A Semantic Theory for Global Computing Systems

Daniele Gorla Rosario Pugliese
Dipartimento di Sistemi e Informatica, Università di Firenze
email : {gorla, pugliese}@dsi.unifi.it

Abstract

We introduce cKLAIM*, a process calculus that can be thought of as a variant of the π-calculus with process distribution, process mobility and asynchronous communication through distributed repositories. Upon it, we develop a semantic theory to reason about programs. More precisely, we introduce a natural contextually defined behavioural semantics, give a coinductive characterization in terms of a labelled bisimulation and illustrate some significant laws. Then, we smoothly tune the theory to model two more concrete settings obtained by explicitly considering failures and node connections, two low-level features that in real life can affect the underlying network infrastructure and, hence, the ability of processes to perform remote operations.*

1 Introduction

Technological advances of both computers and telecommunication networks, and development of more efficient communication protocols are leading to a ever-increasing integration of computing systems and to diffusion of so called *global computers*. These are massive networked and dynamically reconfigurable infrastructures interconnecting heterogeneous, typically autonomous and mobile components, that can operate on the basis of incomplete information. Global computers are fostering a new style of distributed programming whose key principle is *network awareness*, i.e. applications have information on network characteristics and can adapt to their variation. *Code mobility* has proved to be a suitable abstraction to design and program network-aware applications. To support this programming style new commercial/prototype programming languages with suitable constructs have been designed (e.g. Agent TCL, Facile, Java, Obliq, Pict, TACOMA, Telescript/Odissey); this activity has involved several important ICT companies (e.g. Dec, General Magic, IBM, Microsoft, Mitsubishi, Sun) and academic research institutes.

Global computers spreading has required the foundational research to develop specification and analysis techniques which can be used to build safer and trustworthy systems, to demonstrate their conformance to specifications, and to analyse their behaviour. Theoretical models and calculi could provide a sound basis for constructing systems which are "sound by construction" and which behave in a predictable and analysable manner. The crux is to identify what abstractions are more appropriate and to supply foundational and effective tools to support development and certification (stating and proving correctness) of global computing applications. Several foundational languages, presented as process calculi or strongly based on them, have been developed that have improved the formal understanding of the complex mechanisms underlying network awareness and code mobility. We mention the Ambient calculus [9], the Dπ-calculus [21], the DJoin calculus [18], and KLAIM [16]. Their programming models encompass abstractions to represent the execution contexts of the net where applications roam and run, and primitive mechanisms for coordinating and monitoring the use of resources, and for supporting the specification and the implementation of security policies. This paper should be considered as a contribution to this research effort.

We focus on KLAIM [16], an experimental language with programming constructs for global computing that combines the process algebraic paradigm with the coordination-oriented one. KLAIM shares similar intentions and motivations with the other above mentioned mobile process calculi, but rests on different design choices.[1] Distinctive features are communication through distributed repositories (a very flexible model that fits global computing requirements [15, 11, 17]) and remote operations (these supply a realistic abstraction level and avoid the need to heavily resort to code mobility). KLAIM permits modelling large-scale distributed systems made up of several mobile components that can explicitly refer to the network nodes where they can place and retrieve data and processes. The nodes of a net can be thought of as physically distributed machines, or as logical partitions of the same machine, or, broadly speaking, as shared resources. Each node can be accessed through its locality name and contains a single data repository and processes in execution. KLAIM primitives allow programmers to distribute and retrieve data and processes to and from the nodes of a net. Localities can be dynamically created and communicated over the network and are handled via sophisticated scoping rules *à là* π-calculus.

[1] We refer the interested reader to [4] for a full account of motivations and advantages of the KLAIM approach and for an outline of its theoretical foundations and implementation efforts.

KLAIM has a rich set of constructs that ease the task of programming and are at the basis of the programming language X-KLAIM [5], whose run-time system [6, 14] is written in Java. In this paper we abstract the basic features of KLAIM into a process calculus that we call CKLAIM (*core* KLAIM) whose primitives permit modelling the real world accurately while enable developing effective and analytical tools to reason about program behaviours. Indeed, the bisimulation congruences we introduce in this paper are powerful means to formalize and prove properties of programs (that, once verified, can be translated into Java programs and then executed). CKLAIM can be thought of as a variant of the π-calculus [25] with process distribution, process mobility, and asynchronous remote communication through distributed repositories. The syntax of CKLAIM and its operational semantics, based on a labelled transition system (LTS) inspired by [25, 28], are given in Section 2.

In Section 3, we present (weak) behavioural congruences for CKLAIM. We first introduce a natural contextually defined behavioural semantics by following the approach put forward in [23]. Thus, we define a *reduction barbed congruence* that considers as equivalent those nets that cannot be taken apart by any *observation* during their computations in all (net) contexts. While intuitive, the definition of barbed congruence suffers from universal quantification over contexts, that makes congruence checking very hard. As it has been already done for other process calculi (e.g. [29, 1, 21, 24]), it is then important to devise proof techniques that avoid such quantification. Thus, by exploiting the information carried in the labels of the LTS, we define a more tractable characterization of barbed congruence as a non-standard labelled *bisimilarity*, thus obtaining a powerful coinductive proof technique. The definition of the bisimulations is inspired by [2, 27, 30]. Indeed CKLAIM is asynchronous (both in the communication and in the mobility paradigm) and higher-order (because process mobility is modelled by using labels containing processes); moreover, our definition is complicated by process distribution.

By exploiting bisimilarity, we will establish some equational laws; at first sight, some of them could be quite surprising. For example, let us consider the following ones

$$l :: P \parallel l' :: \mathbf{nil} \;=\; l :: \mathbf{nil} \parallel l' :: P \qquad (1)$$

$$l :: \mathbf{eval}(P)@l'.Q \parallel l' :: \mathbf{nil} \;=\; l :: P|Q \parallel l' :: \mathbf{nil} \qquad (2)$$

where P and Q are generic processes, $\mathbf{nil}$ is the process that does not perform any action, $\mathbf{eval}(P)@l'$ is the operation of spawning process P for execution at l', $l :: P$ means that P is running at l, $\parallel$ and $|$ are (net and process, resp.) parallel composition operators. Law (1) indicates that, once the net is fixed, the actual distribution of processes is irrelevant, while law (2) states that remotely executing a process is observationally equivalent to executing the process locally. These two laws seem to contradict the design principles at the basis of CKLAIM! In fact, the laws highlight that the network model adopted in CKLAIM is too abstract and ideal, e.g. no failures take place. To enable reasoning about advantages of process distribution and mobility and to permit applying the laws to find out possible optimizations, additional features of the underlying network infrastructure must enter the picture. To this aim, we extend the basic calculus with mechanisms for modelling *failures* and *node connections*. Predictably, in both the two settings the above laws will not hold true anymore.

In Section 4, we enrich CKLAIM with a mechanism for modelling corruption of data (message omission), abnormal termination of processes and failures of nodes (fail-silent). This is achieved by simply adding a rule to the LTS defining the operational semantics of CKLAIM (the definitions of the equivalences are not affected) that represents failures as disappearance of a resource (a datum, a process or a whole node). This is a simple, but realistic, way of representing failures in a global computing scenario [8]. 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. The section ends with an illustrative example of application of our semantic theory to model and prove the correctness of a distributed fault-tolerant protocol, the *k-set agreement* [13].

In Section 5, we focus on another relevant aspect of global computers, namely the underlying interconnection structure. In the setting presented so far, we implicitly assumed that the graph connecting the nodes of a net is always complete (i.e. is a clique). However, when a net is made up of millions of nodes (like the Internet), such an assumption is clearly unreasonable because of the amount of physical connections required and anyway because failures can occur. It is then more realistic to think of, and model, the underlying interconnection structure as a (not-completely connected) graph. In this case, sophisticated *routing algorithms* are needed to dynamically select, according to some given parameters, the path that a remote operation must follow. To explicitly model node interconnections, we introduce net components that represent physical (bidirectional) connections between two nodes. We show the modification necessary to the operational semantics and the behavioural theory, and a simple application of the new features.

We discuss related work in the conclusion. Due to space limitation, we sketch the proofs of major results in the Appendixes and refer the interested reader to the full paper [20] for a complete account.

2 The Process Language CKLAIM

The syntax of CKLAIM is reported in Table 1. A countable set $\mathcal{N}$ of *names* $l, l', \ldots, u, \ldots, x, y, \ldots, X, Y, \ldots$ is

NETS:
$N \ ::= \ \mathbf{0} \ \mid \ l :: C \ \mid \ (\nu l)N \ \mid \ N_1 \| N_2$

COMPONENTS:
$C \ ::= \ \langle l \rangle \ \mid \ P \ \mid \ C_1 | C_2$

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

ACTIONS:
$a \ ::= \ \mathbf{in}(p)@u \mid \mathbf{out}(u_2)@u_1 \mid \mathbf{eval}(P)@u \mid \mathbf{new}(l)$

INPUT PARAMETERS:
$p \ ::= \ u \ \mid \ !\, x$

Table 1. CKLAIM **Syntax**

assumed. Names provide the abstract counterpart of the set of *communicable* objects and can be used as localities, basic variables or process variables: we do not distinguish between these three kinds of objects. Notationally, we prefer letters $l, l', \ldots$ when we want to stress the use of a name as a locality, $x, y, \ldots$ when we want to stress the use of a name as a basic variable, and $X, Y, \ldots$ when we want to stress the use of a name as a process variable. We will use u for basic variables and localities.

Processes, ranged over by $P, Q, R, \ldots$, are the CKLAIM 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/adding data from/to node repositories, activating new threads of execution and creating new nodes. 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. Notice that $\mathbf{in}(l)@l'$ differs from $\mathbf{in}(!x)@l'$ in that the former evolves only if datum $\langle l \rangle$ is present in l', while the latter accepts any datum. Hence, $\mathbf{in}(l)@l'$ is a form of *name matching operator* derived from the LINDA [19] pattern matching.

Nets, ranged over by $N, M, \ldots$, are finite collections of nodes. 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 $\langle l \rangle$. In the net $(\nu l)N$, the scope of the name l is restricted to N; the intended effect is that if one considers the net $N_1 \parallel (\nu l)N_2$ then locality l of N_2 cannot be immediately referred to from within N_1.

Names (i.e. localities and variables) occurring in CKLAIM processes and nets can be *bound*. More precisely, prefix $\mathbf{in}(!x)@u._{\_}$ binds x in $\_$; this prefix is similar to the λ-abstraction of the λ-calculus. Prefix $\mathbf{new}(l)._{\_}$ binds l in $\_$, and, similarly, net restriction $(\nu l)_{\_}$ binds l in $\_$. 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. As usual, we say that two terms are *α-equivalent*, written $\equiv_\alpha$, if one can be obtained from the other by renaming bound names. We shall say that a name u is fresh for $\_$ if $u \notin n(\_)$. In the sequel, we shall work with terms whose bound names are all distinct and different from the free ones.

(ALPHA)	$N \equiv N'$	if $N \equiv_\alpha N'$
(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 $l \notin fn(N_1)$
(ABS)	$l :: C \equiv l :: (C \mid \mathbf{nil})$	
(REC)	$l :: \mathbf{rec}\, X.P \equiv l :: P[{}^{\mathbf{rec}\, X.P}/_X]$	
(RNODE)	$(\nu l)N \equiv (\nu l)(N \parallel l :: \mathbf{nil})$	
(CLONE)	$l :: C_1 \mid C_2 \equiv l :: C_1 \parallel l :: C_2$	

Table 2. Nets Structural Congruence

Notation 2.1 We write $A \triangleq B$ to mean that A is of the form B; this notation is used to assign a symbolic name A to the term B. We shall use notation $\widetilde{\cdot}$ to denote tuples of objects (e.g. $\widetilde{l}$ is a tuple of names). Moreover, if $\widetilde{x} = (x_1, ..., x_n)$, we shall assume that $x_i \neq x_j$ for $i \neq j$. If $\widetilde{x} = (x_1, \ldots, x_n)$ and $\widetilde{y} = (y_1, \ldots, y_m)$ then $\widetilde{x}, \widetilde{y}$ will denote the tuple of pairwise distinct elements $(x_1, \ldots, x_n, y_1, \ldots, y_m)$. When convenient, we shall regard a tuple simply as a set. 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 omit trailing occurrences of process **nil** and write $\prod_{i=1}^{n} T_i$ for the parallel composition (both '|' and '$\|$') of terms (components or nets, resp.) T_i.

The operational semantics of CKLAIM is given in terms of a structural congruence and of a labelled transition relation. For the sake of presentation, we introduce the syntactic category of *inert components*

$$I \ ::= \ \mathbf{nil} \ \mid \ \langle l \rangle$$

for grouping those components that are unable to perform any basic operation.

The *structural congruence*, $\equiv$, identifies nets which intuitively represent the same net. It is defined as the least congruence relation over nets that satisfies the laws in Table 2. Most of the laws are taken from the π-calculus (see, e.g., [28]) with law (ABS), that is the equivalent of law

(PZERO) for '|', and law (REC), that freely folds/unfolds recursive definitions. Additionally, we have law (RNODE), saying that any restricted name can be used as the locality of a node[2], and law (CLONE), that transforms a parallel between co-located processes into a parallel between nodes. Notice that commutativity and associativity of '|' can be obtained by (PCOM), (PASS) and (CLONE). In the sequel, by exploiting Notation 2.1 and law (RCOM), we shall write $(\nu\widetilde{l})N$ to denote a net with a (possible empty) set $\widetilde{l}$ of restricted localities.

The *labelled transition relation*, $\xrightarrow{\alpha}$, is defined as the least relation over nets induced by the inference rules in Table 3. Transition labels take the form

$$\begin{array}{lcl} \chi & ::= & \tau \,\big|\, (\nu\widetilde{l})\; I \mathbin{@} l \\ \alpha & ::= & \chi \,\big|\, (\nu\widetilde{l})\; C \triangleright l_1 \;\big|\; l_2 \triangleleft l_1 \end{array}$$

We will write $bn(\alpha)$ for $\widetilde{l}$ if $\alpha = (\nu\widetilde{l})\; I \mathbin{@} l$ or $\alpha = (\nu\widetilde{l})\; C \triangleright l$, and for $\emptyset$, otherwise. $fn(\alpha)$ is defined accordingly. We also use function $tgt(\cdot)$ defined on the labels as follows

$$tgt(\alpha) \triangleq \begin{cases} l & \text{if } \alpha = (\nu\widetilde{l})\; I \mathbin{@} l \,, (\nu\widetilde{l})\; C \triangleright l \\ \mathit{UNDEF} & \text{otherwise} \end{cases}$$

Let us now briefly comment on some rules of the LTS; most of them are adapted from the π-calculus [28]. Rule (EXISTS) points out existence of nodes (label $\mathbf{nil} \mathbin{@} l$) or existence of data (label $\langle l_2 \rangle \mathbin{@} l_1$). Rules (OUT) and (EVAL) express the intention of sending a component; however, the datum/process will be effectively put in the target node only if such a node is present in the net (rule (SEND)). Similarly, for an input to be performed, rules (IN) (given in an early style) and (MATCH) require the existence of the chosen datum in the target node (rule (COMM)). Rule (MATCH) says that action $\mathbf{in}(l_2)@l_1$ consumes exactly the datum $\langle l_2 \rangle$ at l_1, while rule (IN) says that action $\mathbf{in}(!\,x)@l_1$ can consume any datum at l_1. Rule (NEW) says that execution of action $\mathbf{new}(l')$ simply adds a restriction over l' to the net; from then on, a new node with locality l' can be allocated/deallocated by using law (RNODE). Rule (OPEN) signals extrusion of bound names; it can be applied only if function $tgt(\alpha)$ in its premise is defined. As in some presentation of the π-calculus, (OPEN) is used to investigate the capability of processes to export bound names, rather than to really extend the scope of bound names. To this last aim, law (EXT) is used; in fact, in rules (SEND) and (COMM) labels do not carry any restriction on names (that must have been previously extruded). Rules (RES), (PAR) and (STRUCT) are standard.

[2] Restricted names can be thought of as private network addresses, whose corresponding nodes can be activated when needed, and successively deactivated, by the owners of the resource (i.e. the nodes included in the scope of the restriction). If names would represent not only localities but also other communicable objects, the law should be slightly modified for it to deal only with bound locality names.

(EXISTS)	$l :: I \xrightarrow{I \mathbin{@} l} l :: \mathbf{nil}$
(OUT)	$l :: \mathbf{out}(l_2)@l_1.P \xrightarrow{\langle l_2 \rangle \triangleright l_1} l :: P$
(EVAL)	$l :: \mathbf{eval}(Q)@l_1.P \xrightarrow{Q \triangleright l_1} l :: P$
(IN)	$l :: \mathbf{in}(!\,x)@l_1.P \xrightarrow{l_2 \triangleleft l_1} l :: P[l_2/x]$
(MATCH)	$l :: \mathbf{in}(l_2)@l_1.P \xrightarrow{l_2 \triangleleft l_1} l :: P$
(NEW)	$l :: \mathbf{new}(l').P \xrightarrow{\tau} (\nu l')(l :: P)$
(OPEN)	$\dfrac{N \xrightarrow{\alpha} N' \quad l \in fn(\alpha) - \{tgt(\alpha)\}}{(\nu l)N \xrightarrow{(\nu l)\alpha} N'}$
(RES)	$\dfrac{N \xrightarrow{\alpha} N' \qquad l \notin n(\alpha)}{(\nu l)N \xrightarrow{\alpha} (\nu l)N'}$
(SEND)	$\dfrac{N_1 \xrightarrow{C \triangleright l} N_1' \qquad N_2 \xrightarrow{\mathbf{nil} \mathbin{@} l} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2' \parallel l :: C}$
(COMM)	$\dfrac{N_1 \xrightarrow{l_2 \triangleleft l_1} N_1' \qquad N_2 \xrightarrow{\langle l_2 \rangle \mathbin{@} l_1} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$
(PAR)	$\dfrac{N_1 \xrightarrow{\alpha} N_2 \qquad bn(\alpha) \cap fn(N) = \emptyset}{N_1 \parallel N \xrightarrow{\alpha} N_2 \parallel N}$
(STRUCT)	$\dfrac{N \equiv N_1 \quad N_1 \xrightarrow{\alpha} N_2 \quad N_2 \equiv N'}{N \xrightarrow{\alpha} N'}$

Table 3. cKLAIM Operational Semantics

cKLAIM adopts a LINDA-like [19] communication mechanism; thus, 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. On the contrary, a sort of synchronization takes place between a sending process and its target node (because existence of the node is verified before sending data to it). Similarly, a form of synchronization takes place between the node hosting a datum and the process looking for it.

As a matter of notation, we let $\Rightarrow$ to stand for $\xrightarrow{\tau}{}^*$, $\stackrel{\alpha}{\Rightarrow}$ to stand for $\Rightarrow \xrightarrow{\alpha} \Rightarrow$, and $\stackrel{\hat{\alpha}}{\Rightarrow}$ to stand for $\Rightarrow$, if $\alpha = \tau$, and for $\stackrel{\alpha}{\Rightarrow}$, otherwise.

3 Behavioural Congruences

We start by introducing a natural contextually defined behavioural semantics by following the approach put forward in [23]. Thus, we first define a notion of *observation* and then rely on it to define a *net congruence* relation that

considers as equivalent those nets that cannot be taken apart by any observation during their computations in any (net) *context*.

Definition 3.1 (Basic Observables and Net Contexts)
Predicate $N \downarrow l$ holds true if and only if $N \equiv (\nu \widetilde{l})(N' \parallel l :: \langle l' \rangle)$ for some $\widetilde{l}$, N' and l' such that $l \notin \widetilde{l}$.

Predicate $N \Downarrow l$ holds true if and only if $N \Rightarrow N'$ for some N' such that $N' \downarrow l$.

A net context $\mathcal{C}[\cdot]$ *is a* CKLAIM *net with an occurrence of a hole $[\cdot]$ to be filled with any net. Formally,*

$$\mathcal{C}[\cdot] \ ::= \ [\cdot] \ \mid \ N \parallel \mathcal{C}[\cdot] \ \mid \ (\nu \widetilde{l})\mathcal{C}[\cdot]$$

Definition 3.2 (Reduction Barbed Congruence)
Reduction barbed congruence, $\cong$, *is the largest symmetric relation between* CKLAIM *nets such that for each $N_1 \cong N_2$ it holds that:*

1. *if $N_1 \downarrow l$ then $N_2 \Downarrow l$*
2. *if $N_1 \xrightarrow{\tau} N_1'$ then $N_2 \Rightarrow N_2'$ and $N_1' \cong N_2'$*
3. *for all net context $\mathcal{C}[\cdot]$, it holds that $\mathcal{C}[N_1] \cong \mathcal{C}[N_2]$*

We have chosen the basic observables by taking inspiration from the corresponding ones of the asynchronous π-calculus [2]. One may wonder if our choice is "correct" and argue that there are other alternative notions of basic observables that seem quite natural. A first alternative could be to consider as equivalent two nets if they have the same set of free node localities. A second alternative could be to consider as equivalent two nets if they make available the same set of data, possibly in different nodes. A third alternative comes out from bringing together the previous two ones, thus we could consider as equivalent two nets if they have exactly the same data at the same localities. In the full paper [20], we prove that all the reduction barbed congruences induced by these three alternative observables coincide with our congruence (except for the first one, that is too coarse). This means that our results are largely independent from the observable chosen and supports our choice.

As we said in the Introduction, barbed congruence of two nets is hard to prove because its definition exploits a closure under all possible net contexts. We now introduce a coinductive proof technique that will turn out to exactly capture barbed congruence. We use the information carried in the labels of the LTS to throw away the universal quantification over the net contexts which is part of the definition of barbed congruence. In this way, we obtain an alternative characterization of $\cong$ in terms of a (non-standard) labelled *bisimilarity*.

Definition 3.3 (Bisimilarity) *A symmetric relation $\Re$ between* CKLAIM *nets is a* (weak) bisimulation *if for each $N_1 \Re N_2$ it holds that:*

1. *if $N_1 \xrightarrow{\chi} N_1'$ then $N_2 \stackrel{\hat{\chi}}{\Longrightarrow} N_2'$ and $N_1' \Re N_2'$;*
2. *if $N_1 \xrightarrow{(\nu \widetilde{l})\ C \triangleright l} N_1'$ then $N_2 \parallel l :: \mathbf{nil} \Rightarrow N_2'$ and $(\nu \widetilde{l})(N_1' \parallel l :: C) \ \Re \ N_2'$*
3. *if $N_1 \xrightarrow{l_2 \triangleleft l_1} N_1'$ then $N_2 \parallel l_1 :: \langle l_2 \rangle \Rightarrow N_2'$ and $N_1' \parallel l_1 :: \mathbf{nil} \ \Re \ N_2'$.*

Bisimilarity, $\approx$, *is the largest bisimulation.*

Our bisimulation is somehow inspired by that in [27]. The key idea is that, since sending operations are asynchronous, a sending operation by a net N_1, say $N_1 \xrightarrow{(\nu \widetilde{l})\ C \triangleright l} N_1'$, can be simulated by a net N_2 (in a context where the locality l is provided) through execution of some internal actions that lead to N_2'. Indeed, since we want our bisimulation to be a congruence, a context that provides the target locality of the sending action must not tell apart N_1 and N_2. Hence, for $N_1 \parallel l :: \mathbf{nil}$ to be simulable by $N_2 \parallel l :: \mathbf{nil}$, it must hold that, upon transitions, $(\nu \widetilde{l})(N_1' \parallel l :: C)$ is simulable by N_2'. Similar considerations hold also for the case of the input actions (third item of Definition 3.3), but the context now is $[\cdot] \parallel l_1 :: \langle l_2 \rangle$.

In both cases we do not need to add extra requirements on the reducts because the recursive closure of $\approx$ implicitly takes into account their behaviours. This is similar to the higher order bisimulation of [30], but there the sent processes are replicated since they are the object of a communication and could be put in execution several times.

We can now state our main result (a sketch of the proof can be found in the Appendix A), stating that $\approx$ exactly captures $\cong$.

Theorem 3.4 (Alternative characterization) $\approx \ = \ \cong$.

We end this section by presenting some illustrative equational laws that can be easily proved sound by using bisimilarity and argue on their practical applicability.

Proposition 3.5 *The following laws do hold:*

a. $l :: P \parallel l' :: \mathbf{nil} \ \approx \ l :: \mathbf{nil} \parallel l' :: P$
b. $l :: \mathbf{eval}(P)@l'.Q \parallel l' :: \mathbf{nil} \ \approx \ l :: P|Q \parallel l' :: \mathbf{nil}$
c. $(\nu l)(l :: I_1|...|I_k) \ \approx \ \mathbf{0}$
d. $(\nu l)(l :: \mathbf{nil} \parallel l' :: \mathbf{in}(!x)@l.P) \ \approx \ l' :: \mathbf{nil}$, *if* $l' \neq l$
e. $(\nu l)(l :: \langle l_1 \rangle \parallel l' :: \mathbf{in}(l_2)@l.P) \ \approx \ l' :: \mathbf{nil}$, *if* $l' \neq l$ *and* $l_2 \neq l_1$
f. $l' :: \mathbf{rec}\, X.\mathbf{in}(!x)@l.\mathbf{out}(x)@l.X \approx \ l' :: \mathbf{nil}$

Distribution Transparence and Mobile Processes. Property 3.5.a gives an evidence of the fact that the model of process distribution we adopt is (for the moment) too abstract and ideal (e.g. no failures take place), thus, once the net is fixed, the actual distribution of processes is irrelevant. Property 3.5.b is derived from the previous one but gives

deeper information about mobile code applications. The law states that, in our ideal model of distribution, remotely spawning and executing a mobile process is observationally equivalent to executing the process at the sending node. Indeed, the major advantages of mobile code are efficiency (because the burden of the execution of P is not charged to l; thus l can execute Q faster), reduced network load (because code mobility can reduce the number of remote operations), and possibility of performing disconnected operations (because l can disconnect itself from the net and retrieve P's results whenever it will reconnect again). All these features make a difference when we consider more concrete settings, like those obtained by introducing failures or connections. In both cases, the two laws will fail.

Perfect Firewall and Deadlocks. Similarly to [9], Property 3.5.c states that a restricted locality acts like a perfect firewall (i.e. its presence does not influence the computation) if it only hosts inert components and no other nodes try to interact with it. This fact can be generalized to a scenario where one or more processes try to input some data from a restricted locality but

- either no data is present (Property 3.5.d),
- or only non-matching data are present (Property 3.5.e).

These are two special cases where a deadlock can occur within a distributed system: indeed, the input prefix acts as a blocking action for P since no admissible data will ever be available at the restricted locality l.

Asynchrony. Property 3.5.f is inspired by [2] and states that communication in CKLAIM is asynchronous. This law also motivated the choice to omit the LINDA action **read** when reducing KLAIM to CKLAIM. In fact, action **read** is relevant, e.g., for security reasons (removing a datum while reading it via an **in** requires a different capability than simply accessing it via a **read**) that are ignored in this paper.

4 Modelling Failures

We now enrich CKLAIM with a mechanism for modelling various forms of failures. This is achieved by adding to the LTS of Table 3 the rule

$$(\text{FAIL}) \qquad l :: C \xrightarrow{\tau} \mathbf{0}$$

This rule models corruption of data (message omission) if $C \triangleq \langle d_1\rangle | \ldots | \langle d_n\rangle$, node (fail-silent) failure if $l :: C$ collects all the clones of l, and abnormal termination of some processes running at l otherwise.

The recursive closure of both barbed congruence and labelled bisimulation already forces the corruption of the same data and the failure of the same nodes to take place at the same time; as regards process abnormal termination, it will be the evolution of the involved nets that will affect the equivalence. Therefore, we do not need to change the definitions of the equivalences; we simply let $\approx_f$ and $\cong_f$ to denote the labelled bisimilarity and the barbed congruence in the calculus with failures.

Theorem 4.1 $\approx_f \; = \; \cong_f$.

Before applying the theory presented so far to a relevant example, we want to comment on the choice of modelling failures as disappearance of a resource (a datum, a process or a whole node). This is a simple, but realistic, way of representing failures, specifically *fail-silent* and *message omission*, in a global computing scenario [8]. 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.

Notice that, among the failure models consistent with global computing, only *bizantine failures* (where arbitrary behaviours may take place during the computation) cannot be implemented in our setting. A possible way to repair this could be to introduce the 'symmetric' counterpart of (FAIL):

$$\mathbf{0} \xrightarrow{\tau} l :: C$$

However, this rule is too liberal: it could be easily proved that $\approx$ would relate all CKLAIM nets. A more controlled behaviour inspired by [22] would be modelled by the rule

$$\mathbf{0} \xrightarrow{C \triangleright l} \mathbf{0}$$

However, proving $\approx$ would become as hard as proving $\cong$ (because, the corresponding of Definition 3.3.2 would require to consider infinite possible moves for N_1). In conclusion, modelling bizantine failures requires further study and we leave it for future work.

We end this section by applying our coinductively defined bisimulation to verify the correctness of *k–set agreement* [13], a simple distributed fault-tolerant protocol. To this aim, we will firstly present the problem and describe a possible solution; we then implement the protocol as a CKLAIM net and formalize the correctness of the protocol in terms of net equivalences. The detailed proofs of these equivalences are given in the full paper [20]; a proof sketch is in Appendix B.

Let us start by describing the problem and a possible solution by following [3]. Suppose to have an asynchronous message-passing totally-connected distributed system with n principals; each principal has an input value (taken from a totally ordered set) and must produce an output value. The principals can fail and we adopt 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. 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. all 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. [3]) 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 that there are at most $k-1$ principals failing during the execution of the protocol, 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, executed by each principal:

- send the input value to all principals (including yourself)
- wait to receive $n - k + 1$ values
- 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 even if input values are duplicated – in this case $\mathcal{I}$ and $\mathcal{O}$ are multisets).

Before giving the details of our solution, 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* [12, 3]. Recently, one such solution has been formalized and proved sound by using a process algebraic approach [26]. The solution presented 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.

For the sake of readability, we let integers to play the role of the input/output values and assume a way of finding the minimum in a set of integers. These features do not increase the expressive power of the calculus, since integers (and operations over them) can be encoded in CKLAIM by properly adapting their π-calculus encodings. With abuse of notation, we shall write $\mathbf{out}(Expr)@l$ to mean

$$evaluate\ \ Expr\ \ to\, value\, d\,.\,\mathbf{out}(d)@l$$

and similarly for $\mathbf{in}(Expr)@l$. We will always use totally defined operations, thus no action will be ever blocked by the evaluation phase (i.e. the evaluation always succeeds).

We now present the implementation of the protocol. First of all, the principals are represented as distinct nodes, whose addresses are taken from the set $\widetilde{l} \triangleq \{\, l_1, \ldots, l_n\}$; moreover, we let $d_i \in \mathcal{I}$ to be the input value of the principal associated to the node whose address is l_i. Once we fix the value for k, the node l_i hosts the process

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

with $m_i \triangleq min\{\, z_j^i \ :\ j = 1, \ldots, n-k+1\,\}$. Thus, the net implementing the whole protocol is

$$N_n^k \ \triangleq\ (\nu\widetilde{l})\big(\ \prod_{i=1}^{n}\ l_i :: P_i^k\big)$$

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 $\widetilde{l}$, no **out** prefix will ever block P_i^k (because of law (RNODE)); however, this does not exclude the possibility of failures, since the failure of P_i^k is indeed the failure of principal i.

A possible formulation of the three properties for the k–set agreement problem is given by Equations (3) and (4) below. The formalization of k–set agreement and validity properties is given by the Equation

$$N_n^k \ \approx_f\ M_n^k \tag{3}$$

There, we exploit the auxiliary net

$$M_n^k \ \triangleq\ (\nu\widetilde{l}, \widetilde{l'})\big(\ \prod_{i=1}^{n}\ (\, l_i :: Q_i^k \ \parallel\ l_i' :: \prod_{w \in \mathcal{O}} \langle w \rangle\,)\big)$$

where

$$\begin{aligned} Q_i^k \ \triangleq\ & \mathbf{out}(d_i)@l_1.\ \ldots\ .\mathbf{out}(d_i)@l_n.\mathbf{in}(!z_1^i)@l_i. \\ & \ldots\ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{in}(m_i)@l_i'.\mathbf{out}(m_i)@l \end{aligned}$$

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 $\mathcal{O}$), while the last **in** action of Q_i^k is a test for checking that the output value produced by the principal i is in $\mathcal{O}$. The net M_n^k obviously satisfies the wanted properties since its principals output only values present in $\mathcal{O}$. The fact that $|\mathcal{O}| = k$ then implies the k–set agreement property, while the fact that $\mathcal{O} \subseteq \mathcal{I}$ implies validity.

In order to prove the termination property, one can try to establish that $N_n^k \parallel l :: \mathbf{nil}\ \approx_f\ l :: \prod_{j=1}^{n-h} \langle w_j \rangle$ where h is the number of failed processes and $\{w_j : j = 1, \ldots, n-h\}$ is the set of the output values of the non-failed principals. Unfortunately, this equation *does not* hold in general, since h and the w_js depend on the chosen execution. A simple way to overcome this problem is to fix h and prove that

$$\begin{aligned} & \textit{If exactly}\ \ h\ (< k)\ \textit{principals are faulty,} \\ & \textit{then}\ \ \hat{N}_n^k \parallel l :: \mathbf{nil}\ \approx_f\ l :: \prod_{j=1}^{n-h} \langle\rangle \end{aligned} \tag{4}$$

We let $\hat{N}_n^k$ to be defined like N_n^k but using processes $\hat{P}_i^k$. $\hat{P}_i^k$ is defined like P_i^k but action $\mathbf{out}(m_i)@l$ is replaced by $\mathbf{out}()@l$. Clearly, if we just 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.

5 Modelling Connections

For explicitly modelling physical (bidirectional) connections between two CKLAIM nodes, we add the production

$$N ::= \ldots \mid \{l_1 \leftrightarrow l_2\}$$

to the syntax in Table 1. We permit duplicates of the same connection, since the same two nodes could be connected by using different physical links; hence the net $N \parallel \{l_1 \leftrightarrow l_2\} \parallel \{l_1 \leftrightarrow l_2\}$ is allowed. The semantics relies on the following structural laws that intuitively say that connections are bidirectional, that nodes are self-connected and that if there exists a connection $\{l_1 \leftrightarrow l_2\}$ then nodes l_1 and l_2 do exist.

$$\begin{array}{rl} (\text{BIDIR}) & \{l_1 \leftrightarrow l_2\} \equiv \{l_2 \leftrightarrow l_1\} \\ (\text{SELF}) & l :: \mathbf{nil} \equiv \{l \leftrightarrow l\} \\ (\text{CONNODE}) & \{l_1 \leftrightarrow l_2\} \equiv \{l_1 \leftrightarrow l_2\} \parallel l_1 :: \mathbf{nil} \end{array}$$

Moreover, we modify the structural law (RNODE) according to the intuition that a restricted name is a private, always available resource of the processes knowing that name.

$$(\text{RNODE'}) \qquad \frac{N \equiv (\nu \widetilde{l})(N' \parallel l' :: P) \qquad l \in fn(P)}{(\nu l)N \equiv (\nu l)(N \parallel \{l \leftrightarrow l'\})}$$

The operational semantics is modified by adding the rule

$$(\text{CONN}) \qquad \{l_1 \leftrightarrow l_2\} \xrightarrow{l_1 \rightarrow l_2} \{l_1 \leftrightarrow l_2\}$$

to the LTS in Table 3. Label $l_1 \rightarrow l_2$ describes the structure of the net and hence it must be faithfully replied to in the bisimulation game; thus $\chi ::= \ldots \mid l_1 \rightarrow l_2$. We let $fn(l_1 \rightarrow l_2) = \{l_1, l_2\}$, $bn(l_1 \rightarrow l_2) = \emptyset$ and $tgt(l_1 \rightarrow l_2)$ to be undefined. For taking connections into account when performing remote operations, rules (SEND) and (COMM) must be modified so that they can be applied only if the node where an action is performed and the target node of the action are directly connected (later on we will investigate the general case of indirect connections). For keeping track of the node where an action is performed, we introduce rule

$$(\text{SOURCE}) \qquad \frac{l :: a.P \xrightarrow{\alpha} l :: Q}{l :: a.P \xrightarrow{l : \alpha} l :: Q}$$

and let $fn(l : \alpha) = \{l\} \cup fn(\alpha)$ and $bn(l : \alpha) = bn(\alpha)$. Now, we can define the new rules (SEND) and (COMM).

$$(\text{SEND'}) \qquad \frac{N_1 \xrightarrow{l_1 : C \triangleright l_2} N_1' \quad N_2 \xrightarrow{l_1 \rightarrow l_2} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2' \parallel l_2 :: C}$$

$$(\text{COMM'}) \qquad \frac{N_1 \xrightarrow{l_1 : l \triangleleft l_2} N_1' \quad N_2 \xrightarrow{l_1 \rightarrow l_2} \xrightarrow{\langle l \rangle \ @ \ l_2} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

We let $\cong_c$ to be the reduction barbed congruence arising when considering this new LTS. We can now define the corresponding bisimilarity and state their coincidence (a sketch of the proof can be found in the full paper [20]).

Definition 5.1 (Connection-based Bisimilarity) *A symmetric relation $\Re$ between* CKLAIM *nets is a* connection-based (weak) bisimulation *if for each $N_1 \Re N_2$ it holds that:*

1. *if $N_1 \xrightarrow{\chi} N_1'$ then $N_2 \stackrel{\hat{\chi}}{\Longrightarrow} N_2'$ and $N_1' \,\Re\, N_2'$*
2. *if $N_1 \xrightarrow{l_1 : (\nu \widetilde{l})\ C \triangleright l_2} N_1'$ then $N_2 \parallel \{l_1 \leftrightarrow l_2\} \Rightarrow N_2'$ and $(\nu \widetilde{l})(N_1' \parallel l_2 :: C \parallel \{l_1 \leftrightarrow l_2\}) \ \Re \ N_2'$*
3. *if $N_1 \xrightarrow{l_1 : l \triangleleft l_2} N_1'$ then $N_2 \parallel l_2 :: \langle l \rangle \parallel \{l_1 \leftrightarrow l_2\} \Rightarrow N_2'$ and $N_1' \parallel \{l_1 \leftrightarrow l_2\} \ \Re \ N_2'$*

Connection-based bisimilarity, $\approx_c$, *is the largest connection-based bisimulation.*

Theorem 5.2 $\approx_c \ = \ \cong_c$.

To conclude this section, we present an application that exploits the new features of the calculus. In the setting we introduced, a process at l can perform action $\mathbf{out}(d)@l'$ only if l and l' are directly connected. We now supply a protocol to deliver d from l to l' under the assumption that there exists a path of connections from l to l' in the connection graph. For the sake of simplicity, we shall rely on polyadic communication: thus *tuples* of names, ranged over by t, will be used as basic data. Data will be retrieved by using *pattern matching*. A *pattern* T is a sequence of names u and bound names $!x$; $\mathbf{in}(T)@l.P$ matches against $l :: \langle t \rangle$ if T and t have the same number of fields and corresponding fields match (i.e. two names match if they are identical, while a bound name matches against any name). The result of the communication is that bound names in T are replaced in P with the corresponding names of t. In the full paper [20], we prove that polyadic communication does not add expressivity to the language.

For the sake of readability, we define two derived constructs: an operation for accessing tuples without removing them and a conditional for selecting one between two processes for execution while discarding the other. Thus, we let $\mathbf{read}(!x)@l.P$ to denote $\mathbf{in}(!x)@l.\mathbf{out}(x)@l.P$; the polyadic version is obtained similarly. The conditional construct is defined as:

$$\begin{array}{l} \mathbf{if}\ u = u'\ \mathbf{then}\ P\ \mathbf{else}\ Q \ \triangleq \\ \quad \mathbf{new}(l).\mathbf{eval}\big(\,\mathbf{out}(res, \mathtt{ff})@l.\mathbf{out}(u)@l.\mathbf{in}(u')@l. \\ \qquad\qquad \mathbf{in}(res, \mathtt{ff})@l.\mathbf{out}(res, \mathtt{tt})@l\,\big)@l. \\ \quad (\ \mathbf{in}(res, \mathtt{tt})@l.P \mid \mathbf{in}(res, \mathtt{ff})@l.Q\ ) \end{array}$$

where we assume res, $\mathtt{tt}$ and $\mathtt{ff}$ to be reserved fresh names. It can be easily proved that if $u \neq u'$ then only Q can evolve; however, if $u = u'$ then exactly one between

P and Q can evolve (according to the chosen interleaving of actions). This behaviour slightly differs from that of a standard if-then-else; however, for our purposes, it suffices.

We assume that, for each pair of (possibly indirectly) connected localities l_1 and l_2, there is a (permanent) tuple $\langle l_2, l_3 \rangle$ at l_1 recording the next directly connected node l_3 to visit for reaching l_2.[3] Now, let the mobile process delivering datum d from l to l' be

$$Deliver(d, l, l') \triangleq$$
$$\mathbf{new}(l'').\mathbf{out}(l)@l''.\mathbf{rec}\, X.\mathbf{in}(!x)@l''.\mathbf{read}(l', !y)@x.$$
$$\mathbf{if}\ y = l'\ \mathbf{then}\ \mathbf{out}(d)@l'\ \mathbf{else}\ \mathbf{out}(y)@l''.\mathbf{eval}(X)@y$$

Intuitively, the restricted locality l'' acts as a repository storing the locality where the process is currently running. The recursive part first retrieves the current locality x, then gets the next node y to visit before reaching l'; if such a node is l' itself, then the current node is directly connected to l' and action $\mathbf{out}(d)@l'$ ends the process, otherwise the process migrates to node y and iterates its behaviour.

Soundness of the protocol can be formalized as follows. Let l' be the address of a node in N. If l is connected to l' in N, then $N \parallel l :: Deliver(d, l, l') \approx_c N \parallel l' :: \langle d \rangle$; otherwise, $N \parallel l :: Deliver(d, l, l') \approx_c N \parallel l :: \mathbf{nil}$. The proofs can be found in the full paper [20].

6 Conclusions and Related Work

We have presented the calculus CKLAIM together with its semantic theory. Devising equivalences for CKLAIM has been a non trivial task; there are other possible definitions and a comprehensive study of these choices will be essential (a preliminary study can be found in the full paper). For example, different equivalence notions could be obtained by relying on the same observables we defined in this paper and by using the general approach of [7]. The CKLAIM setting has proved to easily fit for modelling failures and node connections. However, there is still some work to be done. For example, further mechanisms should be devised for modelling other forms of failures, like the bizantine ones. Another topic worth of investigation is how to enrich the simple (static) model of node connectivity considered here with the ability of dynamically establishing and managing connections. This feature would be desirable because disconnections and intermittent connections are essential ingredients of global computing applications and could be exploited to model mobile computing. It would also be interesting to analyze efficiency issues when dealing with routing scenarios (some ideas can be found in the full paper). Finally, we have been mainly interested in studying low-level features that in real life can affect the *effective* interconnection network of global computers. We leave for future work the study of abstractions, e.g. administrative domains and security policies, that determine *virtual* networks on top of the effective one. To this aim, dynamically evolving type environments could be exploited to constraint processes ability to perform actions.

We conclude by reviewing related work on defining observational equivalences for calculi with process distribution and mobility (many of them are surveyed in [10]). In the nineties, many CCS-like process calculi have been enriched with localities to explicitly describe the distribution of processes. The aim was mainly to provide these calculi with non interleaving semantics or, at least, to differentiate processes' parallel components (thus obtaining more inspective semantics than the interleaving ones). This line of research is far from the one in which CKLAIM falls, where localities are used as a mean to make processes network aware thus enabling them to refer to the network locations as target of remote communication or as destination of migrations. Localities are not only considered as units of distribution but, according to the case, as units of mobility, of communication, of failure or of security.

[29] and [1] extend, resp., CCS and π-calculus with process distribution and mobility. In both cases, processes run over the nodes of an explicit, flat and dynamically evolving net architecture. Nodes can fail thus causing loss of all hosted processes. There are explicit operations to kill nodes and to query the status of a node. The failures model adopted is fail-stop, thus failures can be detected. This is suitable for distributed computing but clashes with the assumptions underlying global computing. The operational semantics uses information on the state of nodes but it is otherwise very close to that of CCS/π-calculus. In the first paper, non-standard bisimulations are defined that capture barbed congruence and a *symbolic* characterization of the relations is given that improves their tractability. In the second paper, an asynchronous bisimulation is defined that coincide with barbed equivalence: this is done by defining an encoding into the π_1-calculus (a typed version of the asynchronous π-calculus [2] where for each channel c there is only one process that can read along c).

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. Over the LTS defining the semantics of the calculus, a typed bisimulation (with a tractable formulation) is defined that exactly capture typed barbed equivalence. The use of types illustrates the importance of having the rights to observe a given behaviour: indeed, different typings (i.e. observation rights) generate different bisimulations, that are finer as long as the typing is less restrictive. No low-level features, like failures and con-

[3] The main goal of *routing algorithms* is to build this data structure (called *routing table*) at the outset and to maintain its consistency during net evolution. In our setting, connections do not change during net evolution: this simplifies the example, in that the routing table is calculated once and for all at the outset.

nections, are considered; thus, remote operations are always enabled if the corresponding capabilities are owned.

In the Distributed Join calculus [18], located mobile processes are hierarchically structured and form a tree-like structure evolving during the computation. Entire subtrees, and not only single processes, can move and fail. Some interesting laws and properties are proved using a contextual barbed equivalence, but no tractable characterization of the equivalence is explicitly given.

The Ambient calculus [9] is an elegant notation to model hierarchically structured distributed applications. Though the definition of its reduction semantics is very simple, the formulation of a reasonable, possibly tractable, observational equivalence is a very hard task. The calculus is centered around the notion of connections between ambients, that are containers of processes and data. Each 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. This fact greatly complicates the definition of a tractable equivalence. Recently, in [24], a bisimulation capturing Ambient's barbed congruence has been defined. This has been done by structuring the syntax into two levels, namely processes and nets (where the latter ones are particular cases of the former ones), and by exploiting an involved LTS (using three different kinds of labels). However, the defined bisimulation is not standard and still suffers from a quantification over all the possible processes (to fill the 'holes' generated by the operational semantics). To the best of our knowledge, no notion of failure has ever been introduced in the Ambient calculus.

References

[1] R. M. Amadio. On modelling mobility. *Theoretical Computer Science*, 240(1):147–176, 2000.

[2] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[3] H. Attiya and J. Welch. *Distributed Computing*. McGraw Hill, 1998.

[4] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The KLAIM project: Theory and practice. In C. Priami, editor, *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, number 2874 in LNCS. Springer-Verlag, 2003.

[5] L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 110–115, 1998.

[6] L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java Package for Distributed and Mobile Applications. *Software — Practice and Experience*, 32:1365–1394, 2002.

[7] M. Boreale, R. D. Nicola, and R. Pugliese. Basic observables for processes. *Information and Computation*, 149(1):77–98, 1999.

[8] 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.

[9] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[10] I. Castellani. Process algebras with localities. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 945–1045. Elsevier Science, 2001.

[11] S. Castellani, P. Ciancarini, and D. Rossi. The ShaPE of ShaDE: a coordination system. Technical Report UBLCS 96-5, Dip. di Scienze dell'Informazione, Univ. di Bologna, Italy, 1996.

[12] T. Chandra and S.Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[13] S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132–158, 1993.

[14] Concurrency and Mobility Group at Dipartimento di Sistemi e Informatica, Università di Firenze. KLAIM web page. `http://music.dsi.unifi.it`.

[15] 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.

[16] 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.

[17] D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proc. of ISADS 2001*, pages 278–286. IEEE, 2001.

[18] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.

[19] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[20] D. Gorla and R. Pugliese. A semantic theory for global computing system. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2003. Available at `http://rap.dsi.unifi.it/~pugliese/DOWNLOAD/bis4k-full.pdf`.

[21] M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In *Proceedings of FoSSaCS '03*, volume 2620 of *LNCS*, pages 282–299. Springer, 2003.

[22] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer, July 1991.

[23] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[24] M. Merro and F. Z. Nardelli. Bisimulation proof methods for mobile ambients. *Proceedings of ICALP'03*, volume 2719 of *LNCS*, pages 584–598. Springer, 2003.

[25] R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.

[26] U. Nestmann, R. Fuzzati, and M. Merro. Modelling Consensus in a Process Calculus. In *Proc. of CONCUR'03*, volume 2761 of *LNCS*. Springer-Verlag, 2003.

[27] U. Nestmann and B. C. Pierce. Decoding choice encodings. *Journal of Information and Computation*, 163:1–59, 2000.

[28] J. Parrow. An introduction to the pi-calculus. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.

[29] J. Riely and M. Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 266:693–735, 2001.

[30] D. Sangiorgi. Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). *Theoretical Computer Science*, 253(2):311–350, Feb. 2001.

A Proof Sketch of Theorem 3.4

We now present the steps needed to prove the main theorem of this paper. The proofs of the following claims can be found in the full paper [20]. Next proposition relates labels of the LTS with the structure of the net performing the corresponding action.

Proposition A.1 *The following facts hold:*

1. $N \xrightarrow{\mathbf{nil}\ @\ l} N'$ *if and only if* $N \equiv N'' \parallel l :: \mathbf{nil}$*; moreover,* $N'' \equiv N' \equiv N$.
2. $N \xrightarrow{\langle l' \rangle\ @\ l} N'$ *if and only if* $N \equiv N'' \parallel l :: \langle l' \rangle$*; moreover,* $N' \equiv N'' \parallel l :: \mathbf{nil}$.
3. $N \xrightarrow{(\nu l')\ \langle l' \rangle\ @\ l} N'$ *if and only if* $N \equiv (\nu l')(N'' \parallel l :: \langle l' \rangle)$ *for* $l \neq l'$*; moreover,* $N' \equiv N'' \parallel l :: \mathbf{nil}$.

We need the notion of *bisimulation up-to structural congruence*: it is defined as a bisimulation except for the fact that the $\Re$ in the consequents of Definition 3.3 is replaced by the composed relation $\equiv \Re \equiv$.

Lemma A.2 *Let* $\approx_{\equiv}$ *be the largest bisimulation up-to structural congruence. Then* $\approx_{\equiv} \subseteq \approx$.

We can now characterize all the possible executions of the net $\mathcal{C}[N]$ in terms of the evolutions of N and $\mathcal{C}[\cdot]$ separately.

Lemma A.3 $\mathcal{C}[N] \xrightarrow{\alpha} \bar{N}$ *if and only if one of the following conditions hold:*

1. $N \xrightarrow{\alpha} N'$ *with* $n(\alpha) \cap bn(\mathcal{C}[\cdot]) = \emptyset$*, or*
2. $\mathcal{C}[\mathbf{0}] \xrightarrow{\alpha} \mathcal{C}'[\mathbf{0}]$*, or*
3. $N \xrightarrow{\alpha'} N'$ *with* $\alpha = (\nu l)\alpha'$*,* $\mathcal{C}[\cdot] \triangleq \mathcal{C}_1[(\nu l)\mathcal{C}_2[\cdot]]$ *and* $fn(\alpha) \cap bn(\mathcal{C}_1[\cdot], \mathcal{C}_2[\cdot]) = \emptyset$*, or*
4. $\mathcal{C}[\cdot] \triangleq \mathcal{C}_1[\mathcal{C}_2[\cdot] \parallel H]$ *with* $H \xrightarrow{\mathbf{nil}\ @\ l} H'$*,* $N \xrightarrow{(\nu \widetilde{l})C \triangleright l} N'$ *and* $l \notin bn(\mathcal{C}_2[\cdot])$*, or*
5. $\mathcal{C}[\cdot] \triangleq \mathcal{C}_1[\mathcal{C}_2[\cdot] \parallel H]$ *with* $H \xrightarrow{(\nu \widetilde{l})C \triangleright l} H'$*,* $N \xrightarrow{\mathbf{nil}\ @\ l} N'$ *and* $l \notin bn(\mathcal{C}_2[\cdot])$*, or*
6. $\mathcal{C}[\cdot] \triangleq \mathcal{C}_1[\mathcal{C}_2[\cdot] \parallel H]$ *with* $H \xrightarrow{(\nu \widetilde{l})\ \langle l_2 \rangle\ @\ l_1} H'$*,* $N \xrightarrow{l_2 \triangleleft l_1} N'$ *and* $\{l_1, l_2\} \cap bn(\mathcal{C}_2[\cdot]) = \emptyset$*, or*
7. $\mathcal{C}[\cdot] \triangleq \mathcal{C}_1[\mathcal{C}_2[\cdot] \parallel H]$ *with* $H \xrightarrow{l_2 \triangleleft l_1} H'$*,* $N \xrightarrow{(\nu \widetilde{l})\ \langle l_2 \rangle\ @\ l_1} N'$ *and* $l_1 \notin bn(\mathcal{C}_2[\cdot])$.

Moreover, the resulting net $\bar{N}$ *is, respectively, structurally equivalent to* $\mathcal{C}[N']$*, or* $\mathcal{C}'[N]$*, or* $\mathcal{C}_1[\mathcal{C}_2[N']]$*, or* $\mathcal{C}[(\nu \widetilde{l})(N' \parallel l :: C)]$*, or* $\mathcal{C}_1[\mathcal{C}_2[N] \parallel (\nu \widetilde{l})(H' \parallel l :: C)]$*, or* $\mathcal{C}_1[(\nu \widetilde{l})\mathcal{C}_2[N' \parallel H']]$ *(cases 6 and 7.). Finally,* $\alpha = \tau$ *in cases 4., 5., 6., and 7. .*

The last property is used to prove the following key result.

Lemma A.4 $\approx$ *is a congruence relation.*

In particular, the proof consists in showing that the relation $\{ (\mathcal{C}[N], \mathcal{C}[M]) : N \approx M \}$ is a bisimulation up-to $\equiv$. This result then easily implies that $\approx$ is transitive. Now it is easy to prove that $\approx$ satisfies the three requirements of Definition 3.2. By definition, this implies the following

Theorem A.5 (Soundness) $\approx \subseteq \cong$.

To prove the inverse inclusion, we start by giving simple equational laws for barbed congruence that can be easily proved sound by using $\approx$ as a proof-technique (this is enabled by the soundness theorem).

Proposition A.6 *The following facts hold:*

1. $(\nu l')(l :: \mathbf{in}(!x)@l'.P \parallel l' :: \langle d \rangle) \cong (\nu l')(l :: P[d/x])$
2. $l :: \mathbf{out}(l'')@l'.P \parallel l' :: \mathbf{nil} \cong l :: P \parallel l' :: \langle l'' \rangle$
3. $l :: \mathbf{eval}(Q)@l'.P \parallel l' :: \mathbf{nil} \cong l :: P \parallel l' :: Q$
4. $(\nu l)N \cong N$ *whenever* $l \notin fn(N)$

Then, we give an alternative (but easier to prove) characterization of requirement 3.2(3).

Lemma A.7 *Let* $N_1\ \Re\ N_2$*. Then* $\mathcal{C}[N_1]\ \Re\ \mathcal{C}[N_2]$ *for all context* $\mathcal{C}[\cdot]$ *if and only if*

1. $N_1 \parallel l :: P\ \Re\ N_2 \parallel l :: P$ *for any name* l *and process* P*, and*
2. $(\nu l)N_1\ \Re\ (\nu l)N_2$ *for any name* l

The lemma above simplifies the proof of the following key result. It states that a fresh locality (i.e. a locality not occurring in the involved nets) hosting a restricted datum can be removed together with the restriction from two barbed congruent nets without breaking the equivalence.

Lemma A.8 *If* $(\nu l)(N \parallel l_f :: \langle l \rangle) \cong (\nu l)(M \parallel l_f :: \langle l \rangle)$ *and* l_f *is fresh for* N *and* M*, then* $N \cong M$.

Given two barbed congruent nets N and M we are now ready to prove that, for any move of N (say $N \xrightarrow{\alpha} N'$), there exists a possible reply of (a net derived from) M satisfying Definition 3.3. The overall idea is to rely on the context closure and provide, according to α, a context forcing M to properly reply. The context is then erased by exploiting the previous lemma. This leads to the wanted

Theorem A.9 (Completeness) $\cong \subseteq \approx$.

$$
\begin{array}{rcl}
N_n^k & \approx_f & (\nu\widetilde{l})\big(\prod_{i=1}^{n} l_i :: \mathbf{in}(!z_1^i)@l_i.\ \ldots\ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l \mid \langle d_1\rangle \mid \ldots \mid \langle d_n\rangle\big) \\
& \approx_f & (\nu\widetilde{l})\big(\prod_{i=1}^{n} l_i :: \mathbf{out}(m'_i)@l \mid \langle d_{i_1}\rangle \mid \ldots \mid \langle d_{i_{k-1}}\rangle\big) \\
& \approx_f & (\nu\widetilde{l},\widetilde{l'})\big(\prod_{i=1}^{n} (\, l_i :: \mathbf{in}(m'_i)@l'_i.\mathbf{out}(m'_i)@l \mid \langle d_{i_1}\rangle \mid \ldots \mid \langle d_{i_{k-1}}\rangle \ \parallel\ l'_i :: \prod_{w\in\mathcal{O}} \langle w\rangle\,)\,\big) \\
& \approx_f & (\nu\widetilde{l},\widetilde{l'})\big(\prod_{i=1}^{n} (\, l_i :: \mathbf{in}(!z_1^i)@l_i.\ \ldots\ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{in}(m_i)@l'_i.\mathbf{out}(m_i)@l \mid \langle d_1\rangle \mid \ldots \mid \langle d_n\rangle \\
& & \qquad\qquad \parallel\ l'_i :: \prod_{w\in\mathcal{O}} \langle w\rangle\,)\,\big) \\
& \approx_f & M_n^k
\end{array}
$$

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 $\widetilde{z} \triangleq \{z_1,\ldots,z_{n-k+1}\}$.

$$
\begin{array}{rcl}
\hat{N}_n^k \parallel l :: \mathbf{nil} & \approx_f & (\nu\widetilde{l})\big(\prod_{i=1}^{n-h_1} l_i :: \mathbf{in}(!z_1^i)@l_i.\ \ldots\ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}()@l \mid \langle d_1^i\rangle \mid \ldots \mid \langle d_{n_i}^i\rangle\big) \parallel l :: \mathbf{nil} \\
& \approx_f & (\nu\widetilde{l})\big(\prod_{i=1}^{n-h_2} l_i :: \mathbf{out}()@l \mid \langle d_{i_1}\rangle \mid \ldots \mid \langle d_{i_{n_i-(n-k+1)}}\rangle\big) \parallel l :: \mathbf{nil} \\
& \approx_f & (\nu\widetilde{l})\big(\prod_{i=1}^{n-h} l_i :: \langle d_{i_1}\rangle \mid \ldots \mid \langle d_{i_{h_1-1}}\rangle\big) \parallel l :: \prod_{j=1}^{n-h} \langle\rangle \\
& \approx_f & l :: \prod_{j=1}^{n-h} \langle\rangle
\end{array}
$$

Table 4. Steps for Proving Equations (3) and (4)

B Proofs for the k-Set Agreement Example

In this section, we shall prove the properties formulated in Section 4. To this aim, we shall firstly give a proposition listing some basic features of $\approx_f$. In what follows, '$\langle d\rangle$ is not corruptible' means that $\langle d\rangle$ does never fail.

Proposition B.1

1. $(\nu l')(l :: \mathbf{out}(l'')@l'.P) \approx_f (\nu l')(l :: P \parallel l' :: \langle l''\rangle)$
2. $(\nu l')(\, l :: \mathbf{in}(!x)@l'.P \parallel l' :: \langle d\rangle\,) \approx_f (\nu l')(l :: P[d/x])$ *if* $\langle d\rangle$ *is not corruptible*
3. $(\nu l')(\, l :: \mathbf{in}(d)@l'.P \parallel l' :: \langle d\rangle\,) \approx_f (\nu l')(l :: P)$ *if* $\langle d\rangle$ *is not corruptible*
4. *If* P *fails before starting its computation, then* $l :: P \approx_f l :: \mathbf{nil}$

We are left to prove that in our framework Equations (3) and (4) hold. We start with Equation (3); the steps are detailed in the upper part of Table 4. We applied the following derivation:

- the first and the last steps have been inferred by applying several times Proposition B.1.1;
- the second and the forth steps have been inferred by applying several times Proposition B.1.2 (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 3.5.c and Proposition B.1.3. It is worth to notice that $m'_i \in \mathcal{O}$ because, since $|\mathcal{O}| = k$, at least one principal whose input value, say d', is in $\mathcal{O}$ 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 (4). This can be very similarly done by following the steps depicted in the lower part of Table 4. The first step is derived like the first step of the proof of Equation (3); moreover, by exploiting Proposition B.1.4, we let h_1 ($\leq h$) the number of principals failed before performing the **in** actions, and we let $d_1^i,\ldots,d_{n_i}^i$ (for $n-h_1 \leq n_i \leq n$) the data in l_i. The second step is derived like the second step of the proof of Equation (3); we let h_2 (for $h_1 \leq h_2 \leq h$) the number of principals failed when performing the **in** actions, and $d_{i_1},\ldots,d_{i_{n_i-(n-k+1)}}$ (for $d_{i_j} \in \{d_1^i,\ldots,d_{n_i}^i\}$) the data not withdrawn by the $n-k+1$ **in** actions of l_i. The third step is derived using Proposition B.1.1 (notice that $h-h_2$ principals must fail before performing the last **out** action), while the fourth step is derived using Proposition 3.5.c.