Building Reliable Distributed Infrastructures Revisited: a Case Study

Marc Lacoste
France Telecom R & D
marc.lacoste@rd.francetelecom.com

Abstract

Building reliable distributed infrastructures still remains a challenge. Two separate and slowly diverging visions for solutions have emerged: the first relies upon an implementation-based approach of today's now mature middleware object technologies, often without much concern for the underlying theoretical foundations. The second focuses on elaborate theoretical models of distributed computation, to capture various aspects of communication, mobility, or security, but has produced few implementations. We attempt to reconcile on a case study those two opposite views of reliable distributed computing using a simple methodology: define both a carefully designed distributed computing model with a well-defined semantics and a middleware implementation that strictly conforms to the model. We demonstrate the feasibility and the soundness of such a formal approach by describing the refinement steps in the formalization, from the model design to the implementation, using as an example a simple distributed programming model named DCP. *In* DCP, *remote interaction is primitive and the distribution of resources is explicit while keeping network communications transparent to the programmer. The* DCP *implementation is formalized by a distributed abstract machine, and seamlessly integrated with a middleware representative of current distributed technology – the* JONATHAN *Object Request Broker – to support network transparent communications.*

1. Introduction

With the expansion both of the Internet and of Web technologies, the needs for reliable middleware have become increasingly stronger with growing application complexity. A traditional approach to enhance reliability has been to use formal methods during infrastructure design and development, which allows to prove safety or liveness properties, such as deadlock freedom or fairness of execution, to reason about behavioral equivalence, and more generally to obtain provably correct infrastructures with respect to carefully designed models, which paves the way for clean infrastructure design, and validation of the overall system. Unfortunately, a widening gap has appeared between two views of distributed computing it is urgent to reconcile. On one end are today's now mature distributed object technologies such as Java, EJB, and CORBA which do not lend themselves easily to formalism, and where the implementation know-how is mostly detained by practitioners. On the other side, one finds ever more accurate theoretical models of distributed computation like process algebras, which aim to capture various aspects of communication, mobility, failures or security. However, the corresponding implementations are few – often small prototypes written in functional languages – and have become estranged from the mainstream object technology.

In this paper, we attempt to bring together those two facets on a case study by using a simple methodology to build infrastructures for distributed computing: (i) define a core programming model in the form of a distributed process calculus; (ii) specify and implement a virtual machine realizing the model, relying on a middleware reflecting current distributed system technology. The advantages of that approach are twofold: (i) these particularly well-studied models allow easy formal proofs of properties about the system; (ii) the prototype will truly reflect today's widespread middle-

ware implementation techniques. We test the soundness of that approach using as a starting point a high-level language for concurrent programming called PICT [16], whose theoretical foundations are remarkably well-defined: built on top of a minimal calculus named CORE PICT, it can be totally implemented in terms of π-calculus [15] primitives; a concurrent implementation was even formalized in the form of an abstract machine (AM) [20] and proved correct with respect to π-calculus. PICT, however does not address distribution issues. We go beyond PICT by using a specific notion of *domain* [8] to capture partitions commonly found in everyday distributed systems programming, as in administrative domains which constitute the structure of a wide area network, or trust domains where member entities have a high-level of confidence in one another.

The goal of this work is to illustrate on a distributed version of PICT the soundness of the approach, by showing the refinements in the formalization steps, ranging from the design of a distributed programming model taking a minimalist view of domains, to its middleware implementation.

More specifically, we introduce a simple model for distributed programming, called DCP (DISTRIBUTED CORE PICT) which extends the CORE PICT calculus to include a simple form of domain and remote interaction primitives. The model makes the distribution of resources explicit, while keeping network communications transparent to the programmer.

The main contribution of this paper is to demonstrate the soundness of the above approach for building reliable distributed infrastructures using a proof of concept argument, by formalizing an implementation of the model in the form of a distributed AM, and by implementing that formal specification by a distributed runtime written in Java. The runtime is seamlessly integrated with a middleware representative of current distributed system technology, namely the JONATHAN [5] ORB (Object Request Broker) framework to provide support for network transparent communications.

The remainder of the paper is organized as follows. Section 2 shortly discusses the notion of domain. Section 3 reviews related work. The DCP programming model, the formal specification, and its distributed implementation are then presented in sections 4, 5 and 6. Formal proofs are reported in appendix.

2. Domains

The DCP programming model is structured around the fundamental concept of *domain*. To understand its significance for distributed computing, it is worth recalling that in recent years, distributed programming techniques such as RPC (Remote Procedure Call), RMI (Remote Method Invocation), or ORBs have always aimed to abstract away and hide the complexity of network topology to ease the task of the programmer, which then views all remote resources as "one logical step apart" [3]. If a complete hiding of network structure is still possible in local area networks, such an approximation no longer holds for wide area networks like the Web, where the need for a notion of *location* is keenly felt. Why? Insightful answers proposed by L. Cardelli shed a new light on the problem: unlike in a LAN-based setting, a WAN cannot be subject to centralized control, and takes the form of a completely decentralized superposition of administrative domains, separated by strong trust barriers for security purposes [3]. However, these domains interact through communication and mobility primitives, which in turn require the crossing of barriers. The key problem of WAN-based programming then resides in when to authorize such crossings without compromising security.

Thus, we obtain a vision of a WAN as being structured by barriers delimiting a number of administrative domains, and where atomicity for fully transparent communication or migration is lost, taking instead the form of a number of barrier crossing steps. Such a vision is extended in [8] to any large-scale distributed system which "should primarily be understood as a partitioned system. Sub-systems and components of a distributed system can be grouped in different, possibly overlapping sets, generally under the control of a single object or entity. [...] We shall call *domains* such sets of sub-systems or components". The distinguishing domain feature is being organized around some behavior, common to member entities, such as an access control policy, a name management mechanism, or a failure mode.

Domains borders may capture physical barriers, as between two physically distant machines, where com-

munication delays cannot be abstracted away due to an observable absolute lower bound for propagation. For the most part, domains commonly found in distributed mobile programming correspond to the "virtual locations" of [3], which, besides administrative domains bounded by firewalls where access to resources is controlled with respect to a common security policy, may include trust domains[1] where mutually suspicious participants in a collaborative computation have a high-level of confidence in domain members only, but distrust outside participants. Other notions of domain concern failures and latency. For instance, a location can be viewed as a unit of failure [1, 7], all processes or locations contained within failing together according to the same modes[2].

For the purpose of this paper, which is to illustrate on a simple model of distributed computation the methodology described in the introduction, we use a simpler form of domain, leaving richer forms of control for future study. Thus, in accordance with the vision of WANs described previously, the domains used in the DCP model will be limited to making physical distribution explicit, and can represent subnets, network nodes or UNIX processes. We adopt a flat domain topology.

3. Related Work

Technical Context The theory of concurrency, beginning with the asynchronous π-calculus, provided a very rich framework as a starting point for DCP: a number of distributed process calculi such as NOMADIC PICT [19, 21], DITYCO [12, 13], the JOIN-calculus [6, 7], or the AMBIENT calculus [4] introduce constructs similar to domains in terms of sites, systems, locations, networks or ambients.

NOMADIC PICT extends PICT to build infrastructures for mobile agents. Two types of agent communication facilities are provided: low-level primitives require the programmer to be aware of the site hosting the agent; high-level primitives are location-independent and can be expressed in terms of the low-level ones. The *sites* of NOMADIC PICT are similar to the domains of DCP: both have a flat structure. NOMADIC PICT adds the notion of *agent*, hosted by a site, which behaves as a migration unit. Interaction is only possible between processes inside the same agent, or between agents on the same site.

TYCO [22] aims to formalize asynchronous interactions between concurrent objects. Its main abstractions are *messages*, *objects* and *template declarations* to represent classes. In DITYCO (Distributed TYCO), a *site* hosts a number of processes, and a *network* represents a flat organization of sites. Remote communication follows a "move and communicate" (M&C) semantics [3]. An implementation of TYCO was formalized by an AM [14].

Born from the difficulty to implement π-calculus in an asynchronous, distributed setting, the JOIN-calculus is another aim at expressiveness for the foundations of WAN-based programming: as π-calculus channels are not attached to a particular location, one may quickly face the need for a distributed consensus for each communication. The JOIN-calculus solution is to force a given receiver to be defined in a unique location. The basic abstractions are *processes* which communicate according to rules specified in *definitions*, and *locations* which behave as failure and migration units. A special construct called the *join-pattern* provides message synchronisation. JOIN-calculus locations are organised in a tree structure, and can be created dynamically. The unicity of receivers constraint makes implicit routing possible, since the target receiver of a message is known unambiguously.

Contrary to the previous calculi, all derived from the asynchronous π-calculus which focuses on communication using channels, the AMBIENT calculus considers mobility as a primitive concept: all computations

[1] Webs of trust are commonly used in public-key infrastructures, either using hierarchical X500/X509-type models, or more general trust-graphs like in the introducer-based solution of PGP (Pretty Good Privacy [23]). In a trust-graph, each node represents a principal. Trust relations are captured by *trust-paths* that connect the various nodes: along a trust-path, each node has direct confidence in the authenticity of the public key of the next node.

[2] In a WAN, due to asynchrony, failures are indistinguishable from long response times. Thus, unpredictable bandwidth fluctuations and network congestion may also cause additional partitions of the network into subnets, where communication links have similar latencies.

[3] In M&C, remote interaction is viewed as a high-level construct, transparent to the location of the receiver, that can be encoded with low-level primitives of process migration and local communication. Thus, true communication is always purely local.

can be expressed in terms of migration. The aim is to provide a formal foundation for mobile computing in WANs, described as hierarchically partitioned into administrative domains. Mobility is seen as crossing domain boundaries, and security is captured by authorizations to move between domains. Domains are represented by the *ambient* abstraction, or named unit of migration containing *processes* and other ambients. Mobility operations – entering, leaving an ambient or dissolving an ambient boundary – are controlled using *capabilities*. Communications are limited to purely local asynchronous interactions. SAFE AMBIENTS [11] extends the calculus using coactions to avoid the occurrence of some forms of interference considered as programming errors.

Design Principles Demonstrating the feasibility of the overall "design-to-implementation" formal approach meant formalizing an implementation of a simple model of distributed computation, and showing the middleware implementation truly followed the formal specification, itself a refinement of the model. Thus, requirements for the model were simplicity, while still closely reflecting existing infrastructures for distributed computing. With that perspective in mind, we made the following design decisions for the DCP model:

- *Explicit Distribution of Resources.* As shown in section 2, the notion of domain is particularly relevant to capture partititions occurring in distributed systems. Thus, it was chosen as the underlying basis for the model, allowing it to reflect many important features of distributed mobile programming with a single concept. However, to keep the model simple, domains were limited to making physical distribution explicit, without any additional behavior.

- *Primitive Remote Interaction.* The existing distributed calculi always strike a delicate balance between primitives for communication and migration control. We chose to investigate a fully communication-oriented approach[4]. Therefore, the DCP model appears as a direct extension of the asynchronous π-calculus. Including remote message send constructs as core features, rather than encoding them with process migration facilities like the M&C semantics found in NOMADIC PICT, DITYCO, and AMBIENTS also seemed more realistic in terms of implementation costs. DCP is closer to the JOIN-calculus where remote interaction is a primitive distinct from process migration.

- *Network Communications Transparency.* DCP uses network transparent names which make implicit routing possible, thus hiding the network complexity to the programmer as in the JOIN-calculus.

Formalizing the Implementation The DCP AM extends that proposed by Turner for PICT [20] by including remote interaction primitives. It is one of the first attempts to formalize a distributed implementation of a process calculus by a distributed AM. The approach of DITYCO is similar, but the AM which supports the execution of multiple threads is only presented in a non-distributed setting [14]. In NOMADIC PICT, a distributed infrastructure is seen as a formal translation from a high-level language to a low-level calculus. The work closest to ours is [18] where a Java RMI-based implementation of SAFE AMBIENTS is formalized by a distributed AM named PAN which presents two important differences with DCP: first, the underlying model uses mobility as a guiding principle, while DCP is communications-oriented. Second, DCP uses a fully-fledged CORBA-compliant ORB, which supports interoperability with other applications, possibly not coming from the Java world.

To our knowledge, the DCP approach to integrate a distributed AM *à la* PICT with an ORB, focusing on remote communications, is novel. It allows us to combine the strength of functional programming techniques, traditionnally used to implement efficiently process calculi, with the object-oriented communication framework provided by a CORBA-type middleware. The main benefit is to obtain within a single platform, both a very efficient implementation of the sequential core of the language VM, and a communication layer allowing easy interoperability with existing middleware, and other widely spread distributed object technologies. In addition, the compliance of the

[4] This choice brings the benefit of possibly modelling migration by including higher-order features in the model, i.e., processes can be transmitted over communication channels [17].

VM with the formal model of distributed computation, ensures guarantees of reliability for the platform.

4 The DCP Model

DCP represents a distributed system with a limited number of abstractions: *configurations*, *domains*, *processes*, and *channels*. The syntax is the following:

$$\begin{array}{lll} C & ::= & \mathsf{nil} \mid (\nu\ n)\ C \mid M \mid a[P] \mid C \parallel C \\ P & ::= & P \mid P \mid (\nu\ n)\ P \mid \mathsf{0} \mid x?(\tilde{y}).P \mid x!\langle\tilde{y}\rangle \mid \\ & & *P \mid [u\ =\ v]\ P\ ,\ Q \\ M & ::= & \langle x, \tilde{y}\rangle \end{array}$$

A *configuration* $C, D \ldots$ is a parallel composition of domains $a[P]$, and of messages M [5] being routed between domains. The inert configuration nil represents an idle system. A *domain* brings together inside a single scope basic computational units called *processes*. We denote $a[P \mid Q \mid \ldots]$ the domain a hosting several processes P,Q,... running in parallel. Two separate parallel composition operators are used, $\parallel$ for configurations, and $\mid$ for processes running within a domain scope.

Inter-process communication, both within and between domains, is based on asynchronous message passing on *channel* names. Those first class values $m, n, x, y \ldots$ may be transmitted on other channels causing the communication topology to evolve dynamically. The calculus syntax features a single notion of name that encompasses local names, valid only inside a single domain, and network names, valid throughout the distributed system.

Process-related operations are closely inspired from CORE PICT, and include standard π-calculus constructs such as the inert process $\mathsf{0}$, conditional testing for equality of values, parallel composition, channel name creation $(\nu\ n)\ P$ – the scope of name n being restricted to P – or replication $*P$ which can be thought of as an infinite number of copies of P running in parallel. Finally, two processes communicate on some channel x using the $x!\langle\tilde{y}\rangle$ and $x?(\tilde{y}).P$ primitives for an output and input of a tuple of values $\tilde{y}$, the execution resuming as P after reception.

[5] For a message $\langle x, \tilde{y}\rangle$, we denote x the target receiver, and $\tilde{y}$ the arguments.

The operational semantics is based on a structural equivalence $\equiv$ and a reduction relation $\longrightarrow$. The definition of $\equiv$ is standard: it is the smallest equivalence relation, closed by contexts $\mathbf{E}(\cdot)$, $*(\cdot)$, $a[\cdot]$, and $x?(\tilde{y}).\cdot$ on processes, by contexts $\mathbf{F}(\cdot)$ on configurations, and such that[6]:

$$\begin{array}{rcll} *P & \equiv & P \mid *P & \\ a[(\nu\ n)\ P] & \equiv & (\nu\ n)\ a[P] & \\ (\nu\ n)\ P \mid Q & \equiv & (\nu\ n)\ (P \mid Q) & n \notin \mathsf{fn}(Q) \\ (\nu\ n)\ C \parallel D & \equiv & (\nu\ n)\ (C \parallel D) & n \notin \mathsf{fn}(D) \end{array}$$

where evaluation contexts $\mathbf{E}(\cdot)$ and $\mathbf{F}(\cdot)$ are given by:

$$\begin{array}{lll} \mathbf{E}(\cdot) & ::= & \cdot \mid (\nu\ n)\ \mathbf{E}(\cdot) \mid \mathbf{E}(\cdot) \mid P \\ \mathbf{F}(\cdot) & ::= & \cdot \mid (\nu\ n)\ \mathbf{F}(\cdot) \mid \mathbf{F}(\cdot) \parallel C \end{array}$$

In terms of computation, $\longrightarrow$ is the smallest relation compatible with $\equiv$, and closed by contexts $\mathbf{E}(\cdot)$ and $a[\cdot]$ for processes and by contexts $\mathbf{F}(\cdot)$ for configurations, and such that:

$$\frac{}{[u\ =\ u]\ P\ ,\ Q \longrightarrow P} \quad \textbf{(if-true)}$$

$$\frac{u \neq v}{[u\ =\ v]\ P\ ,\ Q \longrightarrow Q} \quad \textbf{(if-false)}$$

$$\frac{}{m?(\tilde{u}).P \mid m!\langle\tilde{v}\rangle \longrightarrow \{\tilde{v}/\tilde{u}\}P} \quad \textbf{(com)}$$

$$\frac{m \notin \mathsf{dn}(P)}{a[P \mid m!\langle\tilde{v}\rangle] \longrightarrow a[P] \parallel \langle m, \tilde{v}\rangle} \quad \textbf{(send)}$$

$$\frac{m \in \mathsf{dn}(P)}{\langle m, \tilde{v}\rangle \parallel a[P] \longrightarrow a[P \mid m!\langle\tilde{v}\rangle]} \quad \textbf{(receive)}$$

Intra-domain communications are performed as in asynchronous π-calculus using the **(com)** rule: the interaction on some channel m of an input and an output process reduces in the continuation of the input process where the formal parameters are replaced by the values having been sent.

Inter-domain communications are captured by rules **(send)** and **(receive)**: if the target receiver m is

[6] $\mid$ and $\parallel$ also satisfy the monoid laws: $P \mid \mathsf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, $C \parallel \mathsf{nil} \equiv C$, $C \parallel D \equiv D \parallel C$, and $C \parallel (D \parallel E) \equiv (C \parallel D) \parallel E$.

located in a different domain, i.e., m is not contained in the set of free receivers $\mathsf{dn}(P)$ found in current domain $a[P \mid \ldots]$, the output process is removed from a, and a message is sent on the network targeted at the remote domain. That message will be consumed when it reaches the domain containing the target receiver. An output process is then created inside the scope of that domain, and the interaction terminates by a local communication.

To guarantee that DCP processes are correctly written, we use a well-formedness inference system WF adapted from the receptive π-calculus $\mathrm{D}\pi_1^r$ [2]. We define environments, ranged over by $\Gamma, \Delta, \ldots$, by the grammar $\Gamma ::= \emptyset \mid \Gamma, x$. Informally, an environment is a finite set of channel names representing the free receivers contained in a process (or configuration). Thus, it captures the interface of that process (or configuration), i.e., the set of communication ports it can use to interact with the outside world. Let Γ be an environment. The judgments of WF are of the form $\Gamma \vdash P$ or $\Gamma \vdash C$ (read as "DCP process P or configuration C are well-formed in environment Γ"). The WF inference rules are given in figure 1. A well-formed DCP process P satisfies the two following properties:

(LOC) P cannot receive messages on a name it just received from another channel.

(UR) P and another process Q may receive on the same name m, if they are located inside the same domain but not if they belong to different domains.

(LOC) means that a received name cannot be used to perform inputs in the continuation of P, e.g., process $x?(y).y?(z).0$ is ill-formed. It stems from a consideration of locality: a process should only be allowed to listen for messages on local receivers. (UR) is a variant of the constraint of the unicity of receivers, advocated by several distributed programming models [6, 9] to be required for a model of distributed computation to be implementable. Thus, only a local consensus is needed instead of a distributed one to determine which receiver should process the communication. In DCP, this constraint is imposed for receivers belonging to different configurations ($\Gamma \cap \Delta = \emptyset$ condition in rule [T-PAR-C]), but not inside a given domain to be compatible with the PICT semantics where several input processes are allowed to listen on the same channel ($\Gamma \cap \Delta \neq \emptyset$ is possible in rule [T-PAR-P]).

Well-formedness of process or configuration terms is preserved by reduction:

Theorem 1 (Subject Reduction) *1. If $\Gamma \vdash P$ and $P \longrightarrow Q$. Then, there is Δ such that $\Delta \vdash Q$.*

2. If $\Gamma \vdash C$ and $C \longrightarrow D$. Then, there is Δ such that $\Delta \vdash D$.

5 A Formal Specification

We now formally specify an implementation of DCP by a distributed AM. We only discuss the machine informally. The main formal definitions of the AM are collected in figure 2. In the sequel, we use the notations $A \mapsto B$ for the set of finite maps from set A to set B, A^* for the set of finite collections of elements of set A, $a_1 :: \ldots :: a_n$ for an element Q from the set $Queue(A)$ of queues of elements of set A, $|Q|$ for the number of elements in Q, and $\bullet$ for the empty queue.

A general DCP configuration is implemented by a *network*, which is a parallel composition of AM and of messages, each domain $a[P]$ being implemented by one AM. The state of the AM is a data structure $S = (a, \mathcal{U}, \mathcal{V}, \mathcal{H}, \mathcal{R})$. The *machine name* a designates the AM in a unique way. $\mathcal{U}$ and $\mathcal{V}$ are two sets of names that represent the *local and remote channel names* known by the machine during its execution. The *heap* $\mathcal{H}$ behaves as a communication buffer. It maps channel names x to FIFO channel queues $\mathcal{H}(x)$ containing input rd or output processes wr blocked on a communication on channel x. We have $x \in \mathcal{U}$ for a local communication, and $x \in \mathcal{V}$ for a remote one. The *run queue* $\mathcal{R}$ contains processes ready for execution, the one at the head of the queue being run first. Newly created processes are generally placed at the end of the run queue, to guarantee a fairness of execution property.

To manage distribution, the single notion of name found in the calculus needs to be refined. We introduce the notion of *pickled name* to represent a network-level name. It is an abstraction both of an on-the-wire representation of a name viewed as a communication capability, and of the underlying communication resources necessary for a remote interaction to occur. Formally, we assume given two name transformation operations pickle and unpickle. $\mathsf{pickle}(a, x)$ returns a pickled name corresponding to local name x created

$$\emptyset \vdash 0 \quad \text{[T-NIL-P]} \qquad \emptyset \vdash x!\langle \tilde{y} \rangle \quad \text{[T-SND]} \qquad \emptyset \vdash \mathsf{nil} \quad \text{[T-NIL-C]} \qquad \emptyset \vdash M \quad \text{[T-MSG]}$$

$$\frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma \cup \Delta \vdash P \mid Q} \quad \text{[T-PAR-P]} \qquad \frac{\Gamma, n \vdash P \quad n \notin \Gamma}{\Gamma \vdash (\nu\ n)\ P} \quad \text{[T-RES-P]} \qquad \frac{\Gamma \vdash P \quad \tilde{y} \notin \Gamma}{\Gamma, x \vdash x?(\tilde{y}).P} \quad \text{[T-RCV]}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash a[P]} \quad \text{[T-DOM]} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash *P} \quad \text{[T-REPL]} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash [u\ =\ v]\ P\ ,\ Q} \quad \text{[T-IF]}$$

$$\frac{\Gamma \vdash C \quad \Delta \vdash D \quad \Gamma \cap \Delta = \emptyset}{\Gamma \cup \Delta \vdash C \parallel D} \quad \text{[T-PAR-C]} \qquad \frac{\Gamma, n \vdash C \quad n \notin \Gamma}{\Gamma \vdash (\nu\ n)\ C} \quad \text{[T-RES-C]}$$

Figure 1. Well-Formed DCP Terms

in domain a. Conversely, $\mathsf{unpickle}(x)$ returns a pair (a, y): pickled name x is resolved into local name y in the context of domain a .

For fairness purposes, a parallel composition of processes is addressed by running one of them immediately, while the other is scheduled for later execution ([PRL] rule). When a new channel name n is created ([RES] rule), all occurrences of n are replaced by a "fresh" name u, a new entry for u is allocated in the heap, and the set of known local names is updated.

Local interactions are performed by the [REDIN], [REDOUT], [REDREPL], and [REDOUT*] rules. For instance, when the redex is composed of an input process in the run queue and of an output process in the heap, execution resumes with the continuation of the input process ([REDIN] rule). The other rules are similar, mainly dealing with replicated processes. A run queue process which cannot be reduced immediately is transferred into the channel queues of the heap by rules [QUEUEOUT], [QUEUEIN], and [QUEUEREPL]. It will eventually leave the heap when local interaction becomes possible – an input/output process appears in the run queue – or when the remote emission rule [REMXMIT] is triggered.

Remote interaction is captured by the [REMXMIT] and [REMRCV] rules, which implement scope extrusion between domains of calculus-level channel names using the pickle/unpickle operations. If the target receiver x is remote, the repeated application of the [REMXMIT] rule will empty the communication buffer $\mathcal{H}(x)$ by sending messages over the network. If the message arguments are locally created names, they must first be pickled to be understood by the target AM. Otherwise, they are already in a pickled form and may be transmitted "as is".

When receiving a message $\langle x, \tilde{y} \rangle$ ([REMRCV] rule), an AM a checks that the target receiver x belongs to the set of known local names, i.e., $\mathsf{unpickle}(x) = (a, x') \quad x' \in \mathcal{U}$. The message arguments are then resolved locally. If resolution does not yield a local name, the set of known remote names is updated. An output process emitting on the local version x' of x is then included at the end of the run queue. It will eventually reduce when a local interaction rule is triggered.

We deem the AM to implement correctly the calculus:

Conjecture 1 (Soundness) *There exists an encoding $[\![\cdot]\!]$ from network states to* DCP *configuration terms, and a notion of equivalence $\sim$ on* DCP *configuration terms such that: $\forall N \in \mathsf{Network},\ N \longrightarrow N' \Longrightarrow [\![N]\!] \xrightarrow{*} [\![N']\!]$, up to $\sim$.*

The AM also satisfies a fairness of execution property. However, the property "a process P located in the AM run-queue will be run after a finite number of reductions" does not hold as an AM a can always fire the [REMRCV] rule, provided there is an infinite number of incoming messages, e.g., if a remote AM b keeps firing messages targeted at a, thus preventing P from making any progress in the run queue. Thus, we must refine on the type of reductions steps. For an AM state S, $S \longrightarrow S'$ is a *computation step* if any reduction rule except [REMRCV] is applied and a *listening step* otherwise. For $n \geq 1, 0 < k \leq n$, a derivation is said to be *(n,k)-fair* iff out of n reduction steps, at least k are computation steps. A derivation is said to be *fair*, iff there is $n \geq 1, 0 < k \leq n$, such that the derivation is *(n,k)-fair*.

Definition 1 (Well-formedness) *A machine state $S = (a, \mathcal{U}, \mathcal{V}, \mathcal{H}, \mathcal{R})$ is* well-formed *if the channels of*

$\mathcal{H}$ do not contain a mixture of input and output processes, and $\mathcal{R}$ contains only well-formed processes, replication being limited to replicated input, and output (resp. input) only occuring on channels from $\mathcal{U} \cup \mathcal{V}$ (resp. $\mathcal{U}$).

Theorem 2 (Fairness) *For an AM only performing fair derivations from a well-formed state, a process located in the run-queue will be run after a finite number of reductions.*

6 A Distributed Implementation

We now describe the implementation of the DCP distributed AM on top of a typical middleware platform. It appears necessary to refine further the formal specification in order to map abstractions issued from the theory of concurrency and of distributed computation like channel names, processes, and domains to notions more commonly found in middleware implementations such as identifiers, references, naming contexts, or protocol and session objects. The mapping is presented using the example of the JONATHAN ORB as support for network communication transparency.

From an implementation perspective, the $\|$ operator of the model may naturally be viewed as capturing physical distribution of domains. Thus, the distributed runtime implementing the DCP model takes the form of a number of interpreters running concurrently on possibly different sites connected by a communication infrastructure. Each interpreter, written in Java, executes the process terms found inside the corresponding DCP domain and is composed of two elements as shown in figure 3. An *execution engine* implements the local reduction rules of the specification. A *communication manager* (CM) handles the remote interaction primitives. The input language allows the programmer to specify a number of well-known names in order for the interpreters to discover one another at system bootstrap.

The Execution Engine A special class called `DCPMachine` encapsulates the state of the interpreter, i.e., the structures $\mathcal{U}$, $\mathcal{V}$, $\mathcal{H}$, and $\mathcal{R}$ found in the formal specification, and which are initialized during the parsing phase. For debugging purposes, a graphical frontend allows the programmer to directly visualize the state of the interpreter, to navigate through the history of past reductions, or to execute a DCP program in a step-by-step fashion.

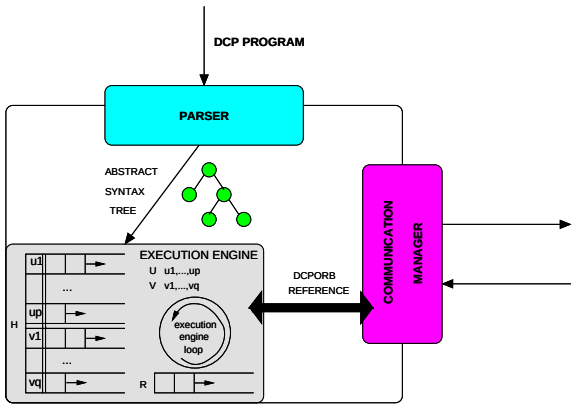

Figure 3. Structure of an Interpreter

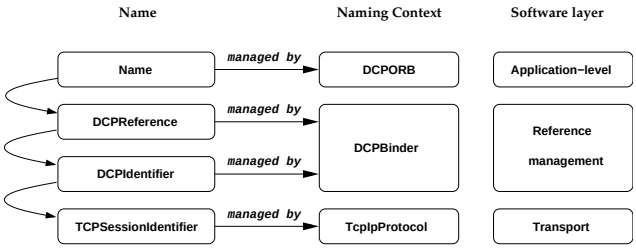

Figure 4. CM Architecture

Each local AM reduction rule is implemented by the body of a *rule method* that faithfully follows the specification. A rule method returns `true` whenever the conditions to fire the rule are met. Scheduling to determine the next applicable rule is done using a simple loop, calling in turn each rule method. Other scheduling policies are easily implemented by overloading the `DCPMachine.run()` method.

The Communication Manager To implement the network transparency of the DCP calculus, we use a minimal framework named JONATHAN [5] that provides support for building highly-flexible ORBs where objects interact transparently through remote method invocation on well-defined interfaces. The originality of JONATHAN compared to traditional object platforms such as CORBA or Java RMI is to provide a core framework for building different types of middleware called *personalities*, based on the notions of *bind-*

$m, n, u, v \ldots \in \mathsf{Name}$ Channel name $\quad a, b, \ldots \in \mathsf{DName}$ Machine name

$P, Q, \ldots \in \mathsf{Pr}$ Process $\quad \aleph \in \mathsf{Msg} = \mathsf{Name} \times \mathsf{Name}^*$ Message

$$\begin{array}{lll} \mathsf{pickle} & : & \mathsf{DName} \times \mathsf{Name} \longrightarrow \mathsf{Name} \quad \mathsf{unpickle}(\mathsf{pickle}(a,x)) = (a,x) \\ \mathsf{unpickle} & : & \mathsf{Name} \longrightarrow \mathsf{DName} \times \mathsf{Name} \quad \mathsf{pickle}(\mathsf{unpickle}(x)) = x \end{array}$$

$\mathcal{U}$	$\in$	$\mathsf{LNames} = \mathsf{Name}^*$	Local names
$\mathcal{V}$	$\in$	$\mathsf{RNames} = \mathsf{Name}^*$	Remote names
$\mathcal{H}$	$\in$	$\mathsf{Heap} = \mathsf{Name} \mapsto \mathsf{Queue}(\mathsf{Pr})$	Heap
$\mathcal{R}$	$\in$	$\mathsf{RQueue} = \mathsf{Queue}(\mathsf{Pr})$	Run queue
S	$\in$	$\mathsf{AM} = \mathsf{DName} \times \mathsf{LNames} \times \mathsf{RNames} \times \mathsf{Heap} \times \mathsf{RQueue}$	AM state
N	$\in$	$\mathsf{Network} = \mathsf{AM}^* \times \mathsf{Msg}^*$	Networks

[PRL] $$a,\mathcal{U},\mathcal{V},\mathcal{H},P \mid Q :: \mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H},P :: \mathcal{R} :: Q$$

[RES] $$\frac{u \notin \mathcal{U} \cup \mathcal{V} \quad \mathcal{U}' = \mathcal{U} \cup \{u\}}{a,\mathcal{U},\mathcal{V},\mathcal{H},(\nu\ n)\ P :: \mathcal{R} \longrightarrow a,\mathcal{U}',\mathcal{V},\mathcal{H}\ \{u \mapsto \bullet\},\{u/n\}P :: \mathcal{R}}$$

[REDIN] $$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = x!\langle\tilde{y}\rangle :: wr \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a,\mathcal{U},\mathcal{V},\mathcal{H},x?(\tilde{z}).P :: \mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H}\ \{x \mapsto wr\},\{\tilde{y}/\tilde{z}\}P :: \mathcal{R}}$$

[REDOUT] $$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = x?(\tilde{z}).Q :: rd \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a,\mathcal{U},\mathcal{V},\mathcal{H},x!\langle\tilde{y}\rangle :: \mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H}\ \{x \mapsto rd\},\mathcal{R} :: \{\tilde{y}/\tilde{z}\}\ Q}$$

[REDREPL] $$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = x!\langle\tilde{y}\rangle :: wr \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a,\mathcal{U},\mathcal{V},\mathcal{H},*x?(\tilde{z}).P :: \mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H}\ \{x \mapsto wr\},*x?(\tilde{z}).P :: \mathcal{R} :: \{\tilde{y}/\tilde{z}\}\ P}$$

[REDOUT*] $$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = *x?(\tilde{z}).Q :: rd \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a,\mathcal{U},\mathcal{V},\mathcal{H},x!\langle\tilde{y}\rangle :: \mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H}\ \{x \mapsto rd :: *x?(\tilde{z}).Q\},\mathcal{R} :: \{\tilde{y}/\tilde{z}\}\ Q}$$

[QUEUEIN] $$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = rd}{a,\mathcal{U},\mathcal{V},\mathcal{H},x?(\tilde{z}).P :: \mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H}\ \{x \mapsto rd :: x?(\tilde{z}).P\},\mathcal{R}}$$

[QUEUEOUT] $$\frac{x \in \mathcal{U} \cup \mathcal{V} \quad \mathcal{H}(x) = wr \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a,\mathcal{U},\mathcal{V},\mathcal{H},x!\langle\tilde{y}\rangle :: \mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H}\ \{x \mapsto wr :: x!\langle\tilde{y}\rangle\},\mathcal{R}}$$

[QUEUEREPL] $$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = rd}{a,\mathcal{U},\mathcal{V},\mathcal{H},*x?(\tilde{z}).P :: \mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H}\ \{x \mapsto rd :: *x?(\tilde{z}).P\},\mathcal{R}}$$

[REMXMIT] $$\frac{x \in \mathcal{V} \quad \mathcal{H}(x) = x!\langle y_1,\ldots,y_s\rangle :: wr \qquad (z_i)_{1\le i\le s} = (\mathsf{pickle}(a,y_i) \text{ if } y_i \in \mathcal{U}\ ;\ y_i \text{ if } y_i \in \mathcal{V})_{1\le i\le s}}{a,\mathcal{U},\mathcal{V},\mathcal{H},\mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V},\mathcal{H}\ \{x \mapsto wr\},\mathcal{R}\ \parallel\ \langle x,\tilde{z}\rangle}$$

[REMRCV] $$\frac{\mathsf{unpickle}(x) = (a,x') \quad x' \in \mathcal{U} \quad \mathcal{V}' = \mathcal{V} \cup (T_i)_{1\le i\le s} \qquad (z_i,T_i)_{1\le i\le s} = \left(\begin{array}{ll} (y_i',\emptyset) & \text{if} \quad \mathsf{unpickle}(y_i) = (a,y_i'), y_i' \in \mathcal{U} \\ (y_i,\{y_i\}) & \text{if} \quad \mathsf{unpickle}(y_i) = (b,\_), b \neq a \end{array}\right)_{1\le i\le s}}{\langle x,y_1,\ldots,y_s\rangle\ \parallel\ a,\mathcal{U},\mathcal{V},\mathcal{H},\mathcal{R} \longrightarrow a,\mathcal{U},\mathcal{V}',\mathcal{H}\ \{v \mapsto \bullet\}_{v\in\mathcal{V}'\setminus\mathcal{V}},\mathcal{R} :: x'!\langle\tilde{z}\rangle}$$

Figure 2. The DCP Abstract Machine: Main Definitions and Reduction Rules

ing[7] and of *flexible communications*[8]. That flexibility allowed us to implement easily the CM by developing a new personality directly above the TCP/IP abstractions provided by the JONATHAN distribution, so as to keep communications lightweight.

JONATHAN is built around a limited number of abstractions:

- An *identifier* is a generic notion of name that uniquely designates an object in a given naming context. Identifier semantics are naming context-specific: distributed, persistent... More generally, a *reference* encapsulates a number of identifiers.
- A *naming context* provides name creation and management facilities. It guarantees that each of the names it controls unambiguously designates some object.
- A *binder* is a naming context that, for a given managed name, is able to create an access path towards the object designated by that name.

These definitions offer a generic and uniform view of bindings, and clearly separate object identification from object access:

- In a given naming context `nc`, a new name for an object `o` is obtained by invoking the `nc.export(o)` method. Reference chains can then be created by exporting that name to other naming contexts.
- The creation of an access path to object `o` designated by identifier `id` is performed by invoking the `id.bind()` method which returns a ready-to-use surrogate to communicate with `o`.

JONATHAN also provides a modular framework for communication protocols:

- A *protocol* object is a binder representing some network protocol. It ensures the management of session identifiers, and the creation of bindings towards session objects.
- A *session* is a logical communication channel dynamically created by a protocol object. It has higher and lower interfaces to send messages down and up a protocol stack, which may be viewed as a stack of sessions.
- A *session identifier* designates an exported session object. Its internal structure is protocol-specific. It is typically created on the server side when an object is exported to a protocol and may be used on the client side to establish a communication channel.

Creating Local Names The CM is organized into three name-management layers (figure 4). The top-level layer manipulates DCP-calculus names: `Name` objects are controlled by a `DCPORB` binder, which acts as the CM interface for other components of the interpreter. Lower-level layers include reference management, and transport over TCP/IP connections. The creation of a fresh DCP channel name is implemented by simply creating a new instance of special subclass of the `Name` class called `LocalName`.

Implementing Scope Extrusion The pickling operations found in the formal specification can be partly implemented using the `export()` method of the JONATHAN framework: a locally created name `n` is exported to the current instance `orb` of the `DCPORB` binder by the call `orb.export(n);` which returns a `RemoteName` surrogate representing the pickled version of `n`. It also exports `n` recursively to lower-level name-management layers. Thus, a TCP/IP server session will be created to listen for communications on `n`: any incoming message from the network will be relayed upwards in the stack of server-side sessions to the `n` object for processing.

At system bootstrap, interpreter discovery is achieved by using a naming service: after the parsing phase, exported versions of the free receivers of the local process term are registered with the naming service, thus making those names visible to other interpreters.

Implementing Remote Communication So far, a pickled name has been viewed and implemented as

[7] Creating a new binding should be understood as setting up access and communication paths between components of a distributed system. Bindings may come in a wide variety of semantics: mobile, persistent, with a QoS guarantee ... Thus, an adaptable ORB framework should provide means to define bindings with various semantics, and to combine them in flexible ways.

[8] With some simple architectural principles such as separating marshalling from protocol implementation, or threading from resource management, JONATHAN allows, for instance, to dynamically introduce new protocols, or to control the level of multiplexing of resources.

a communication capability on some remote channel. Further work is needed to perform a real communication: a new binding must be created to establish a communication path to the object representing the remote channel. That operation is performed using the JONATHAN framework `bind()` method. Emission of a message $\langle x, y \rangle$ on remote channel x, where y is a pickled name represented by the `RemoteName` object `y_surrogate` is implemented by:

```
RemoteName c =
  (RemoteName) orb.bind((RemoteName) x);
c.output(y_surrogate);
```

The first call builds an access path to remote channel x. It recursively performs `id.bind()` calls on lower-level binders, where `id` is the binding data contained at the reference-level in `y_surrogate`, thus creating a stack of client-side session objects. Once the communication path for messages is established between the two interpreters, the second call sends the reference part of y down the stack of sessions.

When a message comes up the stack of sessions to the `LocalName` object representing a local receiver, the identifier `id` representing the argument is unmarshalled, and the `id.bind()` method is invoked. It yields a surrogate for the argument, or the argument itself if it is a local name. The state of the interpreter is then modified accordingly.

The various refinements concerning naming and remote communication, and performed during the different formalization steps of the DCP experiment are summarized in Table 1.

7 Conclusion

By analyzing the formalization steps from the design of a domain-based distributed programming model to its middleware implementation, we have shown on a case-study the feasibility and the soundness of a simple methodology for building reliable and cleanly designed infrastructures for distributed computing: the approach focuses on using a carefully designed model with a well-defined semantics and an implementation that strictly conforms to the model, obtained by refinement of the model. We introduced DCP, a simple extension of the PICT concurrent language with domain and remote interaction primitives. We formalized its implementation by an abstract machine (AM). DCP was implemented by a distributed runtime, written in Java, seamlessly integrated with a typical middleware platform, the JONATHAN ORB to provide support for network communication transparency.

DCP domains are not endowed with any behavior: to obtain truly programmable domains, richer forms of control are desirable at the language-level, e.g. to create new domains, to filter incoming or outgoing messages, etc. To capture such situations of distributed mobile programming, a more powerful calculus is required, and is currently under definition. Thanks to the DCP experiment, we have been able to start developing a software infrastructure for such a calculus, by first defining an AM specification and then implementing a domain-based virtual machine. Future work includes examining scalability issues arising in a tree-structured domain topology, more adapted to wide-area network programming, and taking into account communications security, and process mobility.

DCP is one of the first process calculi both to formalize its implementation in terms of an AM in a distributed setting, and to be implemented on top of an ORB. By making remote interaction constructs as primitive, we allow the model to better reflect the form a provably correct, fully-fledged implementation would take, using available middleware technology. We also make it possible to achieve compatibility with existing de facto standards in the area like CORBA or Java RMI – an important prospect for programming distributed applications in a real, heterogeneous setting.

Acknowledgments We wish to thank Sacha Krakowiak and Jean-Bernard Stefani for their many valuable comments on an earlier version of the paper.

References

[1] R. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Proceedings COORDINATION'97*, LNCS 1282, 1997.

[2] R. Amadio, G. Boudol, and C. Lhoussaine. The Receptive Distributed Pi-Calculus. In *Proceedings FSTTCS'99*, LNCS 1738, pages 304–315, 1999.

[3] L. Cardelli. Abstractions for Mobile Computation. In J. Vitek and C. Jensen, editors, *Secure Internet*

Design Stage	Calculus	Abstract Machine	Distributed Runtime
Naming model	Channel names	Local names	Local names (`LocalName` objects)
		Pickled Names	Exported Names (`RemoteName` objects)
Name creation	ν operator	Creation of a local name : rule [RES]	Creation of a new `LocalName` instance. New names are automatically exported
Scope extrusion	Rule $a[(\nu\ n)\ P] \equiv (\nu\ n)\ a[P]$	pickle operation	`export()` on local names
			Marshalling / unmarshalling of exported names
Inter-domain communication	Rules (send) and (receive)	pickle operation : rules [REMXMIT] and [REMRCV]	Creation of a new binding : `bind()` on exported names
			Asynchronous messaging on TCP bindings

Table 1. Refinements for Naming and Remote Interaction in the DCP Experiment

Programming: Security Issues for Mobile and Distributed Objects, LNCS 1603, pages 51–94, 1999.

[4] L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240:177–213, 2000.

[5] B. Dumant, F. Dang Tran, F. Horn, and J.-B. Stefani. Jonathan : an Open Distributed Platform in Java. In *Proceedings MIDDLEWARE'98*, 1998.

[6] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings POPL'96*, pages 372–385, 1996.

[7] C. Fournet, G. Gonthier, J.-J. Levy, L. Maranget, and D. Rémy. A Calculus of Mobile Agents. In *Proceedings CONCUR'96*, LNCS 1119, pages 406–421, 1996.

[8] F. Germain, E. Najm, and J.-B. Stefani. Elements of an Object-Based Model for Distributed and Mobile Computation. In *Proceedings FMOODS'00*, 2000.

[9] International Standards Organization. *ODP Reference Model: Foundations*, ISO/IEC JTC1/SC21/WG7 10746-2 edition, 1995.

[10] M. Lacoste and J.-B. Stefani. DCP: Towards Distributed Programming with Domains. Unpublished draft.

[11] F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proceedings POPL'00*, pages 352–364, 2000.

[12] L. Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, Faculdade de Ciências da Universidade do Porto, 1999.

[13] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: an Experiment in Code Mobility from the Realm of Process Calculi. In *Proceedings 5th Workshop on Mobile Object Systems*, 1999.

[14] L. Lopes and V. Vasconcelos. An Abstract Machine for an Object Calculus. Technical Report DCC-97-5, Facultade de Ciências da Universidade do Porto, 1997.

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

[16] B. Pierce and D. Turner. Pict: a Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Indiana University, 1997.

[17] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[18] D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings ICALP'01*, 2001.

[19] P. Sewell, P. Wojciechowski, and B. Pierce. Location-Independent Communication for Mobile Agents: a Two-Level Architecture. Technical Report 462, University of Cambridge, 1999.

[20] D. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.

[21] A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructure for Mobile Computation. In *Proceedings POPL'01*, pages 116–127, 2001.
[22] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *Proceedings ISOTAS'93*, LNCS 742, pages 460–474, 1993.
[23] P. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, Massachusetts, 1995.

A Proof of Subject Reduction Theorem (Theorem 1)

We first prove a few useful properties on environments.

Lemma 1 (Environments) *Let Γ be an environment, P a process and C a configuration.*

1. *If $\Gamma \vdash P$, then $\Gamma \subset \mathsf{fn}(P)$.*
2. *If $\Gamma \vdash C$, then $\Gamma \subset \mathsf{fn}(C)$.*
3. *If $\Gamma \vdash P \mid P$, then $\Gamma \vdash P$.*

Proof By induction on the structure of P for cases 1 and 3. By induction on the structure of C for case 2. □

Lemma 2 (Subject Congruence) 1. *If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$.*
2. *If $\Gamma \vdash C$ and $C \equiv D$, then $\Gamma \vdash D$.*

Proof For case 1, the proof is by simultaneous induction on the derivation of $P \equiv Q$ and $Q \equiv P$ and the use of lemma 1 for the case $*P \equiv P \mid *P$. The proof of case 2 is similar. □

Definition 2 (Substitution) *Let u,v be two channel names, and E a set of channel names. Then $\sigma_{uv}(E) \stackrel{def}{=} E$ if $u \notin E$, and $(E \setminus \{u\}) \cup \{v\}$ otherwise.*

Lemma 3 (Substitution) *If $\Gamma \vdash P$, then $\sigma_{uv}(\Gamma) \vdash \{v/u\}P$.*

Proof By induction on the structure of P. We limit ourselves to the case where $u \in \mathsf{fn}(P)$, since otherwise $\{v/u\}\ P = P$, and $\sigma_{uv}(\Gamma) = \Gamma$ since $\Gamma \subset \mathsf{fn}(P)$ by lemma 1. □

We can now prove the subject reduction theorem.
Proof of theorem 1 By induction on the derivation of $P \longrightarrow Q$ for case 1 (of $C \longrightarrow D$ for case 2), and by case analysis of the last rule used. We only show the main cases.

Case (equiv-p) : $P \equiv P'$, $P' \longrightarrow Q'$, $Q \equiv Q'$. From $\Gamma \vdash P$ and lemma 2, we have $\Gamma \vdash P'$. Using the induction hypothesis, there is Δ such that $\Delta \vdash Q'$. Finally, again by lemma 2, $\Delta \vdash Q$.

Case (com) : $P = m?(u).R \mid m!\langle v\rangle$, $Q = \{v/u\}R$. By [T-PAR-P], $\Gamma = \Gamma_1 \cup \Gamma_2$, with $\Gamma_1 \vdash m?(u).R$, and $\Gamma_2 \vdash m!\langle v\rangle$. By [T-SND], $\Gamma_2 = \emptyset$. By [T-RCV], there is Γ_0 such that $\Gamma = (\Gamma_0 \setminus \{u\}) \cup \{m\}$, $\Gamma_0 \vdash R$. By lemma 3, $\Delta = \sigma_{uv}\Gamma_0 \vdash \{v/u\}R$, finishing the case.

Case (send) : $C = a[P \mid m!\langle v\rangle]$, $D = a[P] \parallel \langle m, v\rangle$, $m \notin \mathsf{dn}(P)$. By [T-DOM], $\Gamma \vdash P \mid m!\langle v\rangle$. By [T-PAR-P] and [T-SND], we derive $\Gamma \vdash P$. Hence, $\Gamma \vdash a[P]$ by [T-DOM]. Using [T-MSG] and [T-PAR-C], we finally derive $\Gamma \vdash a[P] \parallel \langle m, v\rangle$.

Case (receive) : $C = \langle m, v\rangle \parallel a[P]$, $D = a[P \mid m!\langle v\rangle]$, $m \in \mathsf{dn}(P)$. The proof is similar to case (send).

□

B Proof of Fairness Theorem (Theorem 2)

Lemma 4 (Well-Formedness) *If $S \longrightarrow S'$ and S is a well-formed AM state, then S' is a well-formed AM state.*

Proof By induction on the derivation of $S \longrightarrow S'$. □

Lemma 5 (Deadlock) *If $S = (a, \mathcal{U}, \mathcal{V}, \mathcal{H}, \mathcal{R})$ is well-formed and $S \not\longrightarrow$, then $\mathcal{R} = \emptyset$.*

Proof By case analysis on the structure of the head of R: if $\mathcal{R}$ is non-empty, then S is reducible. □

Definition 3 (Instructions) *The number $\| P \|$ of instructions required to execute a process P is defined as follows:*

$$\begin{aligned}
\| P \mid Q \| &\stackrel{def}{=} 1 + \| P \| + \| Q \| \\
\| 0 \| &\stackrel{def}{=} 1 \\
\| x!\langle \tilde{y} \rangle \| &\stackrel{def}{=} 1 \\
\| (\nu\ n)\ P \| &\stackrel{def}{=} 1 + \| P \| \\
\| x?(\tilde{y}).P \| &\stackrel{def}{=} 1 + \| P \| \\
\| *x?(\tilde{y}).P \| &\stackrel{def}{=} 1 + \| P \| \\
\| [u\ =\ v]\ P\ ,\ Q \| &\stackrel{def}{=} 1 + \max(\| P \|, \| Q \|)
\end{aligned}$$

Proof Sketch of theorem 2 Consider an AM a with process P just placed at the end of the run queue. The state of a is $S = (a, \mathcal{U}, \mathcal{V}, \mathcal{H}, Q_1 :: \cdots :: Q_m :: P)$. For any state $T = (a, \mathcal{U}', \mathcal{V}', \mathcal{H}', R_1 :: \cdots :: R_l :: P :: \mathcal{R})$ such that $S \stackrel{*}{\longrightarrow} T$, we define the total number of instructions to run to start the execution of P by:

$$\Phi(T) \stackrel{def}{=} \sum_{i=1}^{l} \| R_i \| \text{ if } l \neq 0 \,;\, \Phi(T) \stackrel{def}{=} 0 \text{ if } l = 0$$

Note that $\Phi(T) = 0$ iff P is ready to run.
We first prove that:

(i) If $T \longrightarrow T'$, then $\Phi(T') = \Phi(T)$, if the reduction is derived using either rules [REDREPL], [REMXMIT] or [REMRCV], and $\Phi(T') < \Phi(T)$ otherwise.

(ii) There is no infinite fair derivation $S \longrightarrow U_1 \longrightarrow \cdots \longrightarrow U_p \cdots$ such that $\forall i \geq 1, \Phi(U_i) = \Phi(S)$.

For (ii), assume that there is such a derivation. Thus, using (i), each reduction step can only be derived using [REDREPL], [REMXMIT], or [REMRCV]. If $Q_1 = *x?(\tilde{y}).P_0$, [REDREPL] is only applicable at most $|\mathcal{H}(x)|$ times without Φ strictly decreasing. Similarly, [REMXMIT] is applicable at most $|\mathcal{V}| \times \max_{y \in \mathcal{V}} |\mathcal{H}(y)|$ times. Thus, there is N such that $i \geq N$ implies $U_i \longrightarrow U_{i+1}$ are all listening steps. This contradicts with the fairness assumption.

Let $S \longrightarrow T_1 \longrightarrow \cdots \longrightarrow T_p \cdots$ be a fair derivation. If there is $i \geq 1$ such that $T_i \not\longrightarrow$, since S is a well-formed AM state, we infer from lemmas 4 and 5 that the run queue of T_i is empty. Thus, P will already have been run. Hence, we can assume the derivation to be infinite. Then, the fact that there is $i \geq 1$ such that $\Phi(T_i) = 0$ is an easy consequence of (i) and (ii), which concludes the proof.

□