Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 64 (2007) 187–204
Science of Computer Programming
www.elsevier.com/locate/scico

Global computing in a dynamic network of tuple spaces☆

Rocco De Nicola[a], Daniele Gorla[b,*], Rosario Pugliese[a]

[a] *Dipartimento di Sistemi e Informatica, Università di Firenze, Italy*
[b] *Dipartimento di Informatica, Università di Roma "La Sapienza", Roma, Italy*

Received 15 September 2005; received in revised form 15 March 2006; accepted 15 June 2006
Available online 26 September 2006

Abstract

We present TKLAIM (*Topological* KLAIM), a process description language that retains the main features of KLAIM (process distribution and mobility, remote and asynchronous communication through distributed data spaces), but extends it with new constructs to flexibly model the interconnection structure underlying a network and its evolution in time. We show how TKLAIM can be used to model a number of interesting distributed applications and how systems correctness can be guaranteed, also in the presence of failures, by exploiting observational equivalences to study the relationships between descriptions of systems at different levels of abstraction.

Keywords: Global computing; Formal methods; Observational equivalence; Distributed algorithms; Program verification

1. Introduction

In the last decade, programming computational infrastructures available globally for offering uniform services has become an important topic 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. This has stimulated research on abstractions, models and calculi that could provide the basis for the design and the analysis of network aware programs, where physical and logical mobility of systems plays a crucial role. The research area that considers all the above issues is now called *Global Computing*.

Both the linguistic abstractions and the foundational aspects of Global Computing have been investigated. On the linguistic side, the search is for languages with primitives that support *network awareness* (locations are explicitly referenceable), *disconnected operations* (moved code executes also in the presence of intermittent connections), *flexible communication mechanisms* (like distributed repositories storing content addressable data) and *remote operations* (like asynchronous remote communications). On the foundational side, the demand is instead on the development of tools and techniques to build safe and trustworthy global systems, to analyse their behaviour, and

☆ This work has been partially supported by EU within the FP6-2004-IST-FET Proactive project SENSORIA proposal contract number 016004.
* Corresponding author.
E-mail addresses: denicola@dsi.unifi.it (R. De Nicola), gorla@di.uniroma1.it (D. Gorla), pugliese@dsi.unifi.it (R. Pugliese).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.06.004

to demonstrate their conformance to given specifications. Clearly, such theories should capture the above listed distinctive features of global systems.

KLAIM (Kernel Language for Agents Interaction and Mobility), that we introduced in [12], can be placed along this research line. It is a formalism specifically designed to describe distributed systems made up of several mobile interacting components. KLAIM primitives, inspired by the LINDA coordination model [21], allow programmers to distribute processes at different localities of a net, where data can be placed and retrieved. Localities are first-class citizens that can be dynamically created and communicated. In more recent papers, we have evolved KLAIM to a full-fledged programming language (X-KLAIM [4]) to be used for distributed mobile programming but have also distilled it into a number of simpler process languages [13]. Building on these languages, here we introduce TKLAIM (*Topological* KLAIM), a formalism that relies on inter-node connections to allow or deny remote operations. Connections are programmable, in that they can be explicitly and dynamically activated and deactivated by processes, and remote operations can take place only if the involved nodes are directly connected.

Many process algebraic languages for modelling and programming global computational infrastructures have been proposed in the literature [18,8,28,19]; also a number of languages and systems for distributed computing relying on and extending the powerful LINDA [21] paradigm have been put forward both from Academia and from industries [3,27,17,24]. Notably, in all these formalisms, little attention has been devoted to explicitly modelling the network topology: it usually originates from the linguistic choices concerning the mobility paradigm. Thus, when migration consists in the movement of bare processes, the network is usually seen as a fully connected graph of computing sites where new sites can be dynamically added (see, e.g., Dπ-calculus [22], KLAIM [12] or NOMADIC PICT [28]). Instead, when migration consists in the movement of entities with executable content (such as entire sites), the network is usually seen as a forest of trees that evolves with the addition/pruning/displacement of subtrees (see, e.g., Ambient [8] and its variants, or DJoin [18]). However, global computers (e.g. the Internet) are generic graphs, i.e. their nodes are neither organised in tree-like structures nor are fully (directly) connected. Connections can unpredictably break down rendering nodes (at least temporarily) unreachable. To meet the demands arising from modelling the network topology of global computers and its evolution in time we have extended KLAIM with supports for connections and failures, thus obtaining TKLAIM.

Another major contribution of this paper, is the development of a framework for specifying and proving properties of global computing applications. On the one hand, we show how TKLAIM can be used to model a number of interesting distributed applications. On the other hand, we show how system correctness can be guaranteed by exploiting behavioural equivalences to study the relationships between system descriptions at different levels of abstraction. To state and prove properties of systems, we follow the approach presented in [1]. According to this approach, in correspondence to the detailed description in TKLAIM of a given system (as close as possible to the actual implementation), another, more abstract and intuitive, specification has to be provided that clearly manifests the wanted (or unwanted) behaviour of the system under consideration. The theory of *may testing* [15] can then be used to establish the relationship between the abstract and the concrete specification and thus to check whether the expected (or unexpected) behaviour is possible.

To gently introduce the reader to our approach, in Section 2 we present a very basic formalism where inter-node connections are explicitly fixed at the outset. This scenario is very close to Local Area Networks, where physical connections are usually reliable and immutable (or change very rarely). Section 3 shows how the basic model can be used to program communication between machines that are indirectly connected: we present a routing messenger process and prove that it behaves correctly. In the following sections we present two variants of the basic model. In Section 4, we enrich the language with different forms of failures. We first consider a scenario where only nodes and node components (i.e., data or processes) can fail and, under these assumptions, we establish soundness of a distributed fault-tolerant protocol for the *'k-set agreement'* problem [10]; then, we discuss failures of connections. Dynamic connections, explicitly modifiable by processes, are considered in Section 5; there, we model a more sophisticated routing scenario and establish its soundness. We conclude in Section 6 with a discussion on future and related work.

2. The basic language

In this section, we report syntax, operational and may-testing semantics of a basic variant of TKLAIM with only static connections.

Table 1
Syntax of TKLAIM

NETS: $N ::= \mathbf{0} \mid l :: C \mid \{l_1 \leftrightarrow l_2\} \mid N_1 \| N_2 \mid (\nu l)N$	COMPONENTS: $C ::= \langle t \rangle \mid P \mid C_1 \mid C_2$
PROCESSES: $P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid A(\widetilde{e}, \widetilde{\ell})$	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: $e ::= V \mid x \mid \ldots$

2.1. 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:

- $\mathcal{V}$, the set of *basic values* (integers, strings, booleans, ...), is ranged over by V;
- $\mathcal{L}$, the set of *localities*, is ranged over by l;
- $\mathcal{X}$, the set of *basic variables*, is ranged over by x;
- $\mathcal{U}$, the set of *locality variables*, is ranged over by u;
- $\mathcal{A}$, the set of *process identifiers*, is ranged over by A.

Finally, ℓ is used to denote localities and locality variables.

Nets, ranged over by $N, M, K, \ldots$, are finite collections of nodes and inter-node connections. A *node* is a pair $l :: C$, where locality l is the address (i.e., the network reference) of the node and C is the (parallel) component located at l. *Components*, ranged over by $C, D, \ldots$, can be either processes or tuples of data, denoted by $\langle t \rangle$. *Connections* 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]) connected. In $(\nu l)N$, name l is private to N; the intended effect is that, if one considers the term $M \parallel (\nu l)N$, then locality l of N cannot be referred from within M.

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 process invocation. *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.

The exact syntax of *expressions*, e, is deliberately omitted; we only assume that expressions contain, at least, basic values and variables. *Tuples*, t, are sequences of expressions, localities or 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.

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. 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 terms are *alpha-equivalent*, written $=_\alpha$, if one can be obtained from the other by renaming bound names. In the following, 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.

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

Table 2
Structural congruence

(ALPHA) $N \equiv N'$ if $N =_\alpha N'$	(PZERO) $N \parallel \mathbf{0} \equiv N$
(PCOM) $N_1 \parallel N_2 \equiv N_2 \parallel N_1$	(PASS) $(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$
(RCOM) $(\nu l_1)(\nu l_2)N \equiv (\nu l_2)(\nu l_1)N$	(EXT) $N_1 \parallel (\nu l)N_2 \equiv (\nu l)(N_1 \parallel N_2)$ if $l \notin fn(N_1)$
(GARB) $(\nu l)\mathbf{0} \equiv \mathbf{0}$	(CALL) $l :: A(\widetilde{e}, \widetilde{\ell}) \equiv l :: P[\widetilde{e}/\widetilde{x}, \widetilde{\ell}/\widetilde{u}]$ if $A(\widetilde{x}, \widetilde{u}) \stackrel{\text{def}}{=} P$
(CLONE) $l :: C_1 \mid C_2 \equiv l :: C_1 \parallel l :: C_2$	(ABS) $l :: C \equiv l :: C \mid \mathbf{nil}$
(SELF) $l :: \mathbf{nil} \equiv l :: \mathbf{nil} \parallel \{l \leftrightarrow l\}$	(BIDIR) $\{l_1 \leftrightarrow l_2\} \equiv \{l_2 \leftrightarrow l_1\}$
(CONNNODE) $\{l_1 \leftrightarrow l_2\} \equiv \{l_1 \leftrightarrow l_2\} \parallel l_1 :: \mathbf{nil}$	

Notation $\widetilde{\cdot}$ denotes a (possibly empty) sequence of objects; e.g., $\widetilde{l}$ is a sequence of localities. Thus, $A(\widetilde{e}, \widetilde{\ell})$ stands for the invocation of the process identified by A with actual parameters $\widetilde{e}, \widetilde{\ell}$. It is assumed that each process identifier A has a *single* defining equation $A(\widetilde{x}, \widetilde{u}) \stackrel{\text{def}}{=} P$ where the free variables of P are contained in $\widetilde{x}, \widetilde{u}$. Moreover, to guarantee uniqueness of solution of recursive process definitions, it is assumed that all the identifiers occurring in P are *guarded*, i.e. they occur within the scope of a blocking **in/read** prefix.

We write $Z \triangleq W$ to mean that Z is of the form W; this notation is used to assign a symbolic name Z to the term W. We shall sometimes write $\mathbf{in}(\,)@l$, $\mathbf{out}(\,)@l$ and $\langle\rangle$ to mean that the argument of the actions or the tuple are irrelevant. We will use the standard notation $P[e/x]$ to indicate the capture avoiding substitution of the expression e for the free occurrences of the variable x in P; $P[\widetilde{e}/\widetilde{x}]$ will denote the simultaneous substitution of each free occurrence of $x \in \widetilde{x}$ with the corresponding $e \in \widetilde{e}$ in P. $P[l/u]$ and $P[\widetilde{l}/\widetilde{u}]$ have a similar meaning. We shall omit trailing occurrences of process **nil** and write $\prod_{j \in J} W_j$ for the parallel composition (both '|' and '‖') of terms (components or nets, resp.) W_j. Finally, when in a process definition $A(\widetilde{x}, \widetilde{u}) \stackrel{\text{def}}{=} P$ both $\widetilde{x}$ and $\widetilde{u}$ are empty, we shall simply write A instead of $A(\,)$ to invoke it.

2.2. Operational semantics

TKLAIM operational semantics is given by means of a reduction relation relying on a structural congruence. The *structural congruence*, $\equiv$, identifies those nets which intuitively represent the same net and is inspired to the π-calculus' one (see, e.g., [26]). Formally, it is defined as the smallest congruence satisfying the laws in Table 2. Law (ALPHA) equates alpha-equivalent nets; laws (PZERO), (PCOM) and (PASS) state that '‖' is a monoidal operator with $\mathbf{0}$ as identity element. Laws (RCOM), (GARB) and (EXT) handle restrictions: the first one regulates their commutativity; the second one collects unused (garbage) restricted names; the third one is the standard π-calculus' rule for scope extension and states that the scope of a bound name can be extended, provided that this does not cause any name capture. Law (CALL) permits to freely fold/unfold process invocations; law (CLONE) turns a parallel between co-located components into a parallel between nodes; law (ABS) states that **nil** is the identity for '|' (by using law (CLONE), it is easy to see that also '|' is a monoidal operator). Finally, law (SELF) states that nodes are self-connected; law (BIDIR) states that connections are bidirectional; law (CONNNODE) states that a connection can be placed only between existing nodes.

Table 3
TKLAIM reduction relation

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

$$\text{(R-EVAL)} \quad l :: \mathbf{eval}(Q)@l'.P \parallel \{l \leftrightarrow l'\} \longmapsto l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: Q$$

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

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

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

$$\text{(R-PAR)} \quad \frac{N_1 \longmapsto N_1'}{N_1 \parallel N_2 \longmapsto N_1' \parallel N_2}$$

$$\text{(R-RES)} \quad \frac{N \longmapsto N'}{(\nu l)N \longmapsto (\nu l)N'}$$

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

The reduction relation is given in Table 3 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 precise definition depends on the exact syntax of expressions and, thus, is 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 do match. Formally, function *match* returns a substitution defined as follows:

$$match(V; V) = \epsilon \qquad match(!x; V) = [^{V}/_{x}]$$

$$match(l; l) = \epsilon \qquad match(!u; l) = [^{l}/_{u}]$$

$$\frac{match(T_1; t_1) = \sigma_1 \qquad match(T_2; t_2) = \sigma_2}{match(T_1, T_2; t_1, t_2) = \sigma_1 \circ \sigma_2}$$

where we let 'ϵ' be the empty substitution and '$\circ$' denote substitutions composition.

The operational rules of TKLAIM can be briefly explained as follows. Rule (R-NEW) says that execution of action **new**(l') creates a new node at the restricted address l' and a connection with the creating node l. Rule (R-EVAL) states that a process can be spawned at l' by a process running at l only if l and l' are directly connected. Rule (R-OUT) evaluates the expressions within the argument tuple and sends the resulting tuple to the target node. Again, this is possible only if the source and the target nodes are directly connected. Rules (R-READ) and (R-IN) require existence of a matching tuple in the target node and of a connection between the source and the target node. The tuple is then used to replace the variables bound by the template in the continuation of the process performing the actions. With action **in** the matched tuple is consumed while with action **read** it is not. Rules (R-PAR), (R-RES) and (R-STRUCT) are standard.

TKLAIM adopts a LINDA-like [21] communication mechanism: data are anonymous and associatively accessed via pattern matching; communication is asynchronous. Indeed, although there exist prefixes for placing data to (possibly

remote) nodes, no synchronisation takes place between processes, because their interactions are mediated by nodes, that act as data repositories.

To conclude the presentation of TKLAIM's operational semantics, we want to stress that interactions between nodes that are not directly connected is forbidden. This could be remedied in two ways. The first way is to change the presented operational semantics by adding the structural law

$$\{l_1 \leftrightarrow l_2\} \parallel \{l_2 \leftrightarrow l_3\} \equiv \{l_1 \leftrightarrow l_2\} \parallel \{l_2 \leftrightarrow l_3\} \parallel \{l_1 \leftrightarrow l_3\}$$

that states a sort of 'transitivity' property for direct connections. In this way, 'single-hop' connections are placed also between not directly connected nodes. Alternatively, one could explicitly program remote interactions through 'multi-hop' connections. Thus, if there exists a path of connections from l to l', then a process P running at l willing to interact with l' can do so by means of a mobile process spawned from l 'towards' l'.

Adopting one way or the other only depends on the chosen abstraction level. We prefer the second alternative because we consider it more basic and because it fits better in more dynamic scenarios; thus, in Section 3, we shall present a possible implementation of multi-hop communication and a proof of its correctness.

2.3. Observational semantics

To state and prove properties of TKLAIM nets, we follow the approach put forward in [15] and use the *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 nets. More precisely, *observers*, ranged over by $O, O', O_1, \ldots$, are nets that can use an additional distinct locality name $\texttt{test} \notin \mathcal{L}$ as a node address. *Computations* from $N \parallel O$ are (possibly infinite) sequences of reductions $N \parallel O\,(\triangleq (\nu\widetilde{l_0})(N_0 \parallel O_0)) \longmapsto (\nu\widetilde{l_1})(N_1 \parallel O_1) \longmapsto \cdots$; we call such a computation *successful* if there is some $i \geq 0$ such that $O_i \equiv O' \parallel \texttt{test} :: \langle\rangle$ and $\texttt{test} \notin \widetilde{l_i}$. We write N MAY O whenever there exists a successful computation from $N \parallel O$; moreover, we shall sometimes say that N *satisfies* O. Finally, N NOTMAY O stands for the negation of N MAY O.

Definition 1 (*May Testing Preorder*). *May testing preorder*, $\sqsubseteq$, is the least preorder on TKLAIM nets such that, for every $N \sqsubseteq M$, it holds that N MAY O implies M MAY O, for each observer O.

When $N \sqsubseteq M$, we sometimes write $M \sqsupseteq N$ (i.e. relation $\sqsupseteq$ is the 'inverse' of $\sqsubseteq$). *May testing equivalence*, $\simeq$, is defined as the intersection of $\sqsubseteq$ and $\sqsupseteq$. Given a set of observers $\mathcal{O}$, we will write $\sqsubseteq^{\mathcal{O}}$ (resp. $\simeq^{\mathcal{O}}$) to denote that $\sqsubseteq$ (resp. $\simeq$) holds when considering only observers from $\mathcal{O}$.

To relate nets under both $\sqsubseteq$ and $\simeq$, it is necessary to describe the possible interactions between an observer and the observed net; thus, we need a fully compositional operational semantics for TKLAIM. To this aim, we reformulate the semantics of our language as a *labelled transition system* (LTS) to make apparent the action a net is willing to perform in order to evolve. The labelled transition relation, $\xrightarrow{\alpha}$, is defined as the least relation over nets induced by the rules in Table 4. Labels take the form

$$\alpha ::= \tau \;\mid\; l_1 \curvearrowright l_2 \;\mid\; (\nu\widetilde{l})\,\langle t\rangle @ l_1 : l_2 \;\mid\; l_1 : \triangleright l_2 \;\mid\; l_1 : t \triangleleft l_2$$

We let $bn(\alpha)$ be $\widetilde{l}$ if $\alpha = (\nu\widetilde{l})\,\langle t\rangle @ l_1 : l_2$ and be $\emptyset$ otherwise; $fn(\alpha)$ and $n(\alpha)$ are defined accordingly.

Let us now explain the intuition behind the labels of the LTS and some key rules. Label α in $N \xrightarrow{\alpha} N'$ can be

- τ : this means that N may perform a reduction step to become N' (see Proposition 1).
- $l_1 \curvearrowright l_2$: this means that in N there is a direct connection between nodes l_1 and l_2 (see (LTS-LINK)).
- $(\nu\widetilde{l})\,\langle t\rangle @ l_1 : l_2$: this means that in N there is a tuple $\langle t\rangle$ located at l_1 and a connection $\{l_1 \leftrightarrow l_2\}$; the tuple is available for processes located at l_2 (see (LTS-TUPLE), (LTS-OFFER) and (LTS-LINK)). Moreover, names $\widetilde{l}$ occur restricted in N (see (LTS-OPEN)).
- $l_1 : \triangleright l_2$: this means that in N there is a process located at l_1 willing to send a component at l_2 (see (LTS-OUT) and (LTS-EVAL)). For the sending to take place, a direct connection between such nodes is needed (see (LTS-SEND)).
- $l_1 : t \triangleleft l_2$: this means that in N there is a process located at l_1 willing to retrieve a tuple $\langle t\rangle$ from l_2 (see (LTS-IN) and (LTS-READ)). For the actual retrieval, a direct connection between such nodes and tuple $\langle t\rangle$ at l_2 are needed (see (LTS-COMM)).

Table 4
A labelled transition system

(LTS-NEW)
$$l :: \mathbf{new}(l').P \xrightarrow{\tau} (\nu l')(l :: P \parallel \{l \leftrightarrow l'\})$$

(LTS-LINK)
$$\{l_1 \leftrightarrow l_2\} \xrightarrow{l_1 \frown l_2} l_1 :: \mathbf{nil} \parallel l_2 :: \mathbf{nil}$$

(LTS-EVAL)
$$l_1 :: \mathbf{eval}(Q)@l_2.P \xrightarrow{l_1 : \triangleright\, l_2} l_1 :: P \parallel \{l_1 \leftrightarrow l_2\} \parallel l_2 :: Q$$

(LTS-TUPLE)
$$l_1 :: \langle t \rangle \xrightarrow{\langle t \rangle \ @\ l_1 : l_1} l_1 :: \mathbf{nil}$$

(LTS-IN)
$$\frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l_1 :: \mathbf{in}(T)@l_2.P \xrightarrow{l_1 : t \,\triangleleft\, l_2} l_1 :: P\sigma \parallel \{l_1 \leftrightarrow l_2\}}$$

(LTS-OFFER)
$$\frac{N_1 \xrightarrow{\langle t \rangle \ @\ l_2 : l_2} N_1' \quad N_2 \xrightarrow{l_2 \frown l_1} N_2'}{N_1 \parallel N_2 \xrightarrow{\langle t \rangle \ @\ l_2 : l_1} N_1' \parallel N_2'}$$

(LTS-READ)
$$\frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l_1 :: \mathbf{read}(T)@l_2.P \xrightarrow{l_1 : t \,\triangleleft\, l_2} l_1 :: P\sigma \parallel \{l_1 \leftrightarrow l_2\} \parallel l_2 :: \langle t \rangle}$$

(LTS-RES)
$$\frac{N \xrightarrow{\alpha} N' \quad l \notin n(\alpha)}{(\nu l)N \xrightarrow{\alpha} (\nu l)N'}$$

(LTS-OUT)
$$\frac{t' = \mathcal{E}[\![\, t \,]\!]}{l_1 :: \mathbf{out}(t)@l_2.P \xrightarrow{l_1 : \triangleright\, l_2} l_1 :: P \parallel \{l_1 \leftrightarrow l_2\} \parallel l_2 :: \langle t' \rangle}$$

(LTS-SEND)
$$\frac{N_1 \xrightarrow{l_1 : \triangleright\, l_2} N_1' \quad N_2 \xrightarrow{l_1 \frown l_2} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

(LTS-OPEN)
$$\frac{N \xrightarrow{(\nu \widetilde{l})\ \langle t \rangle \ @\ l_2 : l_1} N' \quad l \in fn(t) - \{\widetilde{l}, l_1, l_2\}}{(\nu l)N \xrightarrow{(\nu l, \widetilde{l})\ \langle t \rangle \ @\ l_2 : l_1} N'}$$

(LTS-PAR)
$$\frac{N_1 \xrightarrow{\alpha} N_2 \quad bn(\alpha) \cap fn(N) = \emptyset}{N_1 \parallel N \xrightarrow{\alpha} N_2 \parallel N}$$

(LTS-COMM)
$$\frac{N_1 \xrightarrow{l_1 : t \,\triangleleft\, l_2} N_1' \quad N_2 \xrightarrow{\langle t \rangle \ @\ l_2 : l_1} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

(LTS-STRUCT)
$$\frac{N \equiv N_1 \quad N_1 \xrightarrow{\alpha} N_2 \quad N_2 \equiv N'}{N \xrightarrow{\alpha} N'}$$

Labels $l_1 : \triangleright\, l_2$ and $l_1 : t \,\triangleleft\, l_2$ describe 'intentions' of processes running in the net. E.g., (LTS-OUT) should be read as: "process $\mathbf{out}(t)@l_2.P$ running at l_1 is willing to send a component at l_2; when such an intention is concretised, l_1 will be left with process P, l_2 will receive the tuple resulting from the evaluation of t, and the execution context will provide the needed connection". Rules (LTS-EVAL), (LTS-IN) and (LTS-READ) should be interpreted similarly.

(LTS-OPEN) signals extrusion of bound names and is used to investigate the capability of processes to export bound names, rather than to actually extend the scope of bound names which is instead achieved through the structural law (EXT). Indeed, in (LTS-COMM) labels do not carry any restriction on names, whose scope must have been previously extended. (LTS-RES), (LTS-PAR) and (LTS-STRUCT) are standard.

It should not be surprising that actions **out** and **eval** yield the same label. Of course, the two actions should be kept distinct for security reasons, because accepting processes for execution is more dangerous than accepting data. However, in our setting, an external observer has not enough power to notice any difference: in both cases, it can just observe that a packet is sent. Similar considerations also hold for actions **in** and **read**.

The following proposition states that the LTS is 'correct' w.r.t. the actual operational semantics of TKLAIM, $\longmapsto$.

Proposition 1. *$N \longmapsto M$ if and only if $N \xrightarrow{\tau} M$.*

Proof. Both directions can be proved by an easy induction on the shortest inference for the judgement in the premise. □

We are now ready to give the key proposition that describes the possible interactions a net N can engage with another net K, when some names $\widetilde{l}$ are restricted.

Proposition 2. *$(\nu\widetilde{l})(N \parallel K) \xrightarrow{\alpha} \bar{N}$ if and only if one of the following conditions holds, possibly exchanging K and N:*

(1) $(\nu\widetilde{l})N \xrightarrow{\alpha} (\nu\widetilde{l'})N'$ *and* $\bar{N} \equiv (\nu\widetilde{l'})(N' \parallel K)$
(2) $N \xrightarrow{l_1 \frown l_2} N'$, $K \xrightarrow{(\nu\widetilde{l'})\ \langle t\rangle\ @\ l_1 : l_1} K'$ *and* $\bar{N} \equiv (\nu\widetilde{l''})(N' \parallel K')$, *for* $\widetilde{l''} = \widetilde{l} - fn((\nu\widetilde{l'})t)$
(3) $N \xrightarrow{l_1 : \triangleright\, l_2} N'$, $K \xrightarrow{l_1 \frown l_2} K'$, $\bar{N} \equiv (\nu\widetilde{l})(N' \parallel K')$ *and* $\alpha = \tau$
(4) $N \xrightarrow{l_2 : t \,\triangleleft\, l_1} N'$, $K \xrightarrow{(\nu\widetilde{l'})\ \langle t\rangle\ @\ l_1 : l_2} K'$, $\bar{N} \equiv (\nu\widetilde{l}, \widetilde{l'})(N' \parallel K')$ *and* $\alpha = \tau$
(5) $N \xrightarrow{l_1 \frown l_2} \xrightarrow{l_2 : t \,\triangleleft\, l_1} N'$, $K \xrightarrow{(\nu\widetilde{l'})\ \langle t\rangle\ @\ l_1 : l_1} K'$, $\bar{N} \equiv (\nu\widetilde{l}, \widetilde{l'})(N' \parallel K')$ *and* $\alpha = \tau$
(6) $N \xrightarrow{(\nu\widetilde{l'})\ \langle t\rangle\ @\ l_1 : l_1} \xrightarrow{l_2 : t \,\triangleleft\, l_1} N'$, $K \xrightarrow{l_1 \frown l_2} K'$, $\bar{N} \equiv (\nu\widetilde{l}, \widetilde{l'})(N' \parallel K')$ *and* $\alpha = \tau$.

Proof. The 'only if' part holds by definition of the LTS; the 'if' part follows by induction over the shortest inference for $(\nu\widetilde{l})(N \parallel K) \xrightarrow{\alpha} \bar{N}$. □

To conclude, we present a few equational laws that will simplify the proofs of the case-studies considered in this paper. The first three laws state a 'confluence' property for actions **out**, **eval** and **new**; the last law states that a restricted node hosting no processes is useless even if it is connected to a non-restricted node. We shall fully prove the first equivalence only; the other ones, like all the equivalences that will be found in the following, can be proved similarly.

Proposition 3.

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

Proof. Let N_{lhs} and N_{rhs} denote the left hand side and the right hand side of the first equality; we have to prove that, for every O, N_{lhs} MAY O if and only if N_{rhs} MAY O. The 'if' part is trivial, since N_{lhs} can reduce to N_{rhs} in one step. For the converse, we know that N_{lhs} MAY O, i.e. $N_{\text{lhs}} \parallel O \triangleq (\nu\widetilde{l_0})(N_0 \parallel O_0) \longmapsto \cdots \longmapsto (\nu\widetilde{l_i})(N_i \parallel O_i)$, where O_i is the first observer in $\{O_0, O_1, \ldots\}$ that reports success. The proof is by induction on i. The base step is trivial. For the inductive step, let us consider the possible interactions among N_0 and O_0 that yielded the first reduction; according to Proposition 2, we have six possibilities (the first two ones correspond to Proposition 2(1) and its symmetric; the third one corresponds to Proposition 2(3); the last two ones correspond to the symmetric of Proposition 2(3)). In all the cases, $\widetilde{l_0} = \widetilde{l_1} = \emptyset$.

(1) $N_0 \xrightarrow{\tau} N_1$ and $O_1 \equiv O_0$: in this case, the only possible τ-action of N_0 leads it to N_{rhs}; thus, $N_1 \equiv N_{\text{rhs}}$ and $N_{\text{rhs}} \parallel O \longmapsto \cdots \longmapsto (\nu\widetilde{l_i})(N_i \parallel O_i)$, i.e. N_{rhs} MAY O.
(2) $O_0 \xrightarrow{\tau} O_1$ and $N_1 \equiv N_0$: this case is simple since, by induction, N_{lhs} MAY O_1 implies that N_{rhs} MAY O_1; hence, N_{rhs} MAY O.
(3) $N_0 \xrightarrow{l : \triangleright\, l'} N_1$ and $O_0 \xrightarrow{l \frown l'} O_1$: by definition of the LTS, it must be that $l :: \mathbf{out}(t)@l'.P \xrightarrow{l : \triangleright\, l'} N_{\text{rhs}}$, $N_1 \equiv N_{\text{rhs}} \parallel \{l \leftrightarrow l'\}$ and $O_0 \equiv O_1 \parallel \{l \leftrightarrow l'\}$; thus, $N_1 \parallel O_1 \equiv N_{\text{rhs}} \parallel O$ and we easily conclude like in case (1).
(4) $O_0 \xrightarrow{l : \triangleright\, l'} O_1$ and $N_0 \xrightarrow{l \frown l'} N_1$: similarly to case (3), $O_1 \equiv O_1' \parallel \{l \leftrightarrow l'\}$ and $N_0 \equiv N_1 \parallel \{l \leftrightarrow l'\}$; thus, $N_1 \parallel O_1 \equiv N_{\text{lhs}} \parallel O_1'$ that, by induction, yields N_{rhs} MAY O_1'. The thesis follows by noting that N_{rhs} MAY $O \longmapsto N_{\text{rhs}}$ MAY O_1'.
(5) $O_0 \xrightarrow{l'' : \triangleright\, l''} O_1$ and $N_0 \xrightarrow{l'' \frown l''} N_1$, for $l'' \in \{l, l'\}$: by definition of the LTS, l'' is a node address in O; thus, by law (SELF), $O_0 \longmapsto O_1$ and we can conclude like in case (2). □

3. Implementing distant communications: A routing messenger

As we have already mentioned in Section 2.2, in our setting a process at l can perform action $\mathbf{out}(t)@l'$ only if l and l' are directly connected. We can however define a protocol to deliver t from l to l' under the assumption that there exists a path of links from l to l' in the connection graph. Remote access/retrieval of tuples and spawning of processes can be dealt with similarly.

To determine a path connecting any pair of nodes, we exploit *routing tables*. These sort of distributed data structures store information on routing paths; they are usually built at the outset by routing algorithms that, during net evolution, take care of maintaining consistency of stored information. In our basic setting links never change during the computation, thus routing tables do not change over time. To implement single entries of routing tables, we use tuples of the form $\langle$"*route*", $l, l'\rangle$. More precisely, we assume that, for each pair of (possibly indirectly) connected localities l_1 and l_2, there is a (permanent) tuple $\langle$"*route*", $l_2, l_3\rangle$ at l_1 storing the directly connected node l_3 to visit next for reaching l_2.

For the sake of readability, we shall use a conditional statement to select one of two processes for execution. In TKLAIM, it is defined as follows:

$$\mathbf{if}\ l_1 = l_2\ \mathbf{then}\ P\ \mathbf{else}\ Q \ \triangleq\ \mathbf{new}(l).\mathbf{out}(l_1 = l_2)@l.(\ \mathbf{in}(\mathtt{tt})@l.P \mid \mathbf{in}(\mathtt{ff})@l.Q)$$

where `tt` and `ff` stand for the boolean values *true* and *false*, and '=' stands for the equality test for locality names. It is easy to prove that P can evolve if and only if $l_1 = l_2$, and Q can evolve if and only if $l_1 \neq l_2$.

Proposition 4.

(1) $l' :: \mathbf{if}\ l = l\ \mathbf{then}\ P\ \mathbf{else}\ Q \simeq l' :: P$
(2) $l' :: \mathbf{if}\ l_1 = l_2\ \mathbf{then}\ P\ \mathbf{else}\ Q \simeq l' :: Q$, *whenever* $l_1 \neq l_2$.

Proof. We just examine the first claim; the second one is very similar. By Proposition 3(3) and 3(1), we have that $l' :: \mathbf{if}\ l = l\ \mathbf{then}\ P\ \mathbf{else}\ Q$ is may testing equivalent to $(\nu l'')(l' :: \mathbf{in}(\mathtt{tt})@l''.P \mid \mathbf{in}(\mathtt{ff})@l''.Q \parallel \{l' \leftrightarrow l''\} \parallel l'' :: \langle\mathtt{tt}\rangle)$, as $\mathcal{E}[\![\, l = l \,]\!] \triangleq \mathtt{tt}$. It is now easy to prove that the latter net satisfies exactly the same observers as $l' :: P$: indeed, the former can only reduce to $l' :: P$ and, before doing this, it can only offer label $l' \curvearrowright l'$ in any interaction with an observer. □

We are now ready to describe the mobile process delivering tuple t from l to l' as $Deliver_t(l, l')$, where

$$Deliver_t(u, v) \stackrel{\text{def}}{=} \mathbf{read}(\text{"}route\text{"}, v, !w)@u.\mathbf{if}\ w = v\ \mathbf{then}\ \mathbf{out}(t)@v\ \ \mathbf{else}\ \mathbf{eval}(Deliver_t(w, v))@w$$

The process gets the address of the next node to visit for reaching v and binds it to w; if the obtained address is v itself, then the current node u is directly connected to v and action $\mathbf{out}(t)@v$ is performed; otherwise, the process migrates to node w and iterates its behaviour.

To prove that execution of process $Deliver_t(l, l')$ does indeed place the result of evaluating tuple t at l', we only consider observers that do not interfere with information stored in routing tables. Thus, let $\mathcal{O}_{rt}$ be the set of observers that do not provide/emit/remove tuples of the form $\langle$"*route*", $\cdot, \cdot\rangle$. Now, correctness of $Deliver_t(l, l')$ can be formalised as follows.

Let l and l' be addresses of nodes in N, `f` be a fresh name (i.e., a name occurring nowhere else) and $t' = \mathcal{E}[\![\, t \,]\!]$. If l is connected to l' in N, then

$$N \parallel l :: Deliver_t(l, l') \simeq^{\mathcal{O}_{rt}} N \parallel l' :: \langle t'\rangle \tag{1}$$
$$N \parallel l :: Deliver_{\mathtt{f}}(l, l') \not\simeq^{\mathcal{O}_{rt}} N \tag{2}$$
$$N \parallel l :: Deliver_{\mathtt{f}}(l, l') \not\simeq^{\mathcal{O}_{rt}} N \parallel l' :: \langle t''\rangle \quad \text{for } t'' \neq \mathtt{f} \tag{3}$$

otherwise

$$N \parallel l :: Deliver_t(l, l') \simeq^{\mathcal{O}_{rt}} N \tag{4}$$
$$N \parallel l :: Deliver_{\mathtt{f}}(l, l') \not\simeq^{\mathcal{O}_{rt}} N \parallel l' :: \langle\mathtt{f}\rangle \tag{5}$$

The first three equations state that, if the target node is reachable from the source one, process *Deliver* properly forwards the tuple to its destination. Indeed, Eq. (1) states that *Deliver* may emit t' at l'; Eq. (2) states that *Deliver*

cannot avoid emitting a tuple at l'; and Eq. (3) states that the emitted tuple must be the tuple carried by *Deliver*. If source and target nodes are not (indirectly) connected, Eq. (4) states that the activity of *Deliver* is essentially the same as doing nothing, while Eq. (5) states that *Deliver* cannot emit the carried tuple at l'. Notice that, in Eqs. (2)/(3)/(5), freshness of $\mathtt{f}$ ensures that N cannot produce it; hence, the presence/absence of a tuple $\langle \mathtt{f} \rangle$ faithfully mirrors the activity of *Deliver*.

Validity of negative requirements can be easily established by providing a proper observer that distinguishes the two nets. For Eqs. (2), (3) and (5) we consider

$$O \stackrel{\text{def}}{=} \{\mathtt{test} \leftrightarrow l'\} \parallel \mathtt{test} :: \mathbf{in}(\mathtt{f})@l'.\mathbf{out}(\,)@\mathtt{test}$$

In Eqs. (2) and (3), it holds that $N \parallel l :: \mathit{Deliver}_{\mathtt{f}}(l, l')$ MAY O while, because of freshness of $\mathtt{f}$, we have N NOTMAY O and $N \parallel l' :: \langle t'' \rangle$ NOTMAY O. Similarly, in Eq. (5), we have that $N \parallel l' :: \langle \mathtt{f} \rangle$ MAY O but $N \parallel l :: \mathit{Deliver}_{\mathtt{f}}(l, l')$ NOTMAY O.

To prove validity of Eqs. (1) and (4), we first give a proposition that regulates the handling of routing information stored in the net of our example.

Proposition 5.

(1) $l :: \mathbf{read}(\text{“}\mathit{route}\text{”}, l', !u)@l.P \simeq^{O_{rt}} l :: \mathbf{nil}$
(2) $l :: \langle \text{“}\mathit{route}\text{”}, l', l'' \rangle \mid \mathbf{read}(\text{“}\mathit{route}\text{”}, l', !u)@l.P \simeq^{O_{rt}} l :: \langle \text{“}\mathit{route}\text{”}, l', l'' \rangle \mid P[l''/u]$.

Proof. The proof proceeds like the proof of Proposition 3(1). The only thing to notice is that, by interacting with the observers of the set O_{rt}, $l :: \mathbf{read}(\text{“}\mathit{route}\text{”}, l', !x)@l.P$ can only exhibit labels of the form $l \curvearrowright l$; similarly, $l :: \langle \text{“}\mathit{route}\text{”}, l', l'' \rangle \mid \mathbf{read}(\text{“}\mathit{route}\text{”}, l', !u)@l.P$ can either exhibit labels of the form $l \curvearrowright l$ or perform a τ-step to become $l :: \langle \text{“}\mathit{route}\text{”}, l', l'' \rangle \mid P[l''/u]$. □

Now, let us consider Eq. (1); we know that, if l and l' are connected, there is a path $l \triangleq l_0 \to l_1 \to \cdots \to l_n \triangleq l'$ (for $n \geq 0$) in the connection graph underlying N, that is faithfully reflected by the routing tables within N. We work by induction on n.

Base case ($n = 0$): In this case, $l = l'$ and hence

$$\begin{aligned} N \parallel l :: \mathit{Deliver}_t(l, l) & \\ & \simeq^{O_{rt}} N \parallel l :: \mathbf{if}\ l = l\ \mathbf{then}\ \mathbf{out}(t)@l\ \mathbf{else}\ \mathbf{eval}(\mathit{Deliver}_t(l, l))@l \\ & \simeq N \parallel l :: \langle t' \rangle \end{aligned}$$

The first equality is proved by using Proposition 5(2), as in the routing table at l there is the entry $\langle \text{“}\mathit{route}\text{”}, l, l \rangle$; the second equality relies on Propositions 4(1) and 3(1).

Inductive case ($n > 0$): Let $l \triangleq l_0 \to l_1 \to \cdots \to l_n \triangleq l'$. Thus

$$\begin{aligned} N \parallel l :: \mathit{Deliver}_t(l, l') & \\ & \simeq^{O_{rt}} N \parallel l :: \mathbf{if}\ l_1 = l'\ \mathbf{then}\ \mathbf{out}(t)@l'\ \mathbf{else}\ \mathbf{eval}(\mathit{Deliver}_t(l_1, l'))@l_1 \\ & \simeq \begin{cases} N \parallel l :: \langle t \rangle & \text{if } l_1 = l' \\ N \parallel l_1 :: \mathit{Deliver}_t(l_1, l') & \text{otherwise} \end{cases} \\ & \simeq^{O_{rt}} N \parallel l :: \langle t' \rangle \end{aligned}$$

The first and the second equalities when $l_1 = l'$ are proved like in the base case. The second equality when $l_1 \neq l'$ is proved by using Propositions 4(2) and 3(2). The third equality follows by a straightforward induction and by reflexivity of $\simeq$.

We are left with Eq. (4). The fact that l and l' are not connected means that in the routing table of l there is no tuple of the form $\langle \text{“}\mathit{route}\text{”}, l', \cdot \rangle$ and, of course, such a tuple will never appear. Thus, Eq. (4) follows by virtue of Proposition 5(1).

4. Modelling failures

We now enrich the basic framework with a mechanism for modelling different forms of failures. We start by modelling failure of nodes and node components, and use the new setting to prove some properties of a distributed fault-tolerant protocol. Then, we take into account failures of connections too.

4.1. Failure of nodes and node components

Failures of nodes and components can be modelled by adding to the reduction relation of Table 3 the rule

$$\text{(R-FAILN)} \qquad l :: C \longmapsto \mathbf{0}$$

This rule models *loss of tuples* 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 tuple, a process or a whole node) is a simple, but realistic, way of representing failures. Indeed, while the presence of tuples/nodes can be ascertained, their absence cannot because there is no practical upper bound to communication delays. This is true in distributed environments [16], but it is especially true in global computing scenarios [7], where failures cannot be distinguished from long delays and should be modelled as totally asynchronous and undetectable events.

Having modified the operational semantics as we have just discussed, the theory of may testing can be straightforwardly adapted; to avoid ambiguities, we shall denote with $\sqsubseteq_f$ and $\simeq_f$ the may testing preorder and equivalence arising in this new framework. Moreover, also the LTS of Section 2.3 can be easily adapted to this framework; indeed, it suffices to add a rule

$$\text{(LTS-FAILN)} \qquad l :: C \xrightarrow{\tau} \mathbf{0}$$

to those in Table 4.

A distributed fault-tolerant protocol: k-set agreement [10]. Let us consider a totally-connected distributed system with n principals relying on an asynchronous message-passing communication paradigm. The communication medium is reliable, i.e. all sent messages are received, but, due to asynchrony, the reception order is unpredictable. Each principal has an input value (taken from a totally ordered set) and must produce an 'agreed' output value. Principals can fail according to a fail-silent model of failures.

The original *agreement* problem requires to find a protocol that satisfies three properties: *agreement* (i.e. the non-faulty principals produce the *same* output value), *validity* (i.e. the output value is one of the input values) and *termination* (i.e. the non-faulty principals eventually produce an output). It is well-known [16] that a solution for this problem does not exist even in the presence of a single failure.

The *k-set agreement* problem relaxes the agreement property to guarantee existence of a solution. Indeed, for each $k \in \{1, ..., n\}$, it requires that, assuming at most $k-1$ faulty principals, the non-failed principals successfully complete their execution and produce a set of outputs whose size is at most k. Notice that for $k = 1$, the problem reduces to the agreement problem without failures.

A possible solution for the k-set agreement problem is given in [10] by letting each principal execute the following protocol:

(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 *IN* the set of the input values, the set of output values *OUT* is formed by the k smallest values in *IN*.

In our TKLAIM implementation of this protocol, we use integers as input/output values, while principals are represented as distinct nodes, whose addresses form the set $\widetilde{l} \triangleq \{l_1, \ldots, l_n\}$; l is a distinct locality used to collect output values. Moreover, we use $d_i \in IN$ to denote the input value of the principal associated to the node whose address is l_i. Given k, the process at l_i is

$$P_i^k \stackrel{\text{def}}{=} \mathbf{out}(d_i)@l_1. \ldots .\mathbf{out}(d_i)@l_n. \\ \mathbf{in}(!z_1^i)@l_i. \ldots .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l \qquad \text{where } m_i \triangleq \min\{z_1^i, \ldots, z_{n-k+1}^i\}$$

The net implementing the whole protocol is

$$N_n^k \triangleq (\nu\widetilde{l})\left(\prod_{i=1}^{n} l_i :: P_i^k \parallel N_c\right) \qquad \text{where } N_c \triangleq \prod_{i\neq j}^{1\ldots n} \{l_i \leftrightarrow l_j\} \parallel \prod_{i=1}^{n} \{l_i \leftrightarrow l\}$$

We restricted the localities associated to the principals because no external context is allowed to interfere with the execution of the protocol.

Validity and k-set agreement. These two properties can be proved at once by establishing the following equality:

$$N_n^k \simeq_f M_n^k \tag{6}$$

There, we exploit the auxiliary net

$$M_n^k \triangleq (\nu\widetilde{l})\left(\prod_{i=1}^{n} l_i :: Q_i^k \parallel N_c\right)$$

where

$$Q_i^k \stackrel{\text{def}}{=} \mathbf{out}(d_i)@l_1.\cdots.\mathbf{out}(d_i)@l_n.\mathbf{in}(!z_1^i)@l_i.\cdots.\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{if}\ m_i \in OUT\ \mathbf{then}\ \mathbf{out}(m_i)@l\ \mathbf{else}\ \mathbf{nil}$$

Here, we have generalised the if-then-else statement presented in Section 3 to check whether a value V belongs to a given set of values S. Indeed, if S is $\{V_1, \ldots, V_k\}$, this check can be implemented as follows:

$$\mathbf{if}\ V \in S\ \mathbf{then}\ P\ \mathbf{else}\ Q \triangleq \mathbf{new}(l).\mathbf{out}(V = V_1 \vee \cdots \vee V = V_k)@l.(\ \mathbf{in}(\mathtt{tt})@l.P \mid \mathbf{in}(\mathtt{ff})@l.Q\ )$$

Intuitively, Q_i^k performs the final output only if the value produced by the principal i is in OUT. The net M_n^k obviously satisfies the wanted properties, since its principals output only values in OUT. The fact that $|OUT| = k$ implies the k-set agreement property, while the fact that $OUT \subseteq IN$ implies the validity property. By following the intuition behind Eq. (3) in Section 3, we can also easily express the fact that N_n^k can only produce data of OUT at l: it is enough to replace OUT with $IN - OUT$ in Q_i^k and prove that the resulting net is not equivalent to N_n^k.

By Proposition 3(1), that also holds in the framework with failures, we have that $N_n^k \simeq_f H$ and $M_n^k \simeq_f K$, where

$$H \triangleq (\nu\widetilde{l})\left(\prod_{i=1}^{n} l_i :: \mathbf{in}(!z_1^i)@l_i.\cdots.\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l \mid \langle d_1\rangle \mid \ldots \mid \langle d_n\rangle \parallel N_c\right)$$

$$K \triangleq (\nu\widetilde{l})\left(\prod_{i=1}^{n} l_i :: \mathbf{in}(!z_1^i)@l_i.\cdots.\mathbf{in}(!z_{n-k+1}^i)@l_i.\ \mathbf{if}\ m_i \in OUT\ \mathbf{then}\ \mathbf{out}(m_i)@l\ \mathbf{else}\ \mathbf{nil} \mid \langle d_1\rangle \mid \ldots \mid \langle d_n\rangle \parallel N_c\right)$$

We now prove that $H \simeq_f K$. Since K is obtained from H by adding the test $m_i \in OUT$ before the last action, it holds that $K \sqsubseteq_f H$. To prove the converse, observe that the only actions of H that in principle K could not be able to exhibit are $\langle m_i'\rangle @ l : l$, where m_i' denotes $m_i[\widetilde{d}/\widetilde{z}]$, with $\widetilde{d} \triangleq \{d_{i_1}, \ldots, d_{i_{n-k+1}}\} \subseteq \{d_1, \ldots, d_n\}$ and $\widetilde{z} \triangleq \{z_1, \ldots, z_{n-k+1}\}$. However, as we now prove, K can perform exactly the same actions $\langle m_i'\rangle @ l : l$ as H. Suppose that H offers to an observer O the datum $\langle m_i'\rangle$ at l; then, principal i in H has collected $n-k+1$ values from $\{d_1, \ldots, d_n\}$ and has output the minimum among them. Since $|OUT| = k$, we have that $m_i' \in OUT$; thus, by letting K select the same tuples, we have that also principal i in K can output m_i' at l. This suffices to conclude $H \sqsubseteq_f K$ and proves Eq. (6).

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

$$l :: \prod_{j=1}^{n-k+1} \langle\rangle \sqsubseteq_f \hat{N}_n^k \tag{7}$$

$$\hat{N}_n^k \not\sqsubseteq_f l :: \prod_{j=1}^{n-k} \langle\rangle \tag{8}$$

where $\hat{N}_n^k \triangleq (\nu\widetilde{l})(\prod_{i=1}^n l_i :: \hat{P}_i^k \parallel N_c)$ and process $\hat{P}_i^k$ is defined like P_i^k but with action $\mathbf{out}()@l$ in place of $\mathbf{out}(m_i)@l$. Clearly, if we only consider termination, N_n^k and $\hat{N}_n^k$ 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. Thus, Eqs. (7) and (8) imply termination of the protocol, since they require that at least $n-k+1$ tuples are produced at l; by definition of the protocol, this is possible only if at least $n-k+1$ principals terminate successfully.

To prove Eq. (8), one can use the observer

$$O \stackrel{\text{def}}{=} \{\mathtt{test} \leftrightarrow l\} \parallel \mathtt{test} :: \overbrace{\mathbf{in}()@l.\cdots.\mathbf{in}()@l}^{n-k+1}.\mathbf{out}()@\mathtt{test}.$$

To prove Eq. (7), we need two new laws, collected in the following proposition. By the way, notice that the same laws would also hold for $\sqsubseteq$ and $\simeq$.

Proposition 6.

(1) *If $n \le m$, then* $l :: \langle t_1\rangle|\ldots|\langle t_n\rangle \sqsubseteq_f l :: \langle t_1\rangle|\ldots|\langle t_m\rangle$.
(2) *If $x \notin fn(P)$, then, for every $n > 0$ and $i \in \{1,\ldots,n\}$, it holds that*

$$(\nu l)\left(l :: \mathbf{in}(!x)@l.P \mid \langle V_1\rangle \mid \ldots \mid \langle V_n\rangle \parallel \prod_k \{l \leftrightarrow l_k\}\right) \simeq_f (\nu l)\left(l :: P \mid \prod_{j\neq i}^{1\ldots n} \langle V_j\rangle \parallel \prod_k \{l \leftrightarrow l_k\}\right)$$

Now,

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

The first two steps are derived by using Propositions 3(1) and 6(2); the third step is derived by using Propositions 3(1) and 3(4) (notice that also the latter one still holds in the framework with failures); the fourth step derives from Proposition 6(1).

4.2. Failure of inter-node connections

Our model of failures can be easily tailored to also deal with failures of connections. To this aim, we add the reduction rule

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

to those of Table 3. The new rule models the (asynchronous and undetectable) failure of the connection between nodes l_1 and l_2. Correspondingly, we add a rule

$$\text{(LTS-FAILC)} \quad \{l_1 \leftrightarrow l_2\} \stackrel{\tau}{\longrightarrow} \mathbf{0}$$

to those of Table 4. We still let $\sqsubseteq_f$ denote the may testing preorder in this new framework.

Discovering neighbours. In the example of Section 3 we assumed unchangeable connections; however, when the (multi)set of connections in a net can change during computations, routing tables must be dynamically updated, because the original topology can change at runtime. This task is usually carried out by so-called *adaptive* (or *dynamic*) routing algorithms. Several proposals have been presented in the literature and different standards adopt different solutions. However, all adaptive routing algorithms are executed at regular time intervals and consist of two main

phases: first, each node discovers its neighbours; then, it calculates its routing table by sharing local information with its neighbours. We present here a simple implementation of the first phase; the (more challenging) implementation of the second phase is left for future work.

Existence of connection $\{l \leftrightarrow l'\}$ can be tested by l through execution of action $\mathbf{eval}(\mathbf{nil})@l'$ which corresponds to sending a sort of 'ping' message to l'. If the 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 connection could be congested and this caused a delay to the message).

Soundness of the solution outlined above follows by proving that

$$l :: \mathbf{eval}(\mathbf{nil})@l'.\mathbf{out}(\text{“}conn\text{”}, l, l')@l \quad \sqsubseteq_f \quad \{l \leftrightarrow l'\} \parallel l :: \langle \text{“}conn\text{”}, l, l' \rangle \tag{9}$$

Indeed, if the left hand side of (9) successfully passes the test of an observer looking for a tuple $\langle \text{“}conn\text{”}, l, l' \rangle$ at l, the connection $\{l \leftrightarrow l'\}$ must exist. By working like in Proposition 3(2), we can prove that $l :: \mathbf{eval}(\mathbf{nil})@l'.\mathbf{out}(\text{“}conn\text{”}, l, l')@l \sqsubseteq_f \{l \leftrightarrow l'\} \parallel l :: \mathbf{out}(\text{“}conn\text{”}, l, l')@l$ that, by Proposition 3(1), implies (9).

5. Modelling dynamic connections

Finally, we present 'full' TKLAIM, i.e. we extend the basic language of Section 2 with dynamically modifiable connections. To this aim, we add specific actions for asking activation, for acceptance and for deactivation of a connection. Formally, we modify the BNF rules for actions of Table 1 as follows:

$$a ::= \ldots \mid \mathbf{conn}(\ell) \mid \mathbf{acpt}(\ell) \mid \mathbf{acpt}(!u) \mid \mathbf{disc}(\ell)$$

Intuitively, when executed at l, action $\mathbf{conn}(l')$ asks for activation of a connection between l and l'. When executed at l', action $\mathbf{acpt}(l)$ authorises the activation of a connection between l' and l, while action $\mathbf{acpt}(!u)$ authorises the activation of a connection between l' and any network node asking for such an activation. Finally, action $\mathbf{disc}(l')$, when executed at node l, deactivates a connection between l and l', if such a connection exists. Thus, activation of a connection requires mutual agreement; this resembles the handshake of capabilities and *co*-capabilities in Safe Ambients [23] needed to authorise movements. On the contrary, connection deactivations can be asynchronously decided by any of the involved nodes.

These intuitions are formalised by the following operational rules, that are added to those in Table 3:

$$\text{(R-CONN}_1\text{)} \quad l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{acpt}(l).Q \longmapsto l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: Q$$
$$\text{(R-CONN}_2\text{)} \quad l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{acpt}(!u).Q \longmapsto l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: Q[{}^{l}/_{u}]$$
$$\text{(R-DISC)} \quad l :: \mathbf{disc}(l').P \parallel \{l \leftrightarrow l'\} \longmapsto l :: P \parallel l' :: \mathbf{nil}$$

Notice that action $\mathbf{acpt}(!u)$ is a binder for u in the continuation. We believe that both forms of **acpt** are useful in practice. On the one hand, $\mathbf{acpt}(!u)$ can be exploited by a server willing to accept connection requests from any, initially unknown, client. On the other hand, $\mathbf{acpt}(l)$ should be used if a process is ready to activate connections only with a specific partner. Indeed, accepting connection requests from any process through $\mathbf{acpt}(!x)$ and then, after checking the partner identity, disconnecting the unwanted partners through **disc**, could expose a node to security risks because the sequence of actions is not guaranteed to be performed atomically. It is worth noticing that the form of client–server interaction enabled by $\mathbf{acpt}(!u)$ could not be flexibly implemented by resorting to a shared tuple space storing connection requests, because a connection between the node hosting the tuple space and that of a potential client should be already in place for the client be able to put its request.

The fact that $\mathbf{acpt}(l)$ cannot be simulated via an $\mathbf{acpt}(!x)$ and a possible $\mathbf{disc}(x)$ is also stressed by the following disequality:

$$l :: \mathbf{acpt}(l') \not\simeq_d l :: A \qquad \text{with } A \stackrel{\text{def}}{=} \mathbf{acpt}(!x).\mathbf{if}\ x = l'\ \mathbf{then\ nil\ else\ disc}(x).A$$

where $\simeq_d$ denotes the may testing equivalence in the language with dynamic connections. One observer that can be used to distinguish $l :: \mathbf{acpt}(l')$ and $l :: A$ is

$$O \stackrel{\text{def}}{=} \texttt{test} :: \mathbf{conn}(l).\mathbf{out}()@\texttt{test}$$

Indeed, $l :: A$ MAY O, while $l :: \mathbf{acpt}(l')$ NOTMAY O.

To properly adapt the LTS of Section 2.3 to the framework with modifiable connections, we add three new labels, corresponding to the new primitives of the language:

$$\alpha ::= \ldots \mid l_1 : ?l_2 \mid l_1 : !l_2 \mid l_1 : \neg l_2$$

These new labels are exploited by the following rules, that are added to those in Table 4:

(LTS-CONN)
$$l_1 :: \mathbf{conn}(l_2).P \xrightarrow{l_1 : ?l_2} l_1 :: P \parallel \{l_1 \leftrightarrow l_2\}$$

(LTS-ACC$_1$)
$$l_1 :: \mathbf{acpt}(!x).P \xrightarrow{l_1 : !l_2} l_1 :: P[l_2/x]$$

(LTS-ACC$_2$)
$$l_1 :: \mathbf{acpt}(l_2).P \xrightarrow{l_1 : !l_2} l_1 :: P$$

(LTS-DISC)
$$l_1 :: \mathbf{disc}(l_2).P \xrightarrow{l_1 : \neg l_2} l_1 :: P$$

(LTS-EST)
$$\frac{N_1 \xrightarrow{l_1 : ?l_2} N_1' \qquad N_2 \xrightarrow{l_2 : !l_1} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

(LTS-REM)
$$\frac{N_1 \xrightarrow{l_1 : \neg l_2} N_1' \qquad N_2 \xrightarrow{l_1 \frown l_2} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

Let us now explain the intuition behind the new labels and rules. Suppose that $N \xrightarrow{\alpha} N'$. If $\alpha = l_1 : ?l_2$, then this means that in N there is a process located at l_1 willing to activate a connection with l_2 (see (LTS-CONN)); for the actual activation, the net must also contain a node with address l_2 accepting such a request (see (LTS-EST)). Vice versa, if $\alpha = l_1 : !l_2$, then this means that in N there is a process located at l_1 willing to accept a request of connection with l_2 (see (LTS-ACC$_1$) and (LTS-ACC$_2$)); for the actual activation, the net must contain a node with address l_1 that requires such a connection (see (LTS-EST)). Finally, if $\alpha = l_1 : \neg l_2$, then this means that in N there is a process located at l_1 willing to deactivate a connection with l_2 (see (LTS-DISC)); for the actual deactivation, the net must contain the connection $\{l_1 \leftrightarrow l_2\}$ (see (LTS-REM)).

In this new framework with modifiable connections, Proposition 2 must be adapted to also include the following two possibilities:

(7) $N \xrightarrow{l_1 : \neg l_2} N'$, $K \xrightarrow{l_1 \frown l_2} K'$, $\bar{N} \equiv (\nu \widetilde{l})(N' \parallel K')$ and $\alpha = \tau$
(8) $N \xrightarrow{l_1 : ?l_2} N'$, $K \xrightarrow{l_2 : !l_1} K'$, $\bar{N} \equiv (\nu \widetilde{l})(N' \parallel K')$ and $\alpha = \tau$

Message delivery in a dynamic net. We illustrate now an application of our theory to a simplified scenario inspired by the *handover protocol*, proposed by the European Telecommunication Standards Institute for the GSM Public Land Mobile Network (PLMN). A formalisation of the protocol and its service specification can be found in [25].

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. It may happen that the BS responsible for a MS must be changed during the computation (e.g., because the MS left the area associated to the BS and entered the area associated to a different BS); this process is called 'handover'. In this case, the MSC should carry out the rearrangements needed to cope with the new situation, without affecting the end-to-end communication.

We model the key features of a PLMN in TKLAIM; however, for the sake of simplicity, several aspects will be omitted, such as the criterion used to choose a proper BS for a given MS or the event leading to a 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 $\mathtt{B}_2$). Moreover, we use a private repository at address `Table` used by `M` to store temporary information.

The process that handles the connection requests at `M` is

$$\mathit{Enter} \stackrel{\text{def}}{=} \mathbf{acpt}(!u).(\mathit{Enter} \mid \mathbf{read}(!v)@\mathtt{Table}.\mathbf{eval}(\mathbf{conn}(u))@v.\mathbf{disc}(u).\mathbf{out}(u, v)@\mathtt{Table})$$

When a new user wants to join the PLMN, it has to perform a **conn**(M) from its MS, with address, say, l_1. In actual situations, 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 scenario, however, we let M pick out a BS's address from Table. Then, the MSC activates a new connection from the chosen BS to the MS and deactivates the connection from itself to the MS. Finally, it records in Table the fact that the MS is under the control of the chosen BS.

Having entered the PLMN, the new user can send some message m to (the MS of) a remote user with address, say, l_2; this is achieved by letting the MS with address l_1 perform an action of the form **out**('send', l_2, m)@l_1. Then, the BSs controlling the MSs at l_1 and l_2 take care of delivering the message. In particular, let $\mathtt{B}_i$ be (the address of) the BS associated to l_1 and $\mathtt{B}_j$ be (the address of) the BS associated to l_2 (for $i, j \in \{1, 2\}$). Then, the message is forwarded from $\mathtt{B}_i$ to $\mathtt{B}_j$ by the process

$$Fwd_i \stackrel{\text{def}}{=} \mathbf{read}(!u, \mathtt{B}_i)@\mathtt{Table}.(Fwd_i \mid \mathbf{in}(\text{'send'}, !v, !x)@u.\mathbf{in}(v, !v')@\mathtt{Table}.\mathbf{out}(v, x)@v')$$

which is located at $\mathtt{B}_i$. This process first retrieves the address l_1 of the MS associated to $\mathtt{B}_i$; then, it collects the message at l_1 and forwards it to $\mathtt{B}_j$, i.e. the BS associated to the receiver MS at l_2. Notice that, in doing this, Fwd_i 'locks' the connection between l_2 and $\mathtt{B}_j$ (by withdrawing tuple $\langle l_2, \mathtt{B}_j\rangle$ from Table) until the message can be delivered to l_2; this is necessary to avoid a handover interfering with the delivery of the message. The message is collected by $\mathtt{B}_j$ and delivered to l_2 through the process

$$Clt_j \stackrel{\text{def}}{=} \mathbf{in}(!v, !x)@\mathtt{B}_j.(Clt_j \mid \mathbf{out}(x)@v.\mathbf{out}(v, \mathtt{B}_j)@\mathtt{Table})$$

which is located at $\mathtt{B}_j$. This process retrieves the message sent by $\mathtt{B}_i$ and passes it to the receiver MS; then, it releases the 'lock' acquired by $\mathtt{B}_i$ on the connection $\{\mathtt{B}_j \leftrightarrow l_2\}$ by putting back in Table the tuple $\langle l_2, \mathtt{B}_j\rangle$. Of course, there are also processes Fwd_j and Clt_i running at $\mathtt{B}_j$ and $\mathtt{B}_i$ respectively, but they do not play any rôle in the delivery of message m from l_1 to l_2.

The handover is handled by the MSC through the following process:

$$\begin{aligned} Hndvr \stackrel{\text{def}}{=}\ & \mathbf{in}(!u, !v)@\mathtt{Table}.(Hndvr \mid \mathbf{read}(!v')@\mathtt{Table}. \\ & \mathbf{eval}(\mathbf{disc}(u))@v.\mathbf{eval}(\mathbf{conn}(u))@v'.\mathbf{out}(u, v')@\mathtt{Table}) \end{aligned}$$

This process first selects a MS-to-BS association to be changed (as we said before, we do not model the reason why this is needed); then, it chooses a new BS, properly changes the connections between the MS and the BSs, and updates the repository at Table.

Therefore, the overall system is

$$\begin{aligned} SYS \triangleq\ & (\nu \mathtt{Table}, \mathtt{B}_1, \mathtt{B}_2)(\mathtt{M} :: Enter \mid Hndvr \parallel \mathtt{Table} :: \langle \mathtt{B}_1\rangle \mid \langle \mathtt{B}_2\rangle \\ & \parallel \mathtt{B}_1 :: Fwd_1 \mid Clt_1 \parallel \mathtt{B}_2 :: Fwd_2 \mid Clt_2 \\ & \parallel \{\mathtt{M} \leftrightarrow \mathtt{Table}\} \parallel \{\mathtt{M} \leftrightarrow \mathtt{B}_1\} \parallel \{\mathtt{M} \leftrightarrow \mathtt{B}_2\} \\ & \parallel \{\mathtt{Table} \leftrightarrow \mathtt{B}_2\} \parallel \{\mathtt{Table} \leftrightarrow \mathtt{B}_1\} \parallel \{\mathtt{B}_1 \leftrightarrow \mathtt{B}_2\}) \end{aligned}$$

Its soundness can be formulated as:

$$\begin{aligned} & (\nu l_1, l_2)(SYS \parallel l_1 :: \mathbf{conn}(\mathtt{M}).\mathbf{out}(\text{'send'}, l_2, \text{'HI'})@l_1 \\ & \qquad \parallel l_2 :: \mathbf{conn}(\mathtt{M}).\mathbf{in}(\text{'HI'})@l_2.\mathbf{out}(\,)@rcvd \parallel \{l_2 \leftrightarrow rcvd\}) \\ & \simeq_d \ SYS \parallel rcvd :: \langle\rangle \end{aligned} \tag{10}$$

This law states that the message from l_1 to l_2 is dispatched by the PLMN in any execution context. We want to remark that l_1 and l_2 have been restricted only to simplify proofs: the soundness of the protocol is not affected by the fact that the MSs are not public. By following the intuition behind Eq. (2) in Section 3, we can also express the fact that the system must produce a datum at *rcvd*; we leave this task to the reader.

To prove Eq. (10), notice that the only visible actions that both sides of Eq. (10) can perform are: $\mathtt{M} \curvearrowright \mathtt{M}$, $rcvd \curvearrowright rcvd$, $\langle\rangle$ @ $rcvd$: $rcvd$ and $\mathtt{M} : !l$, for $l \notin \{\mathtt{Table}, \mathtt{B}_1, \mathtt{B}_2, l_1, l_2\}$. Moreover, only the third one can be executed at most once; the remaining ones can be executed an unbounded number of times. By relying on inductive arguments and on the modified version of Proposition 2, this suffices to conclude the proof.

6. Conclusions and related work

We have experimented with TKLAIM, a process description language obtained by enriching KLAIM with explicit 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 of nodes and connections, and with dynamic activation/deactivation of connections. In each setting, we have used our formalisms to specify and verify some non-trivial global computing applications. We have used the may-testing preorder and equivalence to carry out proofs. Given the direct correspondence of TKLAIM with X-KLAIM, we believe that the study at the level of the process description language can be faithfully transposed to guarantee correctness of programs running on actual global computational infrastructures.

The equivalence we have introduced has allowed us to establish interesting properties of the systems taken into account. However, as a future work, we plan to define finer equivalences (e.g. bisimulation-based equivalences), that could guarantee a more stringent correspondence between specifications and implementations. We also plan to enrich connections with *weights* that would permit establishing quantitative properties of global computing programs along the lines of [11] and [14].

The use of behavioural equivalences to prove soundness of protocols is well-established in the field of process calculi; some notable examples are [1,25,29]. In particular, the handover protocol of the PLMN example has been first specified in the π-calculus, then verified algebraically [25] and finally verified by means of an automatic tool for proving π-calculus' equivalences [29]. It is worth noting that, however, our specification is radically simpler than that in [25,29]. Indeed, TKLAIM and π-calculus can be seen as formalisms standing at two different levels of abstraction: TKLAIM is *network aware* and allows users to directly exploit knowledge of the topology of the net; the π-calculus is at network level and permits to directly refer network sockets (that can be represented as communication channels). We can say that TKLAIM clearly enlightens the key features of WANs, such as process distribution, process mobility and inter-node connections. An encoding of such features in the π-calculus (or in any simpler language) would hide such features within complex process structures.

The problem of formalising and proving correctness of solutions to distributed consensus problems in the presence of failures has been tackled with process algebraic techniques also in [20]. In [20], a new process algebra with *failure detectors* [9] is introduced and properties of its operational semantics are exploited to carry out the correctness proofs. The approach followed by the authors is somewhat 'ad hoc' and heavier than ours, that instead exploits a (simpler) equational setting. Moreover, it has to be said that failure detectors are difficult to implement in a global computing scenario.

We conclude by comparing TKLAIM with other recently proposed formalisms for distributed computing from the linguistic point of view. The process language most closely related to TKLAIM is Dπ_F [19]. This is a distributed version of the π-calculus with an explicit representation of the state of the underlying network on which processes execute. The state of each node and link is affected by the failures that can occur during net evolution; processes can detect and react to such failures. Differently from TKLAIM, in Dπ_F failures are programmable (via two specific primitives, **kill** and **break**) and detectable (via the primitive **ping**); we do not consider these assumptions realistic in a global computing framework. Another notable difference is that in TKLAIM a deactivated connection can be re-established later on, via the primitives **conn/acpt**, while this is not possible in Dπ_F. Thus we have that in Dπ_F link failures are permanent, while in TKLAIM they can also be transient. We may say that TKLAIM's dynamic connections are more similar to *software connections* rather than to *physical links*; physical links are better modelled by TKLAIM's static connections (as presented in Sections 2 and 3) and by Dπ_F's links.

NOMADIC PICT [28] is a distributed and agent-based core language inspired by the π-calculus that relies on a net with flat topology where named agents can roam. Communication between agents can take place only if they are co-located. However, the language also provides a (high-level) primitive for remote communication, that transparently delivers a message to an agent not co-located with the sender. This primitive is then encoded by only using the local communication primitives via a central forwarding server. The assumption that only co-located agents can communicate is, in our opinion, not natural in a global computing scenario; moreover, it is not clear to us how the theory can be adapted to consider failures.

In DJoin [18], 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. Like in Dπ_F, failures are programmable and can be detected by processes. In our view, these choices make DJoin not a suitable model of global computers; it

is a more natural candidate for modelling the *logical* organisation of a distributed system. Similar considerations hold for the Ambient calculus [8], an elegant notation to model hierarchically structured distributed applications. Moreover, no explicit notion of failures, close to actual global computing requirements, has been developed for Ambient so far.

Acknowledgements

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

References

[1] M. Abadi, A. Gordon, Reasoning about cryptographic protocols in the Spi calculus, in: Proc. of CONCUR'97, in: LNCS, vol. 1243, Springer, 1997, pp. 59–73.
[2] R.M. Amadio, I. Castellani, D. Sangiorgi, On bisimulations for the asynchronous π-calculus, Theoretical Computer Science 195 (2) (1998) 291–324.
[3] K. Arnold, E. Freeman, S. Hupfer, JavaSpaces Principles, Patterns and Practice, Addison-Wesley, 1999.
[4] L. Bettini, R. De Nicola, Interactive mobile agents in X-KLAIM, in: SFM-05:Moby, in: LNCS, vol. 3465, Springer, 2005, pp. 29–68.
[5] M. Boreale, R. De Nicola, R. Pugliese, Trace and testing equivalence on asynchronous processes, Information and Computation 172 (2002) 139–164.
[6] M. Boreale, R. De Nicola, R. Pugliese, Basic observables for processes, Information and Computation 149 (1) (1999) 77–98.
[7] L. Cardelli, Abstractions for mobile computation, in: Secure Internet Programming, in: LNCS, vol. 1603, Springer, 1999, pp. 51–94.
[8] L. Cardelli, A. Gordon, Mobile ambients, Theoretical Computer Science 240 (1) (2000) 177–213.
[9] T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of the ACM 43 (2) (1996) 225–267.
[10] S. Chaudhuri, More choices allow more faults: Set consensus problems in totally asynchronous systems, Information and Computation 105 (1) (1993) 132–158.
[11] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, E. Tuosto, A process calculus for QoS-Aware applications, in: Proc. of COORDINATION'05, in: LNCS, vol. 3454, Springer, 2005, pp. 33–48.
[12] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: A Kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 315–330.
[13] R. De Nicola, D. Gorla, R. Pugliese, On the expressive power of KLAIM-based calculi, in: Proc. of EXPRESS'04, ENTCS 128(2) 117–130. Extended version to appear in Theoretical Computer Science.
[14] R. De Nicola, D. Latella, M. Massink, Formal modeling and quantitative analysis of KLAIM-based mobile systems, in: Proc. of SAC'05, ACM, 2005, pp. 428–435.
[15] R. De Nicola, M. Hennessy, Testing equivalence for processes, Theoretical Computer Science 34 (1984) 83–133.
[16] M.J. Fischer, N.A. Lynch, M. Paterson, Impossibility of distributed consensus with one faulty process, Journal of the ACM 32 (2) (1985) 374–382.
[17] D. Ford, T. Lehman, S. McLaughry, P. Wyckoff, T Spaces, IBM Systems Journal (August) (1998) 454–474.
[18] C. Fournet, G. Gonthier, J. Lévy, L. Maranget, D. Rémy, A calculus of mobile agents, in: Proc. of CONCUR'96, in: LNCS, vol. 1119, Springer, 1996, pp. 406–421.
[19] A. Francalanza, M. Hennessy, A theory of system behaviour in the presence of node and link failures, in: CONCUR'05, in: LNCS, vol. 3653, pp. 368–382.
[20] R. Fuzzati, M. Merro, U. Nestmann, Modelling consensus in a process calculus, in: Proc. of CONCUR'03, in: LNCS, vol. 2761, Springer, 2003.
[21] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1) (1985) 80–112.
[22] M. Hennessy, J. Riely, Resource access control in systems of mobile agents, Information and Computation 173 (2002) 82–120.
[23] F. Levi, D. Sangiorgi, Controlling interference in ambients, in: Proceedings of POPL'00, ACM, January 2000, pp. 352–364.
[24] A. Omicini, F. Zambonelli, Coordination of mobile information agents in TuCSoN, Internet Research 8 (5) (1998) 400–413.
[25] F. Orava, J. Parrow, An algebraic verification of a mobile network, Formal Aspects of Computing 4 (1992) 497–543.
[26] J. Parrow, An introduction to the pi-calculus, in: Handbook of Process Algebra, Elsevier Science, 2001, pp. 479–543.
[27] G. Picco, A. Murphy, G.-C. Roman, LIME: Linda meets mobility, in: Proc. of the 21st Int. Conference on Software Engineering, ACM, 1999, pp. 368–377.
[28] A. Unyapoth, P. Sewell, Nomadic Pict: Correct communication infrastructures for mobile computation, in: Proc. of POPL'01, ACM, 2001, pp. 116–127.
[29] B. Victor, F. Moller, The mobility workbench — a tool for the π-calculus, in: Proc. of CAV'94, in: LNCS, vol. 818, Springer, 1994, pp. 428–440.