On how Distribution and Mobility Interfere with Coordination

Antónia Lopes[1] and José Luiz Fiadeiro[2]

[1]Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, PORTUGAL
mal@di.fc.ul.pt

[2] Department of Mathematics and Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@fiadeiro.org

Abstract. With the advent of the e-Economy, system architectures need to take into account that Distribution and Mobility define a dimension that is not orthogonal to the traditional two — coordination and computation. In this paper, we address the way that this additional dimension interferes with coordination. First, we address the effectiveness of connectors in place in a system when the properties of the media through which its components can be effectively coordinated are taken into account. Then, we consider the modelling of coordination styles that are location-dependent or even involve the management of the location of the coordinated parties.

1 Introduction

Architecture-based approaches to the design and construction of software systems focus on the gross modularisation of systems as collections of interacting components. At the architecture level, the aspects related to distribution, such as the way the system is supposed to be mapped into a network, are not usually addressed. This conforms with traditional forms of distributed systems that consider that the environment where a system executes — the physical nodes and links — is statically configured, the distribution of the system over these nodes is also static and that all forms of communication and access to resources can be provided (directly or indirectly) by the operating system or the middleware.

With the advent of Mobile Computing in wide and ad-hoc networks, new forms of distributed systems come into play. In mobile computing systems, components are entitled to move across a network that is not necessarily statically determined. The network itself may be constituted by mobile nodes without a fixed infrastructure and, hence, their connectivity may change over time. In this situation, it is no longer reasonable to abstract away from component location and the properties of the physical distribution topology of locations and communication infrastructure. This is particularly true at the architectural level. For instance, in architectures that are structured in

terms of components and connectors [1][2], it is no longer possible to assume that the coordination mechanisms put in place through connectors can be made effective across the physical links that connect the hosts of the components in the underlying network. Another reason to address the distribution and mobility dimension of systems at an architectural level is the fact that this third dimension (together with computation and coordination) is an additional source of complexity in system development and, hence, should be addressed at the highest possible level of abstraction. Therefore, concepts and techniques are needed to support the description of the aspects related to distribution and mobility at the architectural level of software design.

As a first step towards the more ambitious goal of having an architectural approach to distribution and mobility, we have been investigating the addition of this new dimension to the architectural framework that we developed in the past [9]. In our first step in this direction [11], an extension of CommUnity was proposed in order to support the description of the distribution and mobility dimension of systems. This extension was developed having in mind the goal of being able to represent distribution and mobility explicitly in architectures. In spite of its apparent simplicity, this goal comprises many different aspects that result from the different ways computation, coordination and mobility can interfere. For instance, in what concerns the interference between computation and mobility, we have already shown that patterns of distribution and mobility of components, or groups of components, can be explicitly represented in architectural descriptions through what we have called *distribution connectors*, similarly to the way *coordination connectors* represent component interactions.

In this paper we will explore the interference between coordination and mobility, in particular the emergence of several new abstractions for program interaction such as transient interaction (e.g., transient variable sharing) [13] and code mobility (e.g., remote evaluation and mobile agents) [5]. We will show that these coordination constructs, that facilitate component interactions in mobile systems, can be modelled as coordination connectors, and used in systems architectures together with the connectors that model traditional communication primitives such as asynchronous communication and remote procedure call. We shall also address the impact on system architecture of the properties of the locations in which components perform their computations and the properties of the media through which their interconnections can be effectively coordinated.

The paper is organised as follows. Section 2 briefly introduces the extension we have proposed for CommUnity. It encompasses an extension of the program design language in order to support the design of components that have location-dependent patterns of computing, namely mobile components, an extension of the primitive mechanisms that support program interaction, and the definition of program composition in this context. Then, in section 3, we shall illustrate how coordination mechanisms put in place through coordination connectors can become ineffective and need to be replaced with ones that are compatible with the connection topology among the components available at physical level. Then, section 4 shows how some of the new forms of coordination raised by mobility can be expressed as coordination connectors and used in systems architectures. Section 5 closes the paper with some conclusions.

2 The CommUnity framework

CommUnity, introduced in [6], is a parallel program design language that is similar to Unity [3] in its computational model but adopts a different coordination model. CommUnity relies on the sharing (synchronisation) of actions and exchange of data through input and output channels and, moreover, it requires interactions between components to be made explicit. In CommUnity, the separation between "computation" and "coordination" is taken to an extreme in the sense that the definition of the individual components of a system is completely separated from the interconnections through which these components interact.

Syntax

CommUnity was recently extended in order to support the design of the distribution and mobility dimension of systems [11]. It adopts an explicit representation of the space within which movement takes place, but no specific notion of space is assumed. This is achieved by considering that "space" is constituted by the set of possible values of a special data type *Loc* included in a fixed data type specification over which components are designed. The data sort *Loc* models the positions of the space in a way that is considered to be adequate for the particular application domain in which the system is or will be embedded. The only requirement that we make is for a special location $\perp$ to be distinguished (its role will be discussed further below).

In order to model systems that are location-aware, we make explicit how system "constituents" (output and private channels, actions, or any group of these) are mapped to the positions of the space statically determined by *Loc*. This is achieved by associating each "constituent" of a system with a location variable. Mobility is then associated with the change of value of location variables.

A CommUnity design is defined in terms of input and output location variables (resp. denoted by lI, lO), input, ouput and private channels (resp. denoted by I, O and V) and a set of action names (Γ).

Channels. Private channels model internal communication. Input channels are used for reading data from the environment of the component and the component has no control on the values that are made available there. Output channels allow the environment to read data produced by the component. Each channel v is typed with a sort $sort(v)$. We shall use X to denote $I \cup O \cup V$.

Location Variables. Location variables (locations, for short) in a component design can be declared as *input* or *output* in the same way as channels but are all typed with sort *Loc*. Input locations are read from the environment and cannot be modified by the component. Hence, if $l \in lI$, the movement of any constituent located at l is under the control of the environment. Output locations (lO) can only be modified locally but can be read by the environment. Hence, if $l \in lO$, the movement of any constituent located at l is under the control of the component.

Each local channel x of a design is associated with a location l. We make this assignment explicit by writing $x@l$. The value of l indicates the current position of the space where the values of x are made available. A modification in the value of l en-

tails the movement of x as well as of the other channels and actions located at that location variable.

Input channels are located at a special output location λ whose value is invariant and given by $\perp$. The intuition is that this location variable is a non-commitment to any particular location. The idea is that input channels will be assigned a location when connected with a specific output channel of some other component of the system. Every channel x is associated with a set of locations $\Lambda(x)$ which is $\{\lambda\}$ if x is an input channel and is $\{l,\lambda\}$ in the case of an output channel $x@l$. We shall use L to denote the pointed set of locations $(lI \cup lO)_\lambda$ and $local(X,L)$ to denote the union $V \cup O \cup lO$ of *local* channels and locations.

Actions. Actions can be declared either as *private* or *shared*. Private actions represent internal computations in the sense that their execution is uniquely under the control of the component. Shared actions represent possible interactions between the component and the environment, meaning that their execution is also under the control of the environment. As we will see, actions provide points of *rendez-vous* at which components can synchronise.

Each action name g is associated with a set $\Lambda(g)$ of locations including λ, meaning that the execution of action g is distributed over those locations. In other words, the execution of g consists of the synchronous execution of a guarded command in each of these locations. Guarded commands are associated with located actions, i.e. pairs $g@l$, for $l \in \Lambda(g)$, as follows:

- $D(g@l)$ is a subset of $local(X,L)$ consisting of the local channels into which executions of the action can place values and of the locations to which the action can inflict movement. This is what is sometimes called the *write frame* of $g@l$. For simplicity, we will often omit the explicit reference to the write frame when $D(g@l)$ can be inferred from the assignments. Given a private or output channel v, we will also denote by $D(v)$ the set of actions $g@l$ such that $v \in D(g@l)$. The fact that the special location λ is invariant is ensured by the condition $D(\lambda)=\emptyset$. We denote by $F(g@l)$ the *frame* of $g@l$, i.e., the channels that are in $D(g@l)$ or used in $G(g@l)$.
- $G(g@l)$ is the guard condition.
- $R(g@l)$ is a conditional multiple assignment on the local channels and locations declared in $D(g@l)$. When the write frame $D(g@l)$ is empty, $R(g@l)$ is denoted by *skip*.

When a design does not explicitly refer the guarded command associated with $g@\lambda$, this is because it is the empty command $[true \rightarrow skip]$. In this way, every standard CommUnity design (location-unaware) defines trivially a distributed design: the one that has all its actions and channels located at λ.

Variations in the context of execution of a mobile system are not limited to the locations of its components and respective hosts. It is important that other observables, s.a. network bandwidth, battery power or the communication range, can be used at the programming level. In CommUnity, we have only a construct *inrange:Loc->bool* that allows a program to observe if a given position of the space is in its communication range.

Semantics

We assumed that the distribution space is constituted by the set of possible values of a given data sort Loc. We consider that, once we fix an algebra U for the data types, namely a domain U_{Loc} for Loc, the relevant properties of the mobility space are captured by two binary relations over U_{Loc}:

- A relation *bt* s.t. *n bt m* means that *n* and *m* are positions in the space "in touch" with each other. Interactions among components can only take place when they are located in positions that are "in touch" with one another. Because the special location variable λ intends to be a position to locate entities that can communicate with any other entity in a location-transparent manner, we require that the value of λ is always set at configuration time as being $\perp_U$ and, furthermore, $\perp_U$ *bt m*, for every $m \in U_{Loc}$.
- A relation *reach* s.t. *n reach m* means that position *n* is reachable from *m*. Permission to move a component or a group of components is conceded when the new position "is reachable" from the current one.

In general, the topology of locations is dynamic and, hence, the operational semantics for a program is given in terms of an infinite sequence of relations $(bt_i, reach_i)_{i \in N}$. At each execution step, one of the actions that can be executed is chosen and executed. The conditions under which a distributed action *g* can be executed at time *i* are the following:

1. *for every* $l_1, l_2 \in \Lambda(g)$, $[l_1]^i \; bt_i \; [l_2]^i$: the execution of *g* involves the synchronisation of its local actions and, hence, their locations have to be in touch.

2. *for every* $l \in \Lambda(g)$, *g@l can be executed, i.e.,*

i. *for every* $x \in F(g@l)$, $[l]^i \; bt_i \; [\Lambda(x)]^i$: the execution of *g@l* requires that every channel in the frame of *g@l* can be accessed and, hence, *l* has to be in touch with their locations.

ii. *for every location* $l_1 \in D(g@l)$ *and* $m \in [R(g)]^i(l_1)$, $m \; reach_i \; [l_1]^i$: if a location l_1 can be effected by the execution of *g@l*, then every possible new value of l_1 must be a position reachable from the current one.

iii. *the local guard G(g@l) evaluates to true*

where $[e]^i$ denotes the value of the expression *e* at time *i*. In the case of expressions *e* involving *inrange(exp),* the value of $[e]^i$ also depends on the location *l* where the expression is being evaluated and is defined by $[l]^i \; bt \; [exp]^i$.

Given this, when, in an execution step, one of the actions whose enabling condition holds of the current state is selected, its assignments are executed atomically as a transaction. Furthermore, it is guaranteed that private actions that are infinitely often enabled are selected infinitely often.

Interaction and Composition

The primitive mechanisms that support component interaction in CommUnity are the synchronisation of actions and the interconnection of input channels of a component with output channels of other components. The extension of the language with a distribution dimension also entails the definition of mechanisms that support the interac-

tion of designs at the level of their locations. Such mechanisms are essentially the interconnection of input locations of a component with output locations of other components.

In CommUnity, interactions between components have to be made explicit and external to the components by providing the name bindings that express input/output communication and action synchronisation. This externalisation of interactions can be expressed via the following notion of morphism:

Given designs P_i with channels X_i, actions Γ_i and locations L_i, a morphism σ: $P_1 \to P_2$ consists of a total function σ_{ch}: $X_1 \to X_2$, a partial mapping σ_{ac}: $\Gamma_2 \to \Gamma_1$ and a total function σ_{lc}: $L_1 \to L_2$ that preserves the pointed element (λ), satisfying:

1. *for every $x \in X_1$ and $l \in L_1$:*

   a. $sort_2(\sigma_{ch}(x)) = sort_1(x)$ **b.** *if* $x \in out(X_1)$ $(prv(X_1))$ *then* $\sigma_{ch}(x) \in out(X_2)$ $(prv(X_2))$ **c.** *if* $x \in in(X_1)$ *then* $\sigma_{ch}(x) \in out(X_2) \cup in(X_2)$ **d.** *if* $l \in outloc(L_1)$ *then* $\sigma_{lc}(l) \in outloc(L_2)$ **e.** $\sigma_{lc}(\Lambda_1(x)) \subseteq \Lambda_2(\sigma_{ch}(x))$

2. *for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined:*

   a. *if* $g \in sh(\Gamma_2)$ $(prv(\Gamma_2))$ *then* $\sigma_{ac}(g) \in sh(\Gamma_1)$ $(prv(\Gamma_1))$
   b. $\sigma_{lc}(\Lambda_1(\sigma_{ac}(g))) \subseteq \Lambda_2(g)$

3. *for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined and $l \in \sigma_{lc}^{-1}(\Lambda(g))$:*

   a. $\sigma_{ch}(D_1(\sigma_{ac}(g)@l)) \subseteq D_2(g@\sigma_{lc}(l))$ **b.** $\sigma_{ac}(D_2(\sigma(x))) \subseteq D_1(x)$ *for* $x \in local(X_1, L_1)$
   c. $\Phi \models R_2(g@\sigma_{lc}(l)) \supset \underline{\sigma}(R_1(\sigma_{ac}(g)@l))$ **d.** $\Phi \models G_2(g@\sigma_{lc}(l)) \supset \underline{\sigma}(G_1(\sigma_{ac}(g)@l)$

where $\models$ means validity in the first-order sense taken over the axiomatisation Φ of the underlying data types (which includes the location space). Designs and morphisms constitute a category ***DSGN****.*

This notion of morphism extends what in the literature on parallel program design is known as superposition [4][10][8] by taking the distribution aspects into account. A morphism σ: $P_1 \to P_2$ identifies a way in which P_1 is "augmented" to become P_2 so that it can be considered as having been obtained from P_2 through the superposition of additional behaviour, namely the interconnection of one or more components. In other words, σ identifies P_1 as a component of P_2.

The map σ_{ch} identifies for every channel of the component the corresponding channel of the system. The first group of constraints establish that sorts and types of channels have to be preserved but input channels of a component may become output channels of the system.

The partial mapping σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. This mapping is partial and contravariant to account for the fact that, on the one hand, superposition may unfold actions of the original program and, on the other hand, new actions may be added. Condition 2.a states that the type of actions is preserved.

The map σ_{lc} identifies for every location variable of the component the corresponding location variable of the system. As for channels, output locations are mapped into output locations but input locations of a component may become output locations of the system as the result of interconnecting an input location of a compo-

nent with an output location of another component. Conditions 1.e and 2.b state that the locations of channels and actions are preserved.

Conditions 3.a and b require that change within a component be completely encapsulated in the structure of actions defined for the component, namely they ensure that the actions of a component leave the local channels and locations of the other components out of their scope.

The last two conditions require that the computations performed by the system reflect the interconnections established between its components. Condition 3.c reflects the fact that the effects of the actions of the components can only be preserved or made more deterministic in the system and condition 3.d allows the guard of the action to be strengthened but not weakened. Strengthening of the guard reflects the fact that all the components that participate in the execution of a joint action have to give their permission for the action to occur.

In the next sections several examples illustrate the way these morphisms can be used for establishing interconnections and the way diagrams in the category ***DSGN*** define systems configurations. The semantics of a system configuration is given by a categorical construction: the colimit of the underlying diagram. Taking the colimit of a diagram collapses the configuration into an object by internalising all the interconnections and distribution aspects, thus delivering a design for the system as a whole. Colimits in CommUnity capture a generalised notion of parallel composition in which the designer makes explicit what interconnections are used between components:

- channels and locations involved in each i/o-communication established by the configuration are amalgamated;
- every set $\{g_1,\ldots,g_n\}$ of actions that are synchronised is represented by a single action $g_1\|\ldots\|g_n$ whose occurrence captures the joint execution of the actions in the set. The transformations performed by the joint action are distributed over the locations of the synchronised actions. Each located action $g_1\|\ldots\|g_n@l$ is specified by the conjunction of the specifications of the local effects of each of the synchronised actions g_i that is distributed over l, and the guards of joint actions are also obtained through the conjunction of the guards specified by the components.

3 Effectiveness of Coordination Connectors

In *structural models* of Software Architecture, i.e., models that share the view that system architectures are structured in terms of components and coordination connectors, the properties of the physical distribution topology of locations and communication links are not usually taken into account. It is assumed that the physical links ("wires") that enable communication between hosts in the underlying communication network are fixed and statically determined. Consequently, these models rely on the fact that the coordination mechanisms put in place through connectors can be made effective across the wires that link the components' hosts.

When mobility comes to play, for instance in contexts where physical mobility of computation hosts, such as laptops, exists, the wired or wireless physical links that enable communication between hosts can change. Furthermore, the ability of a com-

ponent to communicate with others is influenced by its location because of the barriers to communication that are erected in communication networks by system managers. Finally, the communication network is also subject to frequent changes.

In this section, we use an example of a client-server system to show that when the distribution aspects of systems are addressed at the architectural level, it is necessary to take into account the properties of the network communication infrastructure in order to understand whether the coordination connectors in place are effective or have to be replaced with ones that are compatible with the topology of distribution available at physical level.

For developing the example, we use the CommUnity architectural framework. In particular, we shall made use of distribution connectors that were proposed in [11], through which the mechanisms that define the distribution topology of systems can be externalised and explicitly represented in system architectures.

We consider a very simple client-server system. As usual in this kind of systems, the server exports a service that the client may request at some point in its execution. The service is the calculation of *f(v,x)* for a certain function *f,* where *v* is a resource local to the server and *x* is a value given by the client.

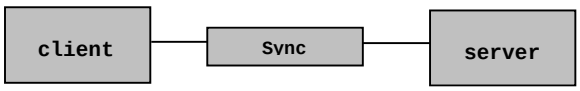
In CommUnity, clients and servers can be modelled as follows.

```
design server is
in   x: T
out  r: T
prv  v: T', lx: T, s: [0..2]
do   req: [s=0→lx:=x‖s:=1]
[]   serv:[s=1→r:=f(v,lx)‖s:=2]
[]   ret: [s=2→s:=0]
[]   chg: [true→v:∈T']
```

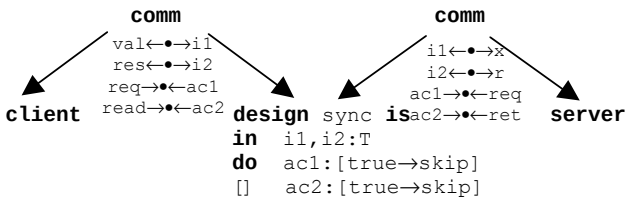
```
design client is
in    res: T
out   val: T
prv   rq, inp: bool
do    req:[¬rq∧ inp→rq:=true‖
                          inp:=false]
[]    read: [rq → rq:=false]
[] prv prod:[¬inp→inp:=true‖val:∈T]
```

In *server,* it is modelled that a server repeatedly accepts requests, executes the service and returns the result. Moreover, its resource *v* may be updated at any time through the execution of action *chg*. In *client*, the typical behaviour of a client is modelled, ignoring for the moment the details of its internal computation: First, it produces the data needed for the service, then it requests the service and, finally, it reads the result.

It remains to describe the way the client and the server interact. There are several possibilities; we start by considering the exchange of messages through synchronous communication. In other words, the client and the server must synchronise first for the transmission of the request's data and then again for the transmission of the result.



In order to achieve this form of interaction, we have to make explicit the synchronisation of the actions *req* of *clien*t and *server*, the synchronisation of actions *read* of *client* and *ret* of *server,* and the i/o interconnection of the channels used for the transmission of the service's parameter and result. Such interconnections can be described by the following diagram in the category ***DSGN***,



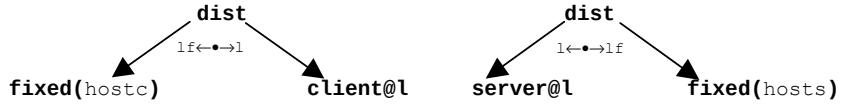
where *comm* consists of two input variables to model the medium through which data is to be transmitted between the client and the server, and two shared actions for the two components to synchronise in order to transmit the data. Because names in CommUnity are local, the identities of the shared input variables and the shared actions in *comm* are not relevant: they are just placeholders for the projections to define the relevant bindings. Hence, we normally do not bother to give them explicit names, and represent them through the symbol •.

This architectural description of the client-server system abstracts away how the system is supposed to be distributed on the nodes of a network (every design considered so far is location-unaware) and takes for granted that the synchronous communication between the client and the server can be made effective. This can be easily confirmed by analysing the design of the system as a whole, obtained by taking the colimit of its configuration diagram. Such design has, for instance, the joint action

$$\texttt{req|req:[}\neg rq \wedge inp \wedge s=0 \rightarrow \texttt{rq:=true}\|\texttt{inp:=false}\|\texttt{lx:=val}\|\texttt{s:=1]}$$

that models the synchronous execution of actions *req* of *client* and *server*. This action is enabled provided that enabling conditions of both actions evaluate to true. Nothing else is required, even though the execution of *req* by *server* involves the reading of *val*, a resource local to the client.

Let us now suppose that the server exists at a fixed location — *hosts*, which is a node in a subnet protected by a firewall. That is to say, there exists a *filter* node and every external communication with *hosts* has to pass through it. If the *client* exists in a node *hostc* that is not in this subnet, then it is not realistic to expect that the coordination of the client and the server put in place through the connector *Sync* is effective. Such distribution aspects of the client-server system can be explicitly modelled as follows,



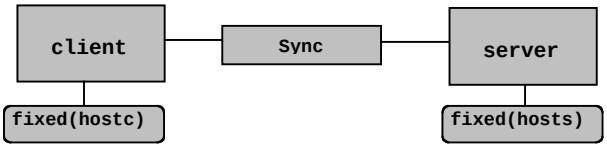
where

- *fixed(hostc)* and *fixed(hosts)* are instances of a parameterised design *fixed(v:Loc)* that consists of an output location named *lf* that is constrained to be initialised at configuration time (i.e. when this component is included in the configuration of the system being built) with a value denoted by *v*.
- *client@l* (resp., *server@l*) denotes the design *client* (resp., *server*) augmented with an input location variable *l*, where every action and local channel of *client* (resp., *server*) is located.

In this way, it is specified that components *client* and *server* are bound statically to the network nodes identified by *hostc* and *hosts*, respectively. In what concerns the connector *Sync*, namely its glue, our design decision was to keep it location-transparent. This choice is justified by the fact that *sync* does not perform any computation but simply provides a pure coordination function just like an ideal, neutral "cable". In the system architecture, this does not need to be specified because every constituent of a design is by default considered to be located at the distinguished location variable λ.

It is assumed that *hosts, hostc* and *filter* are constants of type *Loc*. This data type in this case has no other requirements that to have these constants and the axioms *hosts≠hostc, hosts≠filter, hostc≠filter*, that ensure that the three hosts are actually different.

By adopting a simple box-and-line notation, the architecture of the system can be represented below. Notice that, in this architecture, we achieved the envisaged separation between the individual software components of the system, the coordination connector through which they interact, and the mechanisms that are responsible for the distribution topology of the system.



A design of the system as a whole, where all interconnections and distribution aspects are internalised (obtained by taking the colimit of the underlying diagram) is given by *client-sync-server*. This design exposes clearly the system dependence on the properties of the communication infrastructure that is available at physical levels, namely the links that connect *hostc* and *hosts* in the underlying network.

```
design client-sync-server is
outloc lc, ls
inv    lc=chost ∧ ls=shost
out    val@lc, res@ls: T
prv    req@lc, inp@lc: bool, v@ls: T', lx@ls: T, s@ls: [0..2]
do     req|req@lc:[¬rq∧inp →rq:=true‖ inp:=false]
              @ls:[s=0 → lx:=val‖s:=1]
[]     serv@ls:[s=1→res:=f(v,lx)‖s:=2]
[]     read|ret@lc:[rq→rq:=false]
[]              @ls:[s=2→s:=0]
[] prv prod@lc:[¬inp →inp:=true‖val:∈T]
[]     chg@ls:[true→v:∈T']
```

In the situation described previously, *hosts* is in a subnet protected by a firewall and every external communication to *hosts* has to pass through *filter* node. This means that the current relation *be in touch* has the following property:

$$\textit{for every } n \in U_{Loc},\ \textit{if } (hosts_U \ \mathbf{bt}\ n) \textit{ then } n = \bot_U \textit{ or } n \in FW \textit{ or } n = filter_U$$

where $FW \subseteq U_{Loc}$ represents the set of nodes of the subnet protected by the firewall. Given that $hostc_U \notin FW$, we have that $\neg(hosts_U \ \mathbf{bt}\ hostc_U)$ and, hence, we may conclude that, in *client-sync-server*, once the client produces the data for the service, the issue of the request gets blocked and the system enters a deadlock. This clearly con-

firms that the connector in place in the system is currently not effective and that a connector compatible with the current distribution topology has to be used instead. Moreover, it points out the importance of being able to make systems evolve through dynamic reconfiguration in response to changes in the topology of the network.

4 New Patterns of Coordination

In the context of the separation of computation and coordination supported by architecture-based approaches, the distributed and mobile dimension of systems is also reflected on the kind of the coordination constructs that are appropriate and should be available for component interconnection.

On the one hand, the advent of mobility calls for new abstractions for component interaction that facilitate systems design. The fact that components may move across a network of locations demands the adoption of coordination styles that are location-dependent. *Transient interaction*, for instance, is a form of interaction that is conditional on the relative positions of components. More precisely, interaction is limited to the situations in which the components are in the communication range of each other.

On the other hand, the opportunities made available by technological developments have been explored and have led to new programming paradigms and new conceptual mechanisms for structuring systems. For instance, mobile code paradigms such as *Remote Evaluation* and *Mobile Agents* became very popular in the design of distributed applications. As mentioned in [14], coordination no longer involves just communication, it may also involve the management of the locations of the coordinated parties.

Our aim in this section is to show, by means of examples, that at an architectural level of design, such new abstractions for coordination can be formally specified as connectors and used in systems architectures together with the connectors that model traditional coordination primitives. Furthermore, we shall also show how in situations in which coordination encompasses both communication and relocation, the design of these aspects can be carried out independently.

When Coordination is Location-dependent

Transient variable sharing is a context-dependent pattern of interaction proposed in [13] as a variant of traditional variable sharing, which is suitable for mobile computing systems that are subject to frequent disconnections. Roughly speaking, it is based on variable sharing limited to the situations in which the components are in the communication range of each other.

In CommUnity, read-only sharing of variables is supported through I/O interconnection of variables. For instance, the interconnection of an input variable y of design R with the output variable x of design W establishes that the value of y is read from x. If, for instance, W is a mobile component, the reading of variable x is conditioned by the "connectivity" between W and R. The semantics defined for CommUnity designs

ensures that, when *W* is out of the range of communication of *R,* every action of *R* that needs to access the value of *y* gets blocked.

In read-only transient sharing, even while *W* is out of the range of communication of *R* the value of *y* can be accessed. Of course, in this situation there is no way to continue to guarantee that the value of *y* is given by *x*. Instead, the value of *y* is given by the value of *x* on disconnection.

This form of transient sharing can be modelled through the binary coordination connector *TranSh* with roles *writer* and *reader* and the glue *transh*.

```
design writer is
inloc lw
out   x@lw: T
do    wr@lw:[true→ x:∈T]
```

```
design reader is
inloc lr
in    y: T
do    read@lr:[true→skip]
```

The roles define the behaviour required of the components to which the connector can be applied. For a *writer,* we require an action that models every kind of possible operation on *x*. For a *reader,* we require an action that models the access to the input variable *y*. This is because it is essential to know in which location this action is executed.

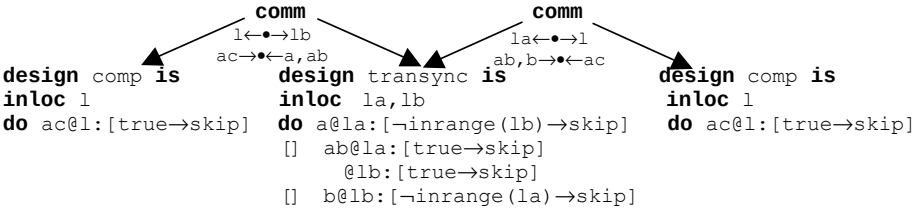
The glue ensures that updates to *x* are propagated to *y* whenever the *reader* and the *writer* are in contact with each other. Whenever the communication between the two components is possible, *transh* prevents the *writer* from writing *x* before the previous change of *x* has been propagated to *y*. In the other situations, *lr* is not in the range of *lw* and, hence, *y* remains with the value of *x* at disconnection time. On re-connection, the value of *x* is sooner or later propagated to *y*. This is achieved through the execution of the action *auto* that is private to *transh* and, hence, subject to fairness requirements.


```
design transh is
inloc lw,lr
in    x:T
out   y@lr:T
prv   chg@lw:bool
do    write@lw:[¬chg→chg:=true]
[]    update@lr:[chg→y:=x||chg:=false]
[]    ignore@lr:[¬inrange(lw)→chg:=false]
[] prv auto@lr:[y≠x→y:=x]
```

Synchronous execution of actions is another form of communication available in many models of distributed systems that has also inspired a transient counterpart [13]. A simple form of transient synchronisation of an action *a* with an action *b* consists of requiring the two actions to be executed synchronously whenever they are located at connected hosts. When this is not the case, the two actions can be executed independently.

This form of transient synchronisation can be represented by a binary connector with two identical roles — *comp*. This design simply identifies the action subject to the synchronisation. The glue of the connector ensures the synchronous execution of

the two actions whenever they are located in locations in contact with each other and allows their independent execution in the other situations.



When Coordination requires Movement

The fact that the location of components becomes a first-class design element also leads to new coordination patterns that explore the relocation of components, namely *Remote Evaluation (RE)* [15]. Like transient sharing and transient synchronisation, *RE* is a variant of the traditional client-server style of interaction.

Let us consider again the client-server system described in section 3. Recall that the server offers a fixed service — the calculation of a certain function *f*, which depends on some resource *v* that is local to the server. The server holds the know-how needed to use the resource (the code that implements function *f*) as well as the resource involved in the service (the variable *v*). In *RE*, the server offers its resources but it is the client that holds the code that describes how to perform the service. The client needs to transmit the data and must provide the server with the code that implements the service.

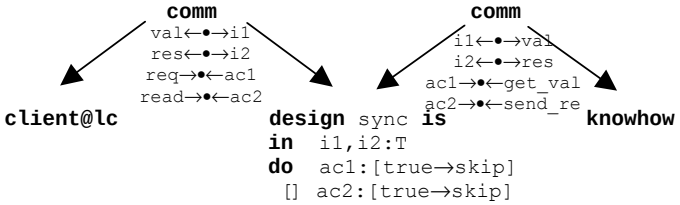
In this situation, the server simply has to be ready to cooperate with the client in the execution of the service. It receives the request and then it allows the use of its local resource, which can be locally changed at any time through the execution of action *chg*. We further consider that the server is bound to the same location for it whole life (see design *re-server*).

```
design re-server is
outloc l
out   v@l: T'
prv   s@l: [0..2]
do    req@l: [s=0→s:=1]
[]    serv@l: [s=1→s:=0]
[]    chg@l: [true→v:∈T']
```

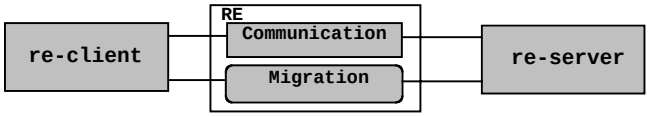
```
design knowhow is
inloc l
in    v: T', val:T
out   res@l: T
prv   lval@l:T, t@l: [0..3]
do    get_val@l: [t=0→lval:=val‖t:=1]
[]    req@l: [t=1→t:=2]
[]    serv@l: [t=2→res:=f(v,lval)‖t:=3]
[]    send_res@l: [t=3→t:=0]
```

The client in this case is slightly more complex because a part of it concerns the know-how needed to use the server's resource and has to be relocated in order to achieve local interaction with the server. The simpler way of describing *re-client* is in terms of a configuration involving a traditional client bound to a fixed location and a design *knowhow* that encapsulates the portion of *re-client* that has to be moved. As modelled above, *knowhow* repeatedly gets the data needed for the execution of the service, requests the use of the server resource, uses the resource for the calculation of *f* and returns the result. The configuration of *re-client* (see below) establishes that

knowhow and *client* exchange messages (service's data and result) through synchronous communication.

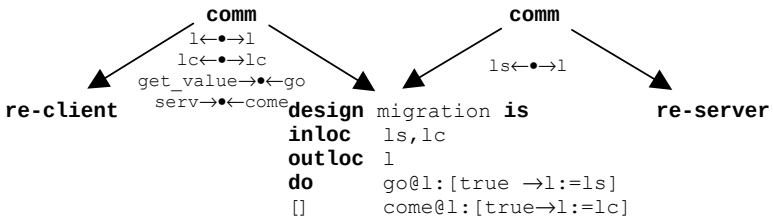


In *RE*, the coordination of *re-client* and *re-server* involves communication and code migration. It is necessary to describe the movement of the subcomponent *knowhow* of *re-client* that makes local interaction with the server possible and, once there, the kind of communication it is used. These two dimensions of coordination can be addressed and described separately.



In what concerns communication, *RE* does not entail a particular form communication. Message passing through synchronous communication or remote procedure call are, for instance, possible choices. Hence, we use the connector *Communication*, which models a generic bi-directional communication protocol in the sense that the glue of this connector leaves completely unspecified the way in which messages are processed and transmitted. In this way, *RE* can be regarded as a parameterised entity that takes the communication protocol as a parameter (similar to higher-order connectors defined in [12]).

The pattern of migration of *RE* can be defined separately by the connector *Migration* below.



The glue carries out the relocation of *knowhow* in the server location as soon as it gets the data needed for the service. Once *knowhow* has terminated the use of the server resources, it returns to the client location. Then, the client may collect the result. Notice that this corresponds to the synchronous execution of actions *send_res* at *knowhow* and *read* at *client*. At this point, these two actions are co-located (i.e., *lc* and *l* have the same value) and hence the location-guards of *send_res* and *read* are both true.

Finally, by putting together the glues of *Migration* and *Communication* (without any kind of interaction) as well as the underlying role connections, we obtain a description of *RE* as a coordination connector.

5 Concluding Remarks

In this paper, we have addressed the interference of distribution and mobility with coordination in the context of system architectures that are structured in terms of components and connectors. By adopting an architectural framework that was extended in order to support the description of the distribution aspects of systems, we have shown how the effectiveness of connectors in place in a system may depend on the locations of the components that are being coordinated as well as on the properties of the wires through which the coordination has to take place. In this situation, it is essential that when connectors in place become ineffective, they can be replaced by different ones. Support for describing this kind of evolution can be conceived in terms of a dynamic reconfiguration language equipped with a set of observables including, for instance, the component locations and the network topology (many other relevant observables are identified in [3]). We plan to extend the reconfiguration language developed for CommUnity architectures [17] in order to make systems evolve in reaction to these new sources of change.

Furthermore, we have focused on new styles of coordination that rely on and use the distribution/mobility dimension of systems. We have considered situations in which the coordination patterns depend on the location of the coordinated parties or even determine their relocation. We have shown that these new forms of coordination, like the traditional ones, do not have to be programmed in the code of components. They can be externalised as first class entities and superposed over other components to regulate their behaviour or their location without intruding in the way they have been implemented. In fact, components will not even know that they are being regulated in the sense that the coordination is performed externally.

The examples we have analyzed also suggest that in the situations in which communication, distribution and mobility, can be regarded as different dimensions of the coordination pattern, these dimensions can be addressed separately and then composed. This separation is important not only because it makes it easier to describe complex interaction patterns and promotes reuse, but also addresses the evolutionary dimension of systems.

Not every language supports the separation of concerns just described. CommUnity was developed precisely to illustrate how the separation between computation and coordination can be supported by Formal Description Techniques. Moreover, the extension with distribution and mobility (in contrast with a former extension presented in [16]) was developed having in mind the support required for the externalisation of the mechanisms that are responsible for managing the distribution topology of systems. In [7], we have provided a mathematical characterisation of the language features that the separation of computation and coordination requires. Future work is going on to establish the corresponding extension taking into account distribution dimension.

Acknowledgements. This work was partially supported through the IST-2001-32747 Project *AGILE – Architectures for Mobility*.

References

[1] R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM*, 6(3), 1997, 213-249.

[2] L.Bass, P.Clements and R.Kasman, *Software Architecture in Practice*, Addison Wesley 1998.

[3] L.Cardelli and A.Gordon, "Mobile Ambients", in Nivat (ed), *FoSSACs'98*, LNCS 1378, 140-155, Springer-Verlag, 1998.

[4] K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley 1988.

[5] G.Cugola, C.Ghezzi, G.P.Picco and G.Vigna, "Analyzing Mobile Code Languages", J.Vitek and C.Tschudin, eds., *Mobile Object Systems*, LNCS 1222, Springer-Verlag, 1997.

[6] J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming* 28, 111-138, 1997.

[7] J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a signature?", in *AMAST'98*, A.Haeberer (ed), Springer-Verlag 1999.

[8] N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.

[9] J.L.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors", submitted, available at www.fiadeiro.org/jose/papers.

[10] S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2), 1993, 337-356.

[11] A.Lopes, J.L.Fiadeiro and M.Wermelinger, "Architectural Primitives for Distribution and Mobility", *SIGSOFT 2002/FSE-10*, 41-50, ACM Press, 2002.

[12] A.Lopes, J.L.Fiadeiro and M.Wermelinger, "A Compositional Approach to Connectors Construction", *Recent Trends in Algebraic Development Techniques – 14th Workshop on ADTs*, LNCS 2267, 201-220, Springer-Verlag, 2001.

[13] G.-C.Roman, P.J.McCann and J.Y.Plun, "Mobile UNITY: reasoning and specification in mobile computing", *ACM TOSEM*, 6(3),250-282, 1997.

[14] G.-C.Roman, A.L.Murphy and G.P.Picco, "Coordination and Mobility", in A.Omicini et al (eds), Coordination of Internet designs: Models, Techniques, and Applications, 253-273, Springer-Verlag, 2001.

[15] J.Stamos and D.Gifford, "Remote Evaluation", *ACM TOPLAS* 12 (4),537-565, 1990.

[16] M.Wermelinger and J.Fiadeiro, "Connectors for Mobile Programs", *IEEE TOSE* 24(5), 331-341, 1998.

[17] M.Wermelinger, A.Lopes and J.Fiadeiro, "A Graph Based Architectural (Re)configuration Language", *Proc. 8th ESEC/9th FSE*, 21-32,ACM Press, 2001.