true \text{ iff} \\ (a \triangleright h[P, DS], a \triangleright h'[P', DS'] \in Ag) \rightarrow \\ (h = h' \ \wedge \ P = P' \ \wedge \ DS = DS') \end{array}$$

An agent $a \triangleright h[P, DS]$ is a pair composed by a program P and a dataspace DS equipped with an agent name a and the indication of the host h on which the agent resides.

In order to model host connectivity we exploit an equivalence relation $\bowtie \subseteq HName \times HName$: $h \bowtie h'$ means that the host h is connected to the host h'. According to this equivalence relation, the set $HName$ is partitioned in classes, each one modeling a cluster of connected hosts. To model physical mobility of hosts (a mobile host may disengage a federation in order to engage another one) we assume that $\bowtie$ is not fixed but it can change dynamically: $\bowtie \mapsto \bowtie'$ represents a modification of the clusters of connected hosts.

3 Data Migration

Data migration is achieved in our framework by exploiting the remote output operation $out(d)@a$ which emits a new datum d towards the dataspace of the agent a. In the case the agent a is located on a non-connected host the emitted datum cannot reach its destination.

Consider the following configuration:

$$b \triangleright h[out(d)@a.P, DS], a \triangleright h'[P', DS']$$

where $h \not\bowtie h'$. The new datum d cannot be introduced in the dataspace DS' of the agent a as long as the two agents are disconnected. In this case several approaches could be adopted.

A first proposal could be to block the output operation until it can be completed, i.e., until the two agents are connected. To model this approach we could admit a unique outgoing transition for the above configuration:

$$\begin{array}{ll} b \triangleright h[out(d)@a.P, DS], a \triangleright h'[P', DS'] \longrightarrow & \\ b \triangleright h[P, DS], a \triangleright h'[P', DS' \oplus a] & \quad h \bowtie h' \end{array}$$

In many applications, this approach is not satisfactory because the agent b could remain blocked forever, e.g., in the case the target agent will never be reachable.

The operational semantics we present in the Appendix follows a different approach, inspired by the modeling of data migration in Lime. An auxiliary datum $d@a$ is introduced in the local dataspace of the source agent b

$$\begin{array}{ll} b \triangleright h[out(d)@a.P, DS], a \triangleright h'[P', DS'] \longrightarrow & \\ b \triangleright h[P, DS \oplus d@a], a \triangleright h'[P', DS'] & \quad h \not\bowtie h' \end{array}$$

When the hosts h and h' become federated, the datum d reaches its destination according to the transition

$$\begin{array}{ll} b \triangleright h[P, DS \oplus d@a], a \triangleright h'[P', DS'] \longrightarrow & \\ b \triangleright h[P, DS], a \triangleright h'[P', DS' \oplus d] & \quad h \bowtie h' \end{array}$$

Following the Lime approach, the auxiliary datum $d@a$ could be read or consumed by other agents even before it reaches the target dataspace. This could be useful, e.g., in order to permit the garbage collection of the data which will never reach their destination.

However, also this approach could give rise to unexpected behaviours. Consider a system composed of three agents a, b, and c:

$$\begin{array}{l} a \triangleright h[out(d)@c, DS], b \triangleright h'[rd(d).out(d')@c, DS'], \\ c \triangleright h''[in(d').inp_L(d)?OK\_Fail, DS''] \end{array}$$

Agent a produces the datum d for the local dataspace of c; agent b performs a read operation on d, and then emits a new datum d' for the local dataspace of c. The agent c performs an input operation on d'; at this point the datum d

is expected to be present in the local dataspace of c as it has been produced before d'.

Nevertheless, it could be the case that d is not available yet. Consider an initial configuration in which a is connected with b but not with c. In this configuration both the data d and d' can be produced. If, in the following, b becomes connected with c before a, the datum d' will be introduced in the dataspace of c before d. At this point the unexpected $Fail$ program could be activated.

An alternative approach to deal with remote output operations is to encapsulate the datum in a message to be delivered to the target agent. More precisely, a remote output is composed of two independent phases: an *emission* which produces the message, and a *rendering* during which the message reaches its destination and the datum is introduced in its target dataspace:

$$\begin{array}{l} a \triangleright h[out(d)@b.P, DS], b \triangleright h'[P', DS'] \;\longrightarrow \\ a \triangleright h[P, DS], \langle d@b \rangle, b \triangleright h'[P', DS'] \;\longrightarrow \\ a \triangleright h[P, DS], b \triangleright h'[P', DS' \oplus d] \end{array}$$

where $\langle d@b \rangle$ represents the migrating datum. According to this alternative semantics, the datum becomes visible only when it reaches its destination: in this way the unexpected $Fail$ program cannot be activated in the above example.

Another interesting effect of this alternative semantics is a sensible simplification of the implementation of the engagement procedure performed on changes of hosts connectivity. Indeed, the Lime-like semantics requires the atomic migration of all those misplaced data which could reach their final destination. This phase is no longer required in the new approach which treats each migrating data independently.

4 Host Connectivity

In our calculus we modeled host connectivity through an equivalence relation $\bowtie$, thus assuming that hosts are partitioned in federations. In this section we analyse alternative assumptions on the $\bowtie$ relation.

Consider two wireless networks such that the areas that they cover have a non-empty intersection. A device located in the intersection of the two areas may connect with both the networks, while devices outside this area have visibility of at most one of the two networks. Let h be the host in the intersection, and let h_1 and h_2 be two hosts respectively connected with only the first and the second network. Suppose that a, b, and c are three agents which reside on h, h_1, and h_2 respectively:

$$\begin{array}{l} a \triangleright h[rd(d).rd(d').rdp(d)?OK\_Fail, \emptyset], \\ b \triangleright h_1[out(d).rd(d').Fail, \emptyset], \\ c \triangleright h_2[out(d').rd(d).Fail, \emptyset] \end{array}$$

The agents b and c produce data d and d', respectively. These data are then read by the agent a. At this point we could assume that a observes the presence of both d and d', agent b cannot see the presence of d', and c cannot observe d. This assumption implies that no one of the $Fail$ program could be activated.

Nevertheless, according to the semantics in the Appendix there exist computations which activate the unexpected $Fail$ program. Indeed, in order for the agent a to read d' we assume that $h \bowtie h_2$. If at this point $h \not\bowtie h_1$ then the subsequent $rdp(d)$ operation could fail thus activating the agent $Fail$. Thus we should assume that $h \bowtie h_1$; by transitivity also $h_1 \bowtie h_2$ holds, thus allowing the execution, e.g., of the $rd(d')$ operation of b. If this happens, the $Fail$ program is activated.

This example proves that there exist situations in which the assumption that $\bowtie$ is an equivalence relation is not adequate, in particular, transitivity reveales to be too strong.

We could be interested also in weakening the symmetry assumption. Consider an application in which, e.g., due to a particular security policy, the agents located on a host h may access the dataspaces of the agents running on h', but not the vice versa. In this case we should have $h \bowtie h'$ but $h' \not\bowtie h$, thus breaking symmetry.

5 Reactive Programming

Most of the recent proposals of shared dataspace coordination languages (Lime, JavaSpaces, TSpaces) exploit some form of reaction mechanism, borrowed from event-based, publish-subscribe middlewares (see, for example, JEDI [5] and the references therein). As an example, the $notify(d, P)$ primitive of JavaSpaces permits to a process to register interest in the incoming arrivals of datum d. The execution of the $notify$ operation creates a listener on datum d; each time a new instance of datum d is produced in the dataspace, a reaction is activated, consisting in spawning a copy of process P. However, when moving to a TSD scenario, some choices need to be performed in order to clarify what is the intended semantics of the $notify$ primitive when the set of data available to an agent dynamically varies, due to changes in network connectivity. First of all, what is the meaning of the sentence "each time a new instance of datum d is produced in the dataspace"? According to the Lime philosophy, each time a new instance of datum d becomes available to the agent on which the listener resides, a reaction takes place. The first observation is that the temporal constraint between $notify$ and out, namely, the fact that a $notify(d, P)$ operation registers interest in *future* productions of datum d, is broken. As an example, consider the following processes in a single shared dataspace scenario:

$$out(d_1),\; rd(d_1).out(d_2),\; in(d_2).notify(d_1, Fail)$$

There is a causal dependency among the execution of $notify$ by the third process and the execution of $out(d_1)$ by the first process, hence the listener created by the third process cannot react to the production of datum d_1; thus, program $Fail$ can never be activated. Now, consider the three processes above as three agents residing on different hosts, as in the following configuration:

$$a_1 \triangleright h_1[out(d_1), \emptyset],\\ a_2 \triangleright h_2[rd(d_1).out(d_2), \emptyset],\\ a_3 \triangleright h_3[in(d_2).notify(d_1, Fail), \emptyset]$$

If the initial network partition is $\{h_1, h_2\}, \{h_3\}$, and it changes to $\{h_1\}, \{h_2, h_3\}$ after a_2 performed the rd operation, and finally it becomes $\{h_1, h_2, h_3\}$ after a_3 performed the $notify$, then the $Fail$ program is executed. Another drawback comes into play in case of a sequence of temporary disconnections of h_1 and h_3; when the connection is restored, datum d_1 becomes available another time to agent a_3. To prevent the listener from reacting again and again, it is necessary to provide a mechanism to distinguish each instance of datum that has been produced in the whole system and to remember the data instances for which a listener has already reacted to.

An alternative approach consists in making listeners to react to the execution of the $out(d)$ operation by a federated agent, that is, only listeners residing on hosts currently connected with the host where the $out(d)$ operation is performed will react. However, in that case it is no longer possible for an agent to react to the presence of a datum as soon as it becomes available in the federation, i.e., also due to a reconfiguration of connections among hosts.

References

[1] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Comput. Sci.*, 96:217–248, 1992.

[2] N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. *Theoretical Comput. Sci.*, 340(1):49–90, 2000.

[3] N. Busi and G. Zavattaro. On the Expressivenes of Event Notification in Data-Driven Coordination Languages. In *Proc. of ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 41–55. Springer-Verlag, Berlin, 2000.

[4] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Commun. ACM*, 35(2):97–107, 1992.

[5] G. Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proc. of the 20th ICSE*, 1998.

[6] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1998.

[7] G. Picco, A. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Proc. of the 21th ICSE*, 1999.

[8] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Syst. J.*, 37(3):454–474, 1998.

A Operational Semantics

We represent the system behaviour by means of a transition system: the configurations represent the state of the system, whereas the system moves are represented by a transition relation $\longrightarrow$ relating two configurations. The configuration of a system evolves by execution of coordination operations or by change of the connectivity relation. We formally define the rules governing the evolution of a system as a set of rewriting rules, following a slight variation of the CHAM style [1]. The rules are listed in Table 1 (we omit the rules for the rd and rdp operations as similar to in and inp, respectively); characters in bold are used to emphasize the elements directly involved in a transition.

The execution of a local output operation has the effect of adding the emitted datum in the local space of the agent. The effect of a remote output of datum d depends on the connectedness of the hosts of the agent a performing the operation and the agent b, target of the operation. If the hosts of a and b are connected, the emitted datum is added to b's dataspace; otherwise, a misplaced datum $d@b$ is added to a's dataspace; the misplaced datum will reach its target dataspace as soon as connectivity between a and b is restored. Finally, a remote output whose target is the agent performing the operation behaves in the same way as a local output. The local input operation removes an instance of the datum, if one exists in the agent's dataspace, otherwise it blocks. The global input looks for the datum both locally and in the dataspace of each agent residing on a connected host; the consumed datum can be either a datum that already reached its target dataspace or a misplaced one.

The lookup policies for the local and global versions of the inp predicate are the same as those for the corresponding versions of blocking operation in. The local inp predicate fail when the required datum is not present in the local dataspace, whereas the global version fail when the required datum is absent also in the dataspaces of all the agents residing on connected hosts.

If the target host k is connected to the actual host, a $go(k)$ operation performed by an agent causes its migration to the target host. The execution of a $new(P)$ operation spawns a new agent, with program P and empty dataspace, on the same host of the agent performing the operation. A program constant K behaves as its defining program.

The most involved rule deals with changes in host connectivity, which cause the migration of some misplaced data to their target dataspace. Upon change of the connectivity relation, the dataspace of each agent a is updated in the following way: any misplaced datum whose target is an agent residing on a host that has become connected with a is removed; for each instance of misplaced datum $d@a$, belonging to agents residing on hosts that have become connected to a's host, a datum d is added.

$$a \rhd h[\mathbf{out(d).P}, DS] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS \oplus \mathbf{d}] \oplus Ag$$

$$a \rhd h[\mathbf{out(d)@b.P}, DS] \oplus b \rhd k[P', DS'] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus b \rhd k[P', DS' \oplus \mathbf{d}] \oplus Ag \qquad \text{if } h \bowtie k$$

$$a \rhd h[\mathbf{out(d)@b.P}, DS] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS \oplus \mathbf{d@b}] \oplus Ag \qquad \text{if } host_{Ag}(b) \not\bowtie h$$

$$\text{where } host_{\emptyset}(b) = \emptyset \text{ and } host_{a \rhd h[P,DS] \oplus Ag}(b) = \begin{cases} h & \text{if } a = b \\ host_{Ag}(b) & \text{otherwise} \end{cases}$$

$$a \rhd h[\mathbf{out(d)@a.P}, DS] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS \oplus \mathbf{d}] \oplus Ag$$

$$a \rhd h[\mathbf{in_L(d).P}, DS \oplus \mathbf{d}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus Ag$$

$$a \rhd h[\mathbf{in(d).P}, DS \oplus \mathbf{d}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus Ag$$

$$a \rhd h[\mathbf{in(d).P}, DS \oplus \mathbf{d@b}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus Ag$$

$$a \rhd h[\mathbf{in(d).P}, DS] \oplus b \rhd k[P', DS' \oplus \mathbf{d}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus b \rhd k[P', DS'] \oplus Ag \qquad \text{if } h \bowtie k$$

$$a \rhd h[\mathbf{in(d).P}, DS] \oplus b \rhd k[P', DS' \oplus \mathbf{d@c}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus b \rhd k[P', DS'] \oplus Ag \qquad \text{if } h \bowtie k$$

$$a \rhd h[\mathbf{go(k).P}, DS] \oplus Ag \longrightarrow a \rhd k[\mathbf{P}, DS] \oplus Ag \qquad \text{if } h \bowtie k$$

$$a \rhd h[\mathbf{new(P).P'}, DS] \oplus Ag \longrightarrow a \rhd h[\mathbf{P'}, DS] \oplus \mathbf{n} \rhd \mathbf{h}[\mathbf{P}, \emptyset] \oplus Ag \qquad \text{with } \mathbf{n} \text{ fresh}$$

$$\frac{a \rhd h[\mathbf{P}, DS] \oplus Ag \longrightarrow Ag'}{a \rhd h[\mathbf{K}, DS] \oplus Ag \longrightarrow Ag'} \qquad \text{if } K = P$$

$$a \rhd h[\mathbf{inp_L(d)?P\_Q}, DS \oplus \mathbf{d}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus Ag$$

$$a \rhd h[\mathbf{inp(d)?P\_Q}, DS \oplus \mathbf{d}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus Ag$$

$$a \rhd h[\mathbf{inp(d)?P\_Q}, DS \oplus \mathbf{d@b}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus Ag$$

$$a \rhd h[\mathbf{inp(d)?P\_Q}, DS] \oplus b \rhd k[P', DS' \oplus \mathbf{d}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus b \rhd k[P', DS'] \oplus Ag \qquad \text{if } h \bowtie k$$

$$a \rhd h[\mathbf{inp(d)?P\_Q}, DS] \oplus b \rhd k[P', DS' \oplus \mathbf{d@c}] \oplus Ag \longrightarrow a \rhd h[\mathbf{P}, DS] \oplus b \rhd k[P', DS'] \oplus Ag \qquad \text{if } h \bowtie k$$

$$a \rhd h[\mathbf{inp_L(d)?P\_Q}, \mathbf{DS}] \oplus Ag \longrightarrow a \rhd h[\mathbf{Q}, \mathbf{DS}] \oplus Ag \qquad \text{if } \mathbf{d} \notin \mathbf{DS}$$

$$a \rhd h[\mathbf{inp(d)?P\_Q}, \mathbf{DS}] \oplus \bigoplus_i a_i \rhd h_i[P_i, \mathbf{DS_i}] \oplus Ag \longrightarrow a \rhd h[\mathbf{Q}, \mathbf{DS}] \oplus \bigoplus_i a_i \rhd h_i[P_i, \mathbf{DS_i}] \oplus Ag \qquad \text{if}$$

$$\mathbf{d} \notin (\mathbf{DS} \oplus \bigoplus_i \mathbf{DS_i}) \text{ and } (hosts(Ag) \times h) \cap \bowtie = \emptyset$$

$$\text{where } hosts(\bigoplus_{i \in I} a_i \rhd h_i[P_i, DS_i]) = \{h_i \mid i \in I\}$$

$$\frac{\bowtie \mapsto \bowtie'}{\bigoplus_i a_i \rhd h_i[P_i, \mathbf{DS_i}] \longrightarrow \bigoplus_i a_i \rhd h_i[P_i, \mathbf{DS'_i}]} \qquad \text{where} \qquad \begin{array}{l} \mathbf{DS'_i}(d) = \mathbf{DS_i}(d) + \sum_{j : h_i (\bowtie' \setminus \bowtie) h_j} \mathbf{DS_j}(d@a_i) \\ \mathbf{DS'_i}(d@a_j) = \begin{cases} 0 & \text{if } h_i (\bowtie' \setminus \bowtie) h_j \\ \mathbf{DS_i}(d@a_j) & \text{otherwise} \end{cases} \end{array}$$

Table 1. The operational semantics.