Global Computing in a Dynamic Network of Tuple Spaces[⋆]

Rocco De Nicola[1], Daniele Gorla[2,⋆⋆], and Rosario Pugliese[1]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze
[2] Dipartimento di Informatica, Università di Roma "La Sapienza"
{denicola, pugliese}@dsi.unifi.it
gorla@di.uniroma1.it

Abstract. We present a calculus inspired by KLAIM whose main features are: explicit process distribution and node interconnections, remote operations, process mobility and asynchronous communication through distributed tuple spaces. We first introduce a basic language where connections are reliable and immutable; then, we enrich it with two more advanced features for global computing, i.e. *failures* and *dynamically evolving connections*. In each setting, we use our formalisms to specify some non-trivial global computing applications and exploit the semantic theory based on an observational equivalence to equationally establish properties of the considered case-studies.

1 Introduction

Programming computational infrastructures available globally for offering uniform services has become one of the main issues in Computer Science. The challenges come from the necessity of dealing at once with issues like communication, co-operation, mobility, resource usage, security, privacy, failures, etc. in a setting where demands and guarantees can be very different for the many different components. A key issue is the definition of innovative theories, computational paradigms, linguistic mechanisms and implementation techniques for the design, realization, deployment and management of global computational environments and their application.

On the linguistic side, we believe that a language for global computing should be equipped with primitives that support *network awareness* (i.e. locations can be explicitly referenced and operations can be remotely invoked), *disconnected operations* (i.e. code can be moved from one location to the other and remotely executed), *flexible communication mechanisms* (like distributed repositories [11, 8, 15] storing content addressable data), and *remote operations* (like asynchronous remote communications). On the foundational side, the demand is on the development of tools and techniques to

[⋆] This work has been partially supported by EU FET - Global Computing initiative project MIKADO IST-2001-32222. The funding bodies are not responsible for any use that might be made of the results presented here.

[⋆⋆] Most of this work was carried on while the second author was a PhD student at the University of Florence.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 157–172, 2005.

build safer and trustworthy global systems, to analyse their behaviour, and to demonstrate their conformance to given specifications. Clearly, such semantic theories should reflect all the above listed distinctive features of global systems.

In this paper, we introduce a foundational language that retains the main features of Klaim [12] (explicit distribution, remote operations, process mobility and asynchronous communication through distributed data spaces), but extends it with new constructs (somehow inspired by [4]) to flexibly model the interconnection structure underlying a network. The resulting formalism, called tKlaim (*topological* Klaim), permits to explicitly model inter-node connections and to establish and remove them dynamically. Connections are then essential to enable tKlaim remote operations: such an operation can be performed only if the node where it is executed and that on which it acts upon are directly connected. Routing algorithms are then needed to enable remote operations between nodes that are not directly connected.

tKlaim takes its origin from two formalisms with opposite objectives. On one hand, we have the programming language X-Klaim [3], a full fledged programming language for global computers based on Klaim; on the other hand, we have the π-calculus [22], the generally recognised minimal common denominator of calculi for mobility. By following well-established techniques for the π-calculus, in a companion paper [13] we formally develop the semantic theory of tKlaim. Here, we use such a theory to state and prove properties of some meaningful global computing applications. Given the direct correspondence of tKlaim with X-Klaim, we believe that the behavioural study carried on at the level of the calculus can be faithfully transposed at the level of the programming language to let programs run in a controlled way on an actual global computer, like the Internet.

To softly introduce the reader to our language, we start in Section 2 by presenting a very basic model where inter-node connections are explicitly programmable but fixed at the outset. This scenario is very close to LANs, where physical connections are reliable and immutable (or change very rarely). We then present two variations of this basic formalism. In Section 3, we enrich the language with different forms of failures, another key feature of global computers. We start with a scenario where only nodes and node components (i.e., data or processes) can fail and use it to establish soundness of a distributed fault-tolerant protocol, the *'k-set agreement'* [10]; then, we briefly present a way to also encompass link failures. The second variation of the basic framework is in Section 4, where links can be dynamically changed by processes. The use of the language with both link failures and dynamic connections is exemplified by programming two routing scenarios and by establishing their soundness.

Section 5 concludes the paper with a discussion on related work. In all the examples, properties of the proposed case-studies are formulated and proved by exploiting *may testing* [14], an intuitive observational equivalence. Our proofs rely on a tractable (bisimulation-based) proof technique whose definition has been omitted from this paper for the sake of space and can be found in [13].

2 The Language

Syntax. The syntax of tKlaim, given in Table 1, is parameterised with respect to the following syntactic sets, which we assume to be countable and pairwise disjoint: $\mathcal{L}$, of

Table 1. Syntax of tKlaim

Nets:
$N ::= \mathbf{0} \mid l :: C \mid \{l_1 \leftrightarrow l_2\} \mid (\nu l)N \mid N_1 \| N_2$

Components:
$C ::= P \mid \langle t \rangle \mid C_1 \mid C_2$

Processes:
$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid X \mid \mathbf{rec}\, X.P$

Tuples:
$t ::= e \mid \ell \mid t_1, t_2$

Actions:
$a ::= \mathbf{in}(T)@\ell \mid \mathbf{read}(T)@\ell \mid \mathbf{out}(t)@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{new}(l)$

Templates:
$T ::= e \mid !\, x \mid \ell \mid !\, u \mid T_1, T_2$

Expressions:
$e ::= V \mid x \mid \ldots$

localities, ranged over by l; $\mathcal{U}$, of *locality variables*, ranged over by u; $\mathcal{V}$, of *basic values*, ranged over by V; $\mathcal{Z}$, of *basic variables*, ranged over by x; $\mathcal{X}$, of *process variables*, ranged over by X. Finally, ℓ is used to denote elements of $\mathcal{L} \cup \mathcal{U}$.

The exact syntax of *expressions*, e, is deliberately omitted; we just assume that expressions contain, at least, basic values and variables. *Localities*, l, are the addresses (i.e. network references) of nodes. *Tuples*, t, are sequences of expressions, localities or locality variables. *Templates*, T, are used to select tuples: in particular, $!\, x$ and $!\, u$, that we call *formal fields*, are used to bind variables to values.

Processes, ranged over by $P, Q, R, \ldots$, are the tKlaim active computational units and may be executed concurrently either at the same locality or at different localities. They are built up from the terminated process **nil** and from the basic actions by using prefixing, parallel composition and recursion. *Actions* permit removing/accessing/adding tuples from/to tuple spaces (actions **in**/**read**/**out**, resp.), activating new threads of execution (action **eval**) and creating new nodes (action **new**). Action **new** is not indexed with an address because it always acts locally; all the other actions explicitly indicate the (possibly remote) locality where they will take effect.

Nets, ranged over by $N, M, \ldots$, are finite collections of nodes and inter-node connections. A *node* is a pair $l :: C$, where locality l is the address of the node and C is the (parallel) component located at l. *Components*, ranged over by $C, D, \ldots$, can be either processes or data, denoted by $\langle t \rangle$. *Connections*, or *links*, are pairs of node addresses $\{l_1 \leftrightarrow l_2\}$ stating that the nodes with address l_1 and l_2 are directly (and bidirectionally[1]) linked. In $(\nu l)N$, name l is private to N; the intended effect is that, if one considers the term $M \,\|\, (\nu l)N$, then locality l of N cannot be referred from within M.

Names (i.e. localities and variables) occurring in tKlaim processes and nets can be *bound*. More precisely, prefixes $\mathbf{in}(T)@\ell.P$ and $\mathbf{read}(T)@\ell.P$ bind T's formal fields in P; prefix $\mathbf{new}(l).P$ binds l in P, and, similarly, net restriction $(\nu l)N$ binds l in N; finally, $\mathbf{rec}\, X.P$ binds X in P. A name that is not bound is called *free*. The sets $fn(\cdot)$ and $bn(\cdot)$ (respectively, of free and bound names of a term) are defined accordingly. The set $n(\cdot)$ of names of a term is the union of its sets of free and bound names. We say that two

[1] For the sake of simplicity, we assumed bidirectional links; nevertheless, all the theory and the examples we develop here could be tailored to the framework where links are directed.

terms are *alpha-equivalent* if one can be obtained from the other by renaming bound names. In the sequel, we shall work with terms whose bound names are all distinct and different from the free ones. Moreover, as usual, we shall only consider *closed* terms, i.e. processes and nets without free variables.

Notation 1. We write $A \triangleq W$ to mean that A is of the form W; this notation is used to assign a symbolic name A to the term W. We shall use notation $\widetilde{\cdot}$ to denote sets of objects (e.g. $\widetilde{l}$ is a set of names). We shall sometimes write $\mathbf{in}()@l$, $\mathbf{out}()@l$ and $\langle\rangle$ to mean that the argument of the actions or the datum are irrelevant. Finally, we shall omit trailing occurrences of process **nil** and write $\Pi_{j\in J}\ W_j$ for the parallel composition (both '|' and '||') of terms (components or nets, resp.) W_j.

Operational Semantics. tKlaim operational semantics is given in terms of a structural congruence and a reduction relation. The *structural congruence*, ≡, identifies nets which intuitively represent the same net. It is inspired to π-calculus's structural congruence (see, e.g., [25]) and includes laws stating that '||' is commutative, associative and has **0** as identity element, laws equating alpha-equivalent nets, laws regulating commutativity of restrictions, and laws allowing to freely fold/unfold recursive processes. Moreover, the following laws are crucial to our setting:

$$\begin{array}{lll}
\text{(Clone)} & \text{(Self)} & \text{(BiDir)} \\
l :: C_1|C_2 \;\; l :: C_1 \parallel l :: C_2 & l :: \mathbf{nil} \;\equiv\; \{l \leftrightarrow l\} & \{l_1 \leftrightarrow l_2\} \;\equiv\; \{l_2 \leftrightarrow l_1\} \\
\\
\text{(RNode)} & \text{(Ext)} & \\
(\nu l)N \equiv (\nu l)(N \parallel l :: \mathbf{nil}) & N \parallel (\nu l)M \equiv (\nu l)(N \parallel M) \;\; \text{if } l \notin fn(N) &
\end{array}$$

Law (Clone) turns a parallel between co-located components into a parallel between nodes; law (Self) states that nodes are self-connected; law (BiDir) states that links are bidirectional; law (Ext) is the standard π-calculus' rule for scope extension. Finally, law (RNode) states that any restricted name can be used as the address of a node; indeed, we consider restricted names as private network addresses, whose corresponding nodes can be activated and deactivated when needed. By relying on rule (RNode), we shall only consider nets where each bound name is associated to a node.

The reduction relation is given in Table 2 and relies on two auxiliary functions: $\mathcal{E}[\![\_]\!]$ and $match(\_;\_)$. The *tuple/template evaluation* function, $\mathcal{E}[\![\_]\!]$, evaluates componentwise the expressions occurring within the tuple/template $\_$; its definition is simple and, thus, omitted. The *pattern matching* function, $match(\_;\_)$, verifies the compliance of a tuple w.r.t. a template and associates values to variables bound in the template. Intuitively, a tuple matches a template if they have the same number of fields, and corresponding fields match. Formally, function *match* is defined as

$$\begin{array}{lll}
match(l;l) = \epsilon & match(!u;l) = [^l/_u] & \dfrac{match(T_1;t_1) = \sigma_1 \quad match(T_2;t_2) = \sigma_2}{match(T_1,T_2;t_1,t_2) = \sigma_1 \circ \sigma_2} \\
match(V;V) = \epsilon & match(!x;V) = [^V/_x] &
\end{array}$$

where we let 'ϵ' be the empty substitution and '$\circ$' denote substitutions composition. Here, a substitution σ is a mapping of localities and basic values for variables; $P\sigma$ denotes the (capture avoiding) application of σ to P.

Table 2. tKlaim Reduction Relation

$$(\text{R-Out})\quad \frac{\mathcal{E}[\![\, t \,]\!] = t'}{l :: \mathbf{out}(t)@l'.P \parallel \{l \leftrightarrow l'\} \;\longmapsto\; l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t' \rangle}$$

$$(\text{R-Eval})\quad l :: \mathbf{eval}(P_2)@l'.P_1 \parallel \{l \leftrightarrow l'\} \;\longmapsto\; l :: P_1 \parallel \{l \leftrightarrow l'\} \parallel l' :: P_2$$

$$(\text{R-In})\quad \frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l :: \mathbf{in}(T)@l'.P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle \;\longmapsto\; l :: P\sigma \parallel \{l \leftrightarrow l'\}}$$

$$(\text{R-Read})\quad \frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l :: \mathbf{read}(T)@l'.P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle \;\longmapsto\; l :: P\sigma \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle}$$

$$(\text{R-New})\quad l :: \mathbf{new}(l').P \;\longmapsto\; (\nu l')(l :: P \parallel \{l \leftrightarrow l'\})$$

$$(\text{R-Par})\quad \frac{N_1 \;\longmapsto\; N_1'}{N_1 \parallel N_2 \;\longmapsto\; N_1' \parallel N_2} \qquad\qquad (\text{R-Res})\quad \frac{N \;\longmapsto\; N'}{(\nu l)N \;\longmapsto\; (\nu l)N'}$$

$$(\text{R-Struct})\quad \frac{N \equiv M \quad M \longmapsto M' \quad M' \equiv N'}{N \;\longmapsto\; N'}$$

The operational rules of tKlaim can be briefly motivated as follows. Rule (R-Out) evaluates the expressions within the argument tuple and sends the resulting tuple to the target node. However, this is possible only if the source and the target nodes are directly connected. Rule (R-Eval) is similar: a process can be spawned at l' by a process running at l only if l and l' are directly connected. Rules (R-In) and (R-Read) require existence of a matching datum in the target node and of a connection between the source and the target node. The tuple is then used to replace the free occurrences of the variables bound by the template in the continuation of the process performing the actions. With action **in** the matched datum is consumed while with action **read** it is not. Rule (R-New) says that execution of action **new**(l') adds a restriction over l' to the net and a link between the creating node l and the created node l'; from then on, a new node with locality l' can be activated/deactivated by using law (RNode). Rules (R-Par), (R-Res) and (R-Struct) are standard.

tKlaim adopts a Linda-like [18] communication mechanism: data are anonymous and associatively accessed via pattern matching, and communication is asynchronous. Indeed, even if there exist prefixes for placing data to (possibly remote) nodes, no synchronization takes place between (sending and receiving) processes, because their interactions are mediated by nodes, that act as data repositories.

To conclude the presentation of tKlaim operational semantics, we want to stress that interactions between directly linked nodes can be used to permit interactions between nodes that are not directly linked. However, as it happens in practice, this feature must be explicitly programmed. If a process running at l wants to send a tuple t to l' and there exists a path of links from l to l' in the underlying connection graph, the one needs a mobile process be spawned by l 'towards' l' that delivers tuple t. The main challenges of such a process is to discover the shortest (or, at least, a possible) path connecting l and l'. This functionality can be accomplished by relying on *routing tables*, i.e. distributed data structures that record information on routing paths. We leave the formal specification of this process, together with a proof of its soundness, for a forthcoming full paper.

Observational Semantics. We now present a preorder on tKlaim nets yielding sensible semantic theories. We follow the approach put forward in [14] and use *may testing* preorder and the associated equivalence. Intuitively, two nets are may testing equivalent if they cannot be distinguished by any external observer taking note of the data offered by the observed net. More precisely, an *observer O* is a net containing a node whose address is a reserved locality name `test`. A computation reports *success* if, along its execution, a datum at node `test` appears; this is written $\overset{OK}{\Longrightarrow}$.

Definition 2 (May Testing Preorder and Equivalence). May testing preorder, $\sqsubseteq$, *is the least preorder on* tKlaim *nets such that, for every* $N \sqsubseteq M$, *it holds that* $N \parallel O \overset{OK}{\Longrightarrow}$ *implies* $M \parallel O \overset{OK}{\Longrightarrow}$, *for any observer O.*

May testing equivalence, $\simeq$, *is defined as the intersection of* $\sqsubseteq$ *and* $\sqsupseteq$.

To conclude, we give a simple Proposition collecting a few equational laws that will simplify the proofs of the case-studies considered in this paper. Soundness of such laws can be easily established by exploiting the co-inductive (bisimulation-based) proof technique provided in [13]. Indeed, directly establishing may-testing may be very hard because of the universal quantification over contexts.

Proposition 1.

1. $l :: \mathbf{out}(t)@l'.P \parallel \{l \leftrightarrow l'\} \;\simeq\; l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle$
2. $l :: \mathbf{eval}(Q)@l'.P \parallel \{l \leftrightarrow l'\} \;\simeq\; l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: Q$
3. $(\nu l')(l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle) \;\simeq\; (\nu l')(l :: P\sigma)$ if $match(\mathcal{E}[\![T]\!]; t) = \sigma$
4. $(\nu l)(l :: C) \;\simeq\; \mathbf{0}$ whenever C is a datum $\langle t \rangle$, a stuck process **nil** or the parallel composition of such components
5. $l :: \mathbf{new}(l').P \;\simeq\; (\nu l')(l :: P \parallel \{l \leftrightarrow l'\})$
6. $(\nu l')\{l \leftrightarrow l'\} \;\simeq\; l :: \mathbf{nil}$

3 Modelling Failures

We now enrich the basic framework with a mechanism for modelling various forms of failures, a key feature of global computers. We start by modelling failure of nodes and of node components; then, we use the new setting to prove some properties of a distributed fault-tolerant protocol. Finally, we sketch a minor modification of our framework to take into account link failures.

3.1 Failure of Nodes and Node Components

We start by modelling a framework where only nodes and node components fail. This can be achieved by adding the operational rule

$$\text{(R-F}\textsc{ail}\text{N)} \qquad l :: C \;\longmapsto\; \mathbf{0}$$

that models corruption of data (*message omission*) if $C \triangleq \langle t_1 \rangle | \ldots | \langle t_n \rangle$, node (*fail-silent*) failure if $l :: C$ collects all the clones of l, and abnormal termination of some processes running at l otherwise. Modelling failures as disappearance of a resource (a datum, a process or a whole node) is a simple, but realistic, way of representing failures in a global computing scenario [6]. Indeed, while the presence of data/nodes can be ascertained, their absence cannot because there is no practical upper bound to communication delays. Thus, failures cannot be distinguished from long delays and should be modelled as totally asynchronous and undetectable events.

For the sake of clarity, we shall denote with $\sqsubseteq_f$ and $\simeq_f$ the may testing preorder and equivalence obtained when adding rule (R-FailN) to the rules in Table 2.

A Distributed Fault-tolerant Protocol: *k*–set Agreement. We now use may testing to verify the correctness of *k–set agreement* [10], a simple distributed fault-tolerant protocol. Let us consider a totally-connected distributed system with n principals relying on an asynchronous message-passing communication paradigm. Moreover, we also assume that principals can fail according to a fail-silent model of failures; however, the communication medium is reliable, i.e. messages sent will surely be received although the order and the moment in which messages will arrive are unpredictable because of asynchrony.

Each principal has an input value (taken from a totally ordered set) and must produce an output value. The *agreement* problem requires to find a protocol that satisfies three properties: *termination* (i.e. the non-faulty principals eventually produce an output), *agreement* (i.e. the non-faulty principals produce the *same* output value) and *validity* (i.e. the output value must be one of the input values). It is well-known (see, e.g. [2]) that a solution for this problem does not exists even if a single failure occurs.

The *k–set agreement* problem relaxes the agreement property to enable the existence of a solution. Indeed, for each $1 \leq k \leq n$, it requires that, assuming at most $k-1$ faulty principals, the non-failed principals successfully complete their execution by producing outputs taken from a set whose size is at most k. Notice that for $k = 1$ we get the agreement problem without failures.

A possible solution for the k-set agreement problem is given by the following protocol, taken from [2], executed by each principal:

(i) send your input value to all principals (including yourself)
(ii) wait to receive $n-k+1$ values
(iii) output the minimum value received

In this way, if we call $\mathcal{I}$ the set of the input values, the set of output values $\mathcal{O}$ is formed by the k smallest values in $\mathcal{I}$. For the sake of simplicity, we assume that the elements in $\mathcal{I}$ are pairwise distinct; however, the protocol works also if input values are duplicated (in this case $\mathcal{I}$ and $\mathcal{O}$ are multisets).

We use integers as input/output values, while principals are represented as distinct nodes, whose addresses are taken from the set $L = \widetilde{l} \triangleq \{l_1, \ldots, l_n\}$; moreover, we use $d_i \in \mathcal{I}$ to denote the input value of the principal associated to the node whose address is l_i. Once we fix the value for k, node l_i hosts the process

$$P_i^k \;\triangleq\; \mathbf{out}(d_i)@l_1. \;\ldots\; .\mathbf{out}(d_i)@l_n.\mathbf{in}(!z_1^i)@l_i. \;\ldots\; .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l$$

with $m_i \triangleq min\{z^i_j : j = 1,\ldots,n-k+1\}$ and l be a distinct locality used to collect output values. The net implementing the whole protocol is

$$N^k_n \triangleq (\nu\widetilde{l})\Big(\prod_{i=1}^{n} l_i :: P^k_i \parallel \Gamma\Big)$$

where

$$\Gamma \triangleq \prod_{i\neq j} \{l_i \leftrightarrow l_j\}$$

We restricted the localities associated to the principals because no external context is allowed to interfere with the execution of the protocol. Notice that, having restricted the $\widetilde{l}$, all the principals are connected and no **out** prefix will ever block P^k_i (because of law (RNode)). However, this does not prevent failures: the failure of (a reduct of) P^k_i is indeed the failure of principal i.

A formulation of the three properties for the k-set agreement problem is given by Equations (1) and (2) below. The formalisation of k-set agreement and validity properties is given by the Equation

$$N^k_n \simeq_f M^k_n \qquad (1)$$

There, we exploit the auxiliary net

$$M^k_n \triangleq (\nu\widetilde{l},\widetilde{l'})\Big(\prod_{i=1}^{n} l_i :: \mathbf{new}(l'_i).(Q^k_i \mid \prod_{w\in O} \mathbf{out}(w)@l'_i) \parallel \Gamma\Big)$$

where

$$Q^k_i \triangleq \mathbf{out}(d_i)@l_1.\cdots.\mathbf{out}(d_i)@l_n.\mathbf{in}(!z^i_1)@l_i.\cdots.\mathbf{in}(!z^i_{n-k+1})@l_i.\mathbf{in}(m_i)@l'_i.\mathbf{out}(m_i)@l$$

We assume that nodes whose addresses are in $\widetilde{l'}$ cannot fail; this is reasonable because they are only auxiliary nodes and hence their failure is irrelevant for the original formulation of the problem. Intuitively, node l'_i acts as a repository for l_i and contains the possible output values (i.e. the elements of O), while the last **in** action of Q^k_i is a test for checking that the output value produced by the principal i is in O. The net M^k_n obviously satisfies the wanted properties since its principals output only values present in O. The fact that $|O| = k$ then implies the k-set agreement property, while the fact that $O \subseteq \mathcal{I}$ implies validity.

In order to prove the termination property, it suffices to prove that

$$l :: \prod_{j=1}^{n-k+1} \langle\rangle \quad \sqsubseteq_f \quad \hat{N}^k_n \qquad (2)$$

where $\hat{N}^k_n \triangleq (\nu\widetilde{l})\Big(\prod_{i=1}^{n} (l_i :: \hat{P}^k_i \parallel \{l_i \leftrightarrow l\}) \parallel \Gamma\Big)$ and processes $\hat{P}^k_i$ is defined like P^k_i with action $\mathbf{out}()@l$ in place of $\mathbf{out}(m_i)@l$. Clearly, if we only consider termination, N^k_n and $\hat{N}^k_n$ are equivalent, in the sense that a non-faulty principal produces an output value in the first net if and only if its counterpart produces an output in the second net. Equation (2) implies termination of the protocol, since it requires that at least $n-k+1$ tuples are produced at l; by definition of the protocol, this is possible only if $n-k+1$ principals terminate successfully.

Before proving the equations stating the soundness of the protocol, we want to remark that other solutions to the agreement problem in presence of failures have been given in literature. Some of these solutions use *failure detectors* [9,2]. Recently, one such solution has been formalised and proved sound by using a process algebraic approach [17]. The solution in *loc.cit.* is, however, heavier than ours and exploits properties of the operational semantics, instead of working in a (simpler) equational setting. Moreover, it exploits failure detectors which are hardly implementable in a global computing scenario.

Proof of Equations (1) and (2). To prove the properties formulated above, we first need a new equality

$$l :: I_1|\ldots|I_n \ \sqsubseteq_f \ l :: I_1|\ldots|I_m \qquad \text{if } n \leq m \qquad (\dagger)$$

Second, we need to smoothly adapt some of the equalities put forward by Proposition 1: the first equality holds only under the hypothesis that l' is restricted, while the third equality holds only under the further hypothesis that $\langle t\rangle$ is not corruptible at l' (with "$\langle t\rangle$ is not corruptible at l' ", we mean that $l' :: \langle t\rangle$ does never fail). Then, we prove Equation (1) as follows:

$$\begin{aligned}
N_n^k \simeq_f\ & (\nu\widetilde{l})\Big(\prod_{i=1}^{n}\ l_i :: \mathbf{in}(!z_1^i)@l_i.\ \ldots\ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l\,|\,\langle d_1\rangle\,|\ldots|\,\langle d_n\rangle \parallel \Gamma\Big)\\
\simeq_f\ & (\nu\widetilde{l})\Big(\prod_{i=1}^{n}\ l_i :: \mathbf{out}(m_i')@l\,|\,\langle d_{i_1}\rangle\,|\ldots|\,\langle d_{i_{k-1}}\rangle \parallel \Gamma\Big)\\
\simeq_f\ & (\nu\widetilde{l},\widetilde{l'})\Big(\prod_{i=1}^{n}\ (\,l_i :: \mathbf{in}(m_i')@l_i'.\mathbf{out}(m_i')@l\,|\,\langle d_{i_1}\rangle\,|\ldots|\,\langle d_{i_{k-1}}\rangle\\
&\qquad\qquad \parallel\ \ l_i' :: \prod_{w\in O}\ \langle w\rangle \parallel \{l_i \leftrightarrow l_i'\}\,)\ \parallel \Gamma\ \Big)\\
\simeq_f\ & (\nu\widetilde{l},\widetilde{l'})\Big(\prod_{i=1}^{n}\ (\,l_i :: \mathbf{in}(!z_1^i)@l_i.\ \ldots\ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{in}(m_i)@l_i'.\mathbf{out}(m_i)@l\\
&\qquad\qquad |\,\langle d_1\rangle\,|\ldots|\,\langle d_n\rangle\ \ \parallel\ \ l_i' :: \prod_{w\in O}\ \langle w\rangle\ \parallel \{l_i \leftrightarrow l_i'\}\,)\ \parallel \Gamma\ \Big)\\
\simeq_f\ & M_n^k
\end{aligned}$$

where m_i' denotes $m_i[\widetilde{d}/\widetilde{z}]$, with $\widetilde{d} \triangleq \{d_1,\ldots,d_n\} - \{d_{i_1},\ldots,d_{i_{k-1}}\}$ and $\widetilde{z} \triangleq \{z_1,\ldots,z_{n-k+1}\}$. The first and the last steps have been inferred by applying several times (the revised formulation of) Proposition 1.1 . The second and the fourth steps have been inferred by applying several times (the revised formulation of) Proposition 1.3; notice that, since the number of failures is at most $k-1$, the number of non-corruptible data present in each l_i is at least $n-k+1$. The third step relies on Proposition 1.3, .4 and .6 . It is worth to notice that $m_i' \in O$ because, since $|O| = k$, at least one principal whose input value, say d', is in O has not failed; hence d' has been received by all the (non-failed) principals. Moreover, we assumed that the $\widetilde{l'}$ cannot fail and hence the data they host are uncorruptable.

To conclude, we are left with proving Equation (2). This can be done very similarly as follows:

$$\begin{aligned}
\hat{N}_n^k \simeq_f\ & (\nu\widetilde{l})\Big(\prod_{i=1}^{n}\ l_i :: \mathbf{in}(!z_1^i)@l_i.\ \ldots\ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}()@l\,|\,\langle d_1\rangle\,|\ldots|\,\langle d_n\rangle \parallel \{l_i \leftrightarrow l\}\Big)\\
\simeq_f\ & (\nu\widetilde{l})\Big(\prod_{i=1}^{n}\ l_i :: \mathbf{out}()@l\,|\,\langle d_{i_1}\rangle\,|\ldots|\,\langle d_{i_{k-1}}\rangle \parallel \{l_i \leftrightarrow l\}\Big)
\end{aligned}$$

$$\begin{aligned} &\simeq_f l :: \prod_{j=1}^{n} \langle\rangle \\ &\sqsupseteq_f l :: \prod_{j=1}^{n-k+1} \langle\rangle \end{aligned}$$

The first two steps are derived in the same way. The third step is derived using (the revised version of) Proposition 1.1/.4 and Proposition 1.6 . The fourth step derives from law (†).

3.2 Failure of Inter-node Connections

The philosophy underlying our failure model can be easily adapted to deal with link failures too. To this aim, we only need to add the operational rule

$$(\text{R-FailC}) \quad \{l_1 \leftrightarrow l_2\} \longmapsto \mathbf{0}$$

that models the (asynchronous and undetectable) failure of the link between nodes l_1 and l_2.

Discovering Neighbours. When the (multi)set of links in a net can change during computations, routing tables must be dynamic, because the original topology can change at runtime. This task is usually carried on by the so-called *adaptive* (or *dynamic*) routing algorithms. Several proposals have been presented in literature and different standards use different solutions. However, in general, routing algorithms are repeated at regular time intervals and consist in two main phases: first, each node discovers its neighbours; then, it calculates its routing table by usually sharing local information with its neighbours. We present here a simple way to implement in tKlaim the first phase; the (more challenging) study of the second phase is left for future work.

Neighbours can be discovered in a simple way. Each node l can try to send a “hello” message to another node l'; if this action succeeds, then a connection between l and l' does exist; otherwise, nothing can be said (e.g., the message could get lost or the link could be congested and this caused a delay to the message). In our framework, no explicit message is needed: a simple action $\mathbf{eval}(\mathbf{nil})@l'$ performed at l can be used as test for existence of link $\{l \leftrightarrow l'\}$ in the net.

By letting $\sqsubseteq_f$ still denote the may testing preorder in this refined framework, soundness of our solution follows by proving that

$$l :: \mathbf{eval}(\mathbf{nil})@l'.\mathbf{out}(\text{“}CONN\text{”}, l, l')@l \quad \sqsubseteq_f \quad \{l \leftrightarrow l'\} \parallel l :: \langle \text{“}CONN\text{”}, l, l' \rangle$$

The equation above states that if the left hand side successfully passes the test of an observer looking for a tuple $\langle \text{“}CONN\text{”}, l, l' \rangle$ at l, then the link $\{l \leftrightarrow l'\}$ must exist. Its soundness can be easily proved by exploiting the co-inductive proof technique in [13].

4 Modelling Dynamic Connections

Finally, we present another variation of the basic language that let connections dynamically evolve. To this aim, we add two actions to create and destroy a link, respectively. Formally, we add the productions

$$a \ ::= \ \ldots \ \Big| \ \mathbf{conn}(\ell) \ \Big| \ \mathbf{disc}(\ell)$$

to the syntax of Table 1. Intuitively, the first action, when executed at node l, creates a new link between l and ℓ, if the latter name is associated to a network node. Conversely, the second action, when executed at node l, removes a link between l and ℓ, if such a link exists. These intuitions are formalised by the following operational rules, that must be added to those in Table 2:

$$\text{(R-C{\small ONN})} \quad l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{nil} \ \longmapsto \ l :: P \parallel \{l \leftrightarrow l'\}$$

$$\text{(R-D{\small ISC})} \quad l :: \mathbf{disc}(l').P \parallel \{l \leftrightarrow l'\} \ \longmapsto \ l :: P \parallel l' :: \mathbf{nil}$$

Again, for the sake of clarity, we denote with $\simeq_d$ the may testing equivalence in the calculus with dynamic connections.

Message Delivering in a Dynamic Net. To conclude, we now give an application of our theory in a setting where node links change dynamically. To this aim, we use a simplified scenario inspired by the *handover protocol*, proposed by the European Telecommunication Standards Institute (ETSI) for the GSM Public Land Mobile Network (PLMN). The formal specification of the protocol and its service specification are in [24]; we use here an adaption of their approach.

The PLMN is a cellular system which consists of Mobile Stations (MSs), Base Stations (BSs) and Mobile Switching Centres (MSCs). MSs are mobile devices that provide services to end users. BSs manage the interface between the MSs and a stationary net; they control the communications within a geographical area (a cell). Any MSC handles a set of BSs; it communicates with them and with other MSCs using a stationary net.

A new user can enter the system by connecting its MS with a MSC that, in turn, will decide the proper BS responsible for such a MS. Then, messages sent from the user are routed to their destinations by the BS, passing through the MSC handling the BS. However, it may happen that the BS responsible for a MS should be changed during the computation (e.g., because the MS left the area associated to the BS and entered in the area associated to a different BS). In this case, the MSC should carry on the rearrangements needed to cope with the new situation, without affecting the end-to-end communication.

We now model the key features of a PLMN in tKlaim; however, for the sake of simplicity, several aspects will be omitted, like, e.g., the criterion to choose a proper BS for a given MS, or the event originating an handover. Both MSs, BSs and MSCs are modelled as nodes. For the sake of simplicity, we consider a very simple PLMN, with one MSC (whose address is M) and two BSs (whose addresses are $\mathtt{B}_1$ and $\mathtt{B}_2$, resp.).

Let us start with the process that performs the connecting formalities in M.

$$\begin{aligned} ENTER \ &\triangleq \ < gather\ a\ new\ connection\ from\ l > .\mathbf{read}(!B)@\mathtt{BSlist}. \\ &\mathbf{eval}(\mathbf{conn}(l))@B.\mathbf{disc}(l).\mathbf{out}(l, B)@\mathtt{Table} \end{aligned}$$

When a new user want to enter the PLMN, it has to perform a **conn**(M) from his MS, whose address is l; this generates an interrupt in M (that we do not model here) by which the MSC can gather the address of the MS. This address, together with other information (like the geographical area of the user or its credentials), are used by the

MSC to choose a proper BS; in our simplified framework, we let M take a BS's address from a private repository BSlist. Then, the MSC creates a new link from the chosen BS to the MS and destroys the link from itself to the MS. Finally, it records in a private repository Table the fact that the new MS is under the control of the chosen BS.

Once entered the PLMN, the new user can send some data d to (the MS of) a remote user (whose address is l'); this is achieved by letting his MS (whose address is l) perform an action of the form $\mathbf{out}(\text{'send'}, l', d)@l$. Then, the BSs associated to l and l' come into the picture to properly deliver the message. In particular, let $\mathtt{B}_i$ be the BS associated to l and $\mathtt{B}_j$ be the BS associated to l' (for $i, j \in \{1, 2\}$). Then, the message is forwarded from $\mathtt{B}_i$ to $\mathtt{B}_j$ by the process

$$FWD_i \triangleq \mathbf{read}(!x, \mathtt{B}_i)@\mathtt{Table}.\mathbf{in}(\text{'send'}, !y, !z)@x.\mathbf{in}(y, !B)@\mathtt{Table}.\mathbf{out}(y, z)@B$$

This process first retrieves the address of a MS associated to $\mathtt{B}_i$ (in particular, l); then, it collects the message and forwards it to the BS associated to the destination MS. Notice that, in doing this, it 'locks' the link between l' and $\mathtt{B}_j$ until the message will be delivered to l' (see below); this is necessary to avoid that a handover may interfere with the message delivering. Then, the message is collected by $\mathtt{B}_j$ and passed to l' by the process

$$CLT_j \triangleq \mathbf{in}(!dest, !mess)@\mathtt{B}_j.\mathbf{out}(mess)@dest.\mathbf{out}(dest, \mathtt{B}_j)@\mathtt{Table}$$

This process retrieves the message sent by $\mathtt{B}_i$ and passes it to the final MS; then, it releases the 'lock' on the link $\{\mathtt{B}_j \leftrightarrow l'\}$ acquired by $\mathtt{B}_i$ by putting back in Table the tuple $\langle l', \mathtt{B}_j\rangle$. Clearly, there are also processes FWD_j and CLT_i running in $\mathtt{B}_j$ and $\mathtt{B}_i$ respectively, but they do not play any role here.

Finally, the handover is handled by the MSC via the following process:

$$\begin{aligned} HNDVR \triangleq\ & \mathbf{in}(!x, !B)@\mathtt{Table}.\mathbf{read}(!B')@\mathtt{BSlist}. \\ & \mathbf{eval}(\mathbf{disc}(x))@B.\mathbf{eval}(\mathbf{conn}(x))@B'.\mathbf{out}(x, B')@\mathtt{Table} \end{aligned}$$

This process first selects a MS-to-BS association to be changed (the reason why this is needed is not modelled here); then, it chooses a new BS, properly changes the links between the MS and the BSs, and updates the repository Table.

The overall resulting system is

$$\begin{aligned} SYS \triangleq\ & (\nu\mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\mathtt{M} :: *ENTER \mid *HNDVR \\ & \parallel\ \mathtt{BSlist} :: \langle \mathtt{B}_1\rangle \mid \langle \mathtt{B}_2\rangle\ \parallel\ \mathtt{Table} :: \mathbf{nil} \\ & \parallel\ \mathtt{B}_1 :: *FWD_1 \mid *CLT_1\ \parallel\ \mathtt{B}_2 :: *FWD_2 \mid *CLT_2\ ) \end{aligned}$$

where $*P$ denotes the replication of P and stands for an unbounded number of copies of P running in parallel. Replication can be easily encoded through recursion by letting $*P$ be a shortcut for $\mathbf{rec}\, X.(P|X)$. Soundness of the system can be formulated as:

$$\begin{aligned} & (\nu l)(l :: \mathbf{conn}(\mathtt{M}).\mathbf{out}(\text{'send'}, l', \text{'HI'})@l \parallel l' :: \mathbf{conn}(\mathtt{M}) \parallel SYS) \\ & \quad \simeq_d\ (\nu l)(l' :: \langle \text{'HI'}\rangle \parallel SYS) \end{aligned} \qquad (3)$$

Notice that l is restricted only to simplify proofs: soundness of the protocol is not affected by the fact that the MSs are public or not.

Proof of Equation (3). To prove the equation above, we first need two laws for the primitives **conn** and **disc**, that are quite expectable.

$$l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{nil} \;\simeq_d\; l :: P \parallel \{l \leftrightarrow l'\} \qquad (\star)$$

$$(\nu l')(l :: \mathbf{disc}(l').P \parallel \{l \leftrightarrow l'\}) \;\simeq_d\; (\nu l')(l :: P \parallel l' :: \mathbf{nil}) \qquad (\star\star)$$

Moreover, we also need an adapted version of Proposition 1.3 to deal with action **read**. It is defined as follows:

$$\begin{array}{l} (\nu l')(\, l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle t\rangle\,) \;\simeq_d\; (\nu l')(l :: P\sigma \parallel l' :: \langle t\rangle) \\ \text{if } \; match(\mathcal{E}[\![\, T \,]\!], t) = \sigma. \end{array} \qquad (\ddagger)$$

We are ready to prove Equation (3), yielding the soundness of the protocol for the PLMN. It is easy to prove that

$$\begin{array}{rl}
& (\nu l)(l :: \mathbf{conn}(\mathtt{M}).\mathbf{out}(\text{'send'}, l', \text{'HI'})@l \parallel l' :: \mathbf{conn}(\mathtt{M}) \parallel SYS) \\[1ex]
\simeq_d & (\nu l, \mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\; l :: \langle \text{'send'}, l', \text{'HI'}\rangle \;\parallel\; l' :: \mathbf{nil} \\
& \quad \parallel\; \mathtt{M} :: {*ENTER} \mid {*\,HNDVR} \;\parallel\; \mathtt{BSlist} :: \langle \mathtt{B}_1\rangle \mid \langle \mathtt{B}_2\rangle \\
& \quad \parallel\; \mathtt{Table} :: \langle l, \mathtt{B}_i\rangle \mid \langle l', \mathtt{B}_j\rangle \;\parallel\; \{l \leftrightarrow \mathtt{B}_i\} \;\parallel\; \{l' \leftrightarrow \mathtt{B}_j\} \\
& \quad \parallel\; \mathtt{B}_1 :: {*FWD_1} \mid {*\,CLT_1} \;\parallel\; \mathtt{B}_2 :: {*FWD_2} \mid {*\,CLT_2}\;) \\[1ex]
\simeq_d & (nul, \mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\; l :: \mathbf{nil} \;\parallel\; l' :: \mathbf{nil} \\
& \quad \parallel\; \mathtt{M} :: {*ENTER} \mid {*\,HNDVR} \;\parallel\; \mathtt{BSlist} :: \langle \mathtt{B}_1\rangle \mid \langle \mathtt{B}_2\rangle \\
& \quad \parallel\; \mathtt{Table} :: \langle l, \mathtt{B}_i\rangle \mid \langle l', \mathtt{B}_j\rangle \;\parallel\; \{l \leftrightarrow \mathtt{B}_i\} \;\parallel\; \{l' \leftrightarrow \mathtt{B}_j\} \\
& \quad \parallel\; \mathtt{B}_1 :: {*FWD_1} \mid {*\,CLT_1} \;\parallel\; \mathtt{B}_2 :: {*FWD_2} \mid {*\,CLT_2} \\
& \quad \parallel\; \mathtt{B}_i :: \mathbf{in}(l', !B)@\mathtt{Table}.\mathbf{out}(l', \text{'HI'})@B\;) \\
\triangleq & K
\end{array}$$

The first equality can be inferred using laws ($\star$) and ($\ddagger$), Proposition 1.2, laws ($\star$) and ($\star\star$), and Proposition 1.1; the second equality can be inferred using law ($\ddagger$) and Proposition 1.3. Now we cannot proceed equationally: indeed, there are two parallel components that may want to retrieve the tuple $\langle l', \mathtt{B}_j\rangle$ at $\mathtt{Table}$, i.e. the process $\mathbf{in}(l', !B)@\mathtt{Table}.\mathbf{out}(l', \text{'HI'})@B$ running at $\mathtt{B}_i$ and the process $HNDVR$ running at $\mathtt{M}$. This fact makes Proposition 1.3 not applicable here.

To overcome this problem, we observe that there are only three possible evolutions for K: make a handover for l, make a handover for l', or complete the delivering of the message that l sent to l'. The first evolution is compatible with the latter two ones that, in turn, are mutually exclusive. Thus, let $\mathcal{H}$ be the set of pairs $(N, (\nu l)(l' :: \langle \text{'HI'}\rangle \parallel SYS))$, where N is any reduct of K obtained by giving the precedence to the handover of l' w.r.t. the message delivering. Symmetrically, let $\mathcal{D}$ be the set of pairs $(N, (\nu l)(l' :: \langle \text{'HI'}\rangle \parallel SYS))$, where N is any reduct of K obtained by giving the precedence to the message delivering w.r.t. the handover of l'. Now, it can be easily proved that

$$\{\,(K, (\nu l)(l' :: \langle \text{'HI'}\rangle \parallel SYS))\,\} \,\cup\, \mathcal{H} \,\cup\, \mathcal{D}$$

is a bisimulation. By the fact that $\approx_d \,\subset\, \simeq_d$ and by transitivity of $\simeq_d$, this suffices to prove Equation (3).

5 Conclusions and Related Work

We have presented a calculus inspired by Klaim whose main feature is the handling of inter-node connections. We have first presented a basic setting where connections are reliable and immutable; then, we have enriched the basic framework with failures and dynamically evolving connections, two more advanced features for global computing. In each setting, we have used our formalisms to specify and verify some non-trivial global computing applications, by exploiting a may-testing equivalence.

Related work. In the last decade, several languages for modelling and programming distributed and global computing systems have been proposed in literature; we mention here only the most strictly related ones.

In DJoin [16], located mobile processes are hierarchically structured and form a tree-like structure evolving during the computation. Entire subtrees, not just single processes, can move and fail. Communication takes place in two steps: first, the sending process sends a message on a channel; then, the ether (i.e. the environment containing all the nodes) delivers the message to the (unique) process that can receive on that channel. Failures are programmed (i.e., they result from the execution of some process actions) and can be detected by processes. We believe that the setting presented in this paper is more realistic than DJoin because the considered interconnection topology is more general than trees and also because we do not assume any implicit engine for distant communications. Finally, we model failures in a way that is closer to actual global computers.

The Ambient calculus [7] is an elegant notation to model hierarchically structured distributed applications. Like in our work, the calculus is centered around the notion of connections between ambients, that are containers of processes and data. Each language primitive can be executed only if the ambient hierarchy is structured in a precise way; e.g., an ambient n can enter an ambient m only if n and m are sibling, i.e. they are both contained in the same ambient. However, like DJoin, Ambient strongly relies on a tree-like structure for the ambient hierarchy. Moreover, to the best of our knowledge, no explicit notion of failures, close to actual global computing requirements, has been ever given for Ambient.

[27] presents Nomadic Pict, a distributed and agent-based language based on the π-calculus. It relies on a flat net where named agents can roam. Communication between two agents can take place only if they are located at the same node (thus no low-level remote communication is allowed). However, the language also provides a (high-level) primitive for remote communication, that transparently delivers a message to an agent even if the latter is not co-located with the sender. This primitive is then encoded in the low-level calculus by a central forwarding server, implemented by only using the low-level primitives. The assumption that only co-located agents can communicate is, in our opinion, too demanding. Moreover, it is not clear to us how the theory can be adapted to consider failures.

Another distributed version of the π-calculus is presented in [21]; the resulting calculus contains primitives for code movement and creation of new localities/channels in a net with a flat architecture. The main feature of the language is the possibility of controlling process activities via (sophisticated and non-standard) type systems. No notion of explicit connections and of failures have been integrated in the framework yet.

We now touch upon some formalisms for distributed computing relying on the powerful paradigm put forward by LINDA [18]. In *TuCSoN* [23], tuple spaces are enhanced with the capability of programming their behaviour in response to communication events; moreover, the computational model relies on a hierarchical collection of (possibly) distributed tuple spaces. *MARS* [5] is a coordination tool for Java-based mobile agents that defines LINDA-like tuple spaces programmable to react when accessed by agents. Such mechanisms can be used to control accesses to specific tuples. In TKLAIM, this can be obtained either by using dynamically created (private) nodes or by tailoring the capability-based type systems presented in [19, 20]. *Lime* [26] exploits multiple tuple spaces to coordinate mobile agents and adds mobility to tuple spaces: it allows processes to have private tuple spaces and to transparently and transiently share them. In TKLAIM, sharing of resources can be somehow achieved via dynamic handling of links; however, tuple spaces are bound to nodes and nodes cannot move.

Finally, we want to remark that the use of observational equivalences to state and proof soundness of protocols is a well-established technique in the field of process calculi; some notable examples are [1, 22, 24, 28]. In particular, in the last paper, an automatic verification tool to prove equivalences in the π-calculus is described. As an application, the authors automatically verify an equality, similar to ours, stating the soundness of the PLMN example.

Acknowledgements. We would like to thank the anonymous referees for some suggestions that helped in improving the paper.

References

1. M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. In *Proc. of CONCUR'97*, volume 1243 of *LNCS*, pages 59–73. Springer, 1997.
2. H. Attiya and J. Welch. *Distributed Computing*. McGraw Hill, 1998.
3. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th WETICE*, pages 110–115. IEEE, 1998.
4. L. Bettini, M. Loreti, R. Pugliese. An Infrastructure Language for Open Nets. In *Proc. of the 2000 ACM Symposium on Applied Computing*, pages 373–377, ACM Press, 2002.
5. G. Cabri, L. Leonardi and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc. of the 2nd Int. Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer, 1998.
6. L. Cardelli. Abstractions for mobile computation. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 51–94. Springer, 1999.
7. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
8. S. Castellani, P. Ciancarini, and D. Rossi. The ShaPE of ShaDE: a coordination system. Tech. Rep. UBLCS 96-5, Dip. di Scienze dell'Informazione, Univ. di Bologna, Italy, 1996.
9. T. Chandra and S.Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.
10. S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132–158, 1993.

11. N. Davies, S. Wade, A. Friday, and G. Blair. L^2imbo: a tuple space based platform for adaptive mobile applications. In *Int. Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP'97)*, 1997.
12. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
13. R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. Technical Report 07/2004, Dip. di Informatica, Univ. di Roma "La Sapienza". Available at `http://www.dsi.uniroma1.it/~gorla/papers/bo4k-full.pdf`.
14. R. De Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
15. D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proc. of ISADS 2001*, pages 278–286. IEEE, 2001.
16. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
17. R. Fuzzati, M. Merro, and U. Nestmann. Modelling Consensus in a Process Calculus. In *Proc. of CONCUR'03*, volume 2761 of *LNCS*. Springer-Verlag, 2003.
18. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
19. D. Gorla and R. Pugliese. Resource Access and Mobility Control with Dynamic Privileges Acquisition. In *Proc. of ICALP'03*, volume 2719 of *LNCS*, pages 119-132. Springer, 2003.
20. D. Gorla and R. Pugliese. Enforcing Security Policies via Types. In *Proc. of Security in Pervasive Computing (SPC'03)*, volume 2802 of *LNCS*, pages 88-103. Springer, 2003.
21. M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.
22. R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.
23. A. Omicini and F. Zambonelli. Coordination of Mobile Information Agents in Tucson. *Journal of Internet Research*, 8(5):400-413, 1998.
24. F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4:497–543, 1992.
25. J. Parrow. An introduction to the pi-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.
26. G.P. Picco, A.L. Murphy and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. IEEE, 1999.
27. A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructures for Mobile Computation. In *Proc. of POPL'01*, pages 116–127. ACM Press, 2001.
28. B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In *Proc. of CAV '94*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.