Università di Pisa
Dipartimento di Informatica

Technical Report: TR-04-01

Resources Negotiation in Systems of Mobile Components

Gianluigi Ferrari Emilio Tuosto

January 20, 2004

Address: Via F. Buonarroti 2, 56127 Pisa, Italy.
Tel: +39 050 2212700 — Fax: +39 050 2212726

Resources Negotiation in Systems of Mobile Components

Gianluigi Ferrari Emilio Tuosto

January 20, 2004

Abstract. We introduce a process calculus that contains constructs to express and program resource negotiations in systems of mobile processes. The calculus proposed is an extension of Distributed π-calculus [6] and, therefore, it provides process mobility in networks that may dynamically change their topology. Negotiation takes place between processes, asking credits to achieve their purposes, and locations, that release credits depending on their availability.

We describe a type system for the calculus which guarantees that well-typed processes cannot fail because of misuse of credits over resources. The safety result is more involved due to process mobility: A migrating process should not access locations that do not guarantee safety.

1 Introduction

uring their lifetime, applications require different resources ranging from hardware devices to software components. Resources can be abstractly thought of as special services that applications require to achieve their functionalities. Current software technologies put a special emphasis in applications over wide-area networks. The software model which underpins applications over wide-area networks is quite different from the client-server model which characterized standard distributed applications. In particular, in a wide-area network we cannot assume that all needed resources are known in advance and that they are all available before execution. Moreover, the behaviour of applications is highly exposed to and depends on the dynamic changing of the network topology. For instance, a whole sub-net may detach from the network and re-attach later with different resources and different access control policies.

An emerging design paradigm for structuring applications is *code mobility*. Mobility allows applications to migrate between sites of the net. Hence, nomadic applications roam through different administration domains and their binding to resources is strictly dependent on the security policy fixed by the authority of the administration domain that applications are visiting. In other words, the availability of certain resources is regulated by the security constraints of the administration domain.

At a foundational level, several process calculi have been developed to gain a more precise understanding of mobility. In particular, some approaches addressed the problem of resource access control for mobile processes [4, 6, 7, 1, 5]. These works led to the introduction of suitable type systems where types specify and fully characterize access control policies. A subject reduction theorem ensures that well typed programs cannot fail because of misuse of the resource they access. To our knowledge, what is missing from all of these is the ability of dynamic *negotiation* the amount of resources nomadic applications require in order to maintain their guarantees.

In this paper we introduce an extension of Hennessy and Riely's Distributed π-calculus, Dπ [6]. Dπ provides notations and formal machineries to express migrating processes, administration domains (localities) and types for access control. We equip Dπ with facilities to handle resource negotiation and to monitor resource consumption.

In our approach, types provide an abstraction of the level of usability of the resources. For instance, let us consider a host providing a service s. The type of the host can be used to measure the availability of the service among its clients to avoid "denial of services" attacks. Similarly, a nomadic user may search across the network to find a host granting certain services with a high availability (again expressed by a suitable type).

We measure resource availability and resource consumption by the notion of *credit*. Intuitively, credits are tokens on resources granted to processes: whenever a process accesses a resource it consumes some of the tokens it owns on that resource (e.g. nomadic applications are charged according to the agreement with the service provider).

A first contribution of the paper is the introduction of basic facilities to express and program *resource negotiations*. Negotiation takes place between processes asking credits to achieve their functionalities and locations that release credits depending on their availability and security policy. Credits are divided in two categories: *virtual credits* and *real credits*. Real credits can be effectively consumed by processes. Virtual credits must be renegotiated before their effective usage.

Negotiation can occur by the execution of a local operation (acquire). Migrating processes must explicitly re-negotiate their capabilities on the target location. Alternatively, migration could give rise to an implicit re-negotiation before the execution on the target node starts. During this phase, mobile applications ask for the amount of resources they needs to achieve their functionalities[1].

A process locally asks to the location credits on a resource u expressing a maximal credit, C_M, and a minimal credit, C_m. The location grants credits to the process in the range C_m, C_M. The behaviour of the local credit acquisition is illustrated in Fig. 1. We show three possible situations: a) the process must wait because no credits are available; b) the process totally gains real credits; c) the process gains a real credit between C_m and C_M.

[1]This alternative approach allows one to not consider negotiations triggered by migration, however, this choice make much more complex the formalization of the calculus semantics.

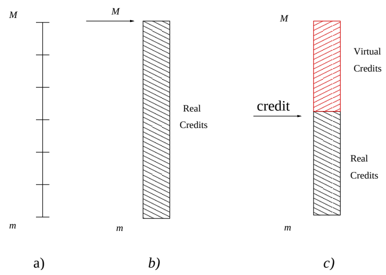

Figure 1: Locally

Since locations act as brokers of their resources, each location stores credit information of both the running applications and of its resources into a suitable structure called the *credit environment*. The credit environment has an irregular structure and may change dynamically. In particular, resources availability and their access modalities are not known in advance. Therefore, we represent the credit environment as a dynamic tree-like structure (i.e. a semi structured data as defined by Cardelli and Ghelli [2]). In our approach, the negotiation phase is modeled as the result of a query over the credit environment. The query language is a simple variation of the TQL language [2].

A second contribution of this paper is the introduction of a notion of type which allows us to measure resource availability and resource usage. Indeed, we extend Dπ capability types by introducing *resource capabilities*. The main results are a subject reduction and type safety property. These results ensure that migrating well-type processes cannot "consume" more resources than the amount of credits acquired in negotiation phases.

This work has strict connections with [10], where a language for controlling resource consumption is described. Their metaphor is that applications need energy for computing and they must acquire tanks of energy that are released by sponsoring a group. The calculus does not make assumption an the distribution model, but it appear that only CCS-like [8] systems are dealt. Here we also encompass systems that may dynamically change their topology. Moreover, we provide a type system and a subject reduction result that guarantees absence of errors in resource negotiation/consumption.

2 Resources & Credits: An abstract view

2.1 Abstract Resources

An application A may access a resource in different ways which depend on some conditions and on the "rôle" played by A at access-time. For instance, if we need to perform some calculation, we may require a server and use it as a normal user. However, if we want to print the results of some computation with several photos we also would need a scanner and a printer (i.e. we become a "special" user that is granted for printing and scanning). Indeed, an application may access the same resource by exploiting different capabilities. Moreover, applications may need pools of resources, all at once. The above observations suggest that

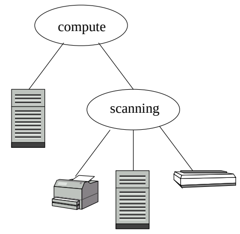

Figure 2: Resource hierarchy metaphor

resources are naturally described by a hierarchical structure (see Figure 2.1 for a pictorial representation of the server metaphor discussed above).

Definition 1 introduces a specific syntax to describe resources. The language is a fragment of the *Ambient* calculus [3] that is suitable for expressing forests of trees.

Definition 1 *Let* $\mathcal{N}$ *be a set of* resource names *ranged over by* r*. A* resource *is a term generated by the grammar*

$$R ::= r[\,] \quad | \quad R|...|R \quad | \quad r[R] \quad | \quad 0$$

Intuitively, $r[]$ describes an *elementary resource*; $R_1|...|R_n$ represents the *resource aggregate* obtained by concatenating R_1,...,R_n; $r[R]$ represents a resource r "containing" the sub-resources specified by R; finally, 0 is the empty resource.

Example 1 *Let* R *be* $r[r_1[] \mid r_2[]]$*, namely,* R *is a resource aggregate containing two elementary resources* r_1 *and* r_2*. The resource* $r'[r \mid r_3[]]$ *is obtained by aggregating* r *together with an elementary resource* $r_3[]$ *and may be graphically represented as*

$$\begin{array}{ccccc} & & & r' & \\ & & / & & \backslash \\ & r & & & r_3 \\ / & & \backslash & & \\ r_1 & & r_2 & & \end{array}$$

We assume that $|$ is commutative, associative and 0 is its neutral element, in other words $(R, |, 0)$ is a commutative monoid.

2.2 Abstract Credits

Credits constitute the "amount" of resources and are the unit of negotiation. Negotiation process is roughly a sequence of interactions between a resource provider and an application: The application tries to obtain credits that would allow certain operations on the resources, while the provider leases/revokes credits depending on a policy (for instance, a fairness requirement among applications). Credits must be equipped with some algebraic structure for adding, decreasing or completely revoking them. Furthermore, credits should be compared because it is necessary to "measure" the amount of resources granted to applications. The algebraic structure that satisfies these requirements is a group structure over an ordered set.

Definition 2 *Let* $Gr = (G, +, 0, \leq)$ *be an ordered group*[2] *and let* $\mathcal{M} = \{\mu_1, ..., \mu_k\}$ *be a set of* modalities. *The set* $\mathcal{EC}$ *of* elementary credit *over* Gr *and* $\mathcal{M}$ *is the set of terms of the form*

$$g_1\mu_1 + ... + g_k\mu_k, \qquad \textit{where } \; g_1, ..., g_k \in G.$$

The additive operation $+$ *can be lifted to* $\mathcal{EC}$ *by defining*

$$(g_1\mu_1 + ... + g_k\mu_k) + (g'_1\mu_1 + ... + g'_k\mu_k) \stackrel{\triangle}{=} (g_1 + g'_1)\mu_1 + ... + (g_k + g'_k)\mu_k.$$

Assuming the order $\mu_1 < \mu_2 < \ldots < \mu_k$ on $\mathcal{M}$, then

$$(g_1\mu_1 + ... + g_k\mu_k) \leq (g'_1\mu_1 + ... + g'_k\mu_k) \iff \exists j : \; 1 \leq j \leq k : \forall i < j : g_i = g'_i \wedge g_j < g'_j$$

is a total order on elementary credit.

Let $g_1\mu_1 + ... + g_k\mu_k$ be an elementary credit, we avoid writing $g_i\mu_i$ whenever $g_i = 0$; we write 0 if $g_i = 0$ for all $i = 1, ..., k$. Furthermore, $-g_1\mu_1 ... -g_k\mu_k)$ is used as a shorthand for $(-g_1)\mu_1 + ... + (-g_k)\mu_k$. We use c as a meta-variable on elementary credits and $-c$ is also denoted with $\bar{c}$. The following proposition may be easily proved.

Proposition 1 *If* $\mathcal{M}$ *is totally ordered then* $(\mathcal{EC}, +, 0, \leq)$ *is an ordered group.*

2.3 Credit Environments

In our approach, resource providers monitor the accesses of applications to resources as long as computation proceeds. Hence, we need to equip providers with a suitable structure that keeps track of resources and their availability; we call such structure *credit environment*.

It is quite natural to represent credits as a tree-like structures (*Ambient*-trees) that may be accessed and modified by using the query language introduced in Section 2.4.

Definition 3 *A* credit *is a term derived from the grammar*

$$C ::= r[C] \quad | \quad C|C \quad | \quad r[c \otimes c'].$$

[2] $(G, +, 0, \leq)$ is an ordered group if $\leq$ is a total order on G, $0 \in G$ is the *unit* or *neutral* element of the group and $+ : G \times G \to G$ is a binary associative operation on G such that

- $\forall g, g_1, g_2 \in G : g_1 \leq g_2 \Rightarrow g_1 + g \leq g_2 + g \;\; \wedge \;\; g + g_1 \leq g + g_2$,
- $\forall g \in G : \exists g' \in G : g + g' = g' + g = 0$ (g' is denoted with $-g$ and is called the *inverse of* g. As usual, $g_1 + (-g_2)$ is written as $g_1 - g_2$).

We have already pointed out that credits granted by providers may be fewer than the amount requested by applications. In this case, the credits that cannot be leased are *virtualized*. Namely, they are recorded as credits that may be bargained later on. Intuitively, the environment $r[c_1 \otimes c_2]$ records that real credit c_1 and virtual credit c_2 are associated to resource r.

Definition 4 *Let $\mathcal{I}$ be a set of* application names *and let $\mathfrak{s}$ be distinguished name in $\mathcal{I}$. The class of Δ of* credit environments *is the set of partial functions with domain $\mathcal{I}$ and codomain C. Δ is ranged over by δ.*

Credits $\delta(\mathfrak{s})$ are the amount of residual credit that a resource provider may release to requesting applications. Hereafter, we let id range over $\mathcal{I} \setminus \{\mathfrak{s}\}$ and let I range over $\mathcal{I}$. Moreover, we write $\delta(I) = \perp$ if δ is not defined in I.

It is useful to introduce some operations. Operator $\oplus : C \times C \to C$ is used to sum up credits. It is a partial function; indeed $C \oplus C'$ is defined only if C and C' have the same tree-structure and differs only on the value of their elementary credits.

$$\begin{array}{rcl} r[c_1 \otimes c_2] \oplus r[c_1' \otimes c_2'] & = & r[(c_1 + c_1') \otimes (c_2 + c_2']] \\ (r_1[C_1] \mid \ldots \mid r_k[C_k]) \oplus (r_1[C_1'] \mid \ldots \mid r_k[C_k']) & = & r_1[C_1 \oplus C_1'] \mid \ldots \mid r_k[C_k \oplus C_k'] \end{array}$$

The zero function, $\mathcal{Z}$ applied to C cancels the credit information in C while the real/virtual credits projections, $\Re/\mathcal{V}$, cancels the virtual/real part[3]:

$$\begin{array}{rclcrcl} \mathcal{Z}(r[c_1 \otimes c_2]) & = & r[0 \otimes 0] & \quad & \Re(r[c_1 \otimes c_2]) & = & r[c_1 \otimes 0] \\ \mathcal{Z}(C \mid C') & = & \mathcal{Z}(C) \mid \mathcal{Z}(C') & & \Re(C \mid C') & = & \Re(C) | \Re(C') \\ \mathcal{Z}(r[C]) & = & r[\mathcal{Z}(C)] & & \Re(r[C]) & = & (r[\Re(C)]) \end{array}$$

2.4 A query language

This section informally reviews the basic ideas of the query language for Ambient trees, TQL. The reader is referred to [2] for the formal treatment.

In our approach, TQL is used to retrieve information and modify allocation environments. Negotiation is modeled as a query on a credit environment which returns an updated environment. The query language provides mechanisms for selecting values and binding them to variables in combination with a mechanism for constructing results starting from bindings. A TQL query has the structure

$$from\ T \models\ \phi\ select\ Q \tag{1}$$

where T is a credit tree, ϕ is a logical formula and Q is either a query or a tree. Formula ϕ may contain variables that are bound in Q.

Example 2 *The formula $\phi \stackrel{\text{def}}{=} n.m[0] \mid \neg m[tt]$ denote the class of trees containing a path labelled with n and m and another component that does not have a top-level tree m. For instance, $T \stackrel{\text{def}}{=} n[m[0] \mid a[0]]$ satisfies ϕ. On the other hand, $T \mid m[0]$ does not satisfy ϕ (for any T).*

[3] We define only the real credit projection. The virtual credit may be defined analogously.

Intuitively, the semantics of query (1) is defined in terms of the set of substitutions σ of variables of ϕ such that T satisfies $\phi\sigma$. The result of the query is the parallel composition of the trees obtained by (the evaluation of) all $Q\sigma$.

Example 3 *Let us consider the query*

$$from\ n[m[0] \mid n[m[0]]] \models\ n[\chi \mid tt]\ select\ \chi.$$

The substitution determined in the select/bind mechanism are $\sigma_1 : \chi \mapsto m[0]$ *and* $\sigma_2 : \chi \mapsto n[m[0]]$. *The result of the query evaluation is the tree* $m[0] \mid n[m[0]]$.

3 The Core Calculus

The distributed π-calculus $D\pi$ [6] is an extension of π-calculus [9], where processes and channels are allocated on *locations*. A process may roam through locations and communication cannot be remote: Two processes may synchronize only if they run on the same location. In this section we introduce a calculus which extends $D\pi$ with facilities for resource negotiation.

3.1 Negotiation Types

$D\pi$ elementary resources are allocated channels which are managed by locations. A process A may access channels by reading/writing from/on them. Hence, the set of modality is $\mathcal{M}_{D^R_\pi} = \{\rho, \omega\}$, i.e. the reading and the writing modality, respectively. We assume that $\rho < \omega$. If we assume that credits are used to measure the number of access to a channel, then the support group is the group of integers and $i\rho + j\omega$ expresses how many readings/writings may be performed.

Both location and resource types are defined by *capabilities* that describe credits granted to processes by locations. Resource capabilities describe how resources can be used by processes.

A capability κ is written as $r : T_R$, where T_R, is defined below, is a resource aggregate type. Hereafter, (possibly void) tuples $(X_1, ..., X_n)$ will be written as $\tilde{X}$.

$$T_R ::= \alpha \ \mid\ \widetilde{T_R} \qquad \alpha ::= \tau \ \mid\ \langle\widetilde{\zeta}\rangle \qquad \zeta ::= \alpha \ \mid\ L \qquad L ::= [\tilde{\kappa}]$$

A channel type α is a simple type τ (e.g. integer, string, etc.) or $\langle\widetilde{\zeta}\rangle$ that represents capabilities for reading/writing values of type ζ along a channel. A location type L is $[r_1 : T_{R_1}, ..., r_n : T_{R_n}]$, where r_i are names of resources allocated on the location and T_{R_i} are their types. Types of exchangeable values ζ are location types or allocated channel types. Resource aggregates have a type obtained by composing the types of their components.

We write $\kappa \in L$ to indicate that the capability κ appears in L.

Definition 5 *A location type* L *is* well defined *if* $r : T_R \in L\ \wedge\ r : T'_R \in L \Rightarrow T_R = T'_R$. *Types* ζ *and* $L[\zeta]$ *are* well defined *if they are structurally well defined.*

3.2 The D_π^R calculus

D_π^R syntax[4] is defined in Table 1. We will discuss only our extension to $D\pi$ and refer the reader to [6] for a deeper description of $D\pi$. Processes are essentially

$$
\begin{array}{lcl}
P - R & ::= & 0 \\
 & | & P \mid Q \\
 & | & !P \\
 & | & if\ BE\ then\ P\ else\ Q \\
 & | & go\, l.P \\
 & | & S!\langle U\rangle.P \\
 & | & S?(X:\zeta).P \\
 & | & (r' \leftarrow \widetilde{r_C}).P \\
 & | & \$^{C_M}_{C_m}\langle u\rangle.P
\end{array}
\qquad
\begin{array}{lcl}
M - N & ::= & 0 \\
 & | & M \mid N \\
 & | & l[\widetilde{id :: P}]_\delta \\
 & & \\
BE & ::= & a = b \ \mid\ \ldots \\
X - Z & ::= & x \ \mid\ z[\tilde{x}] \ \mid\ \tilde{X} \\
U & ::= & e \ \mid\ l[\tilde{e}] \ \mid\ \tilde{U} \\
S & ::= & r.S \ \mid\ a
\end{array}
$$

Table 1: D_π^R Syntax

built out from π-calculus operators, namely, empty process, parallel composition, replication, prefix and match/mismatch (guards of conditional processes may be enriched, however, for simplicity, we only consider classical match and mismatch operator of π-like calculi). However, some features for expressing resource negotiation and resource nesting are added. Prefix *go* l allows a process to move from its location to a remote location l. Output and input prefixes are $S!\langle U\rangle.P$ and $S?(X : \zeta).P$; S is a path reaching a channel name in a resource tree-structure, $leaf(S)$ denotes such channel. For instance, in Example 1, $r'.r.a$ selects the channel a occurring in r' (i.e. $leaf(r'.r'a) = a$). The bind prefix $(r \leftarrow (a_{c_a}, b_{c_b}).P$ composes channels a and b yielding a new resource aggregate, r, whose credits are $c_a|c_b$. The scope of r is P (r is meant to be bound in P). Bind is a blocking atomic operation: P waits until all resources are available with requested credits. Note that bind operator is similar to name restriction of $D\pi$. Last action is acquire; the prefix $\$^{C_M}_{C_m}\langle u\rangle.P$ defines a process waiting for capabilities in the range $[C_m.C_M]$ on resource u. An application $id :: P$ is a process P named with $id \in \mathcal{I}$; applications may be composed in parallel and we let A to range over application terms, while $a(A)$ denotes the application names occurring in A.

A system M is made of the parallel composition of located applications. Each location $l[A]_\delta$ has an associated credit environment δ and a number of applications running in parallel. We distinguish between locations indicated by $k, l, \ldots$ and resources indicated by $a, r, \ldots$; when the difference does not matter we use e.

[4]For simplicity, the restriction of location names is not considered here.

(MON-p)	$\mid$ is associative and commutative and 0 is its identity
(ALPHA-p)	If P differs from Q by alpha-conversion $P \equiv Q$
(BANG-p)	$!P \equiv P \mid !P$
(PAR-a)	$\mid$ is associative and commutative
(SPLIT-a)	$id :: (P \mid Q) \equiv id :: P \mid id :: Q$
(APP-a)	$\dfrac{P \equiv Q}{id :: P \equiv id :: Q}$
(MON-s)	$\mid$ is associative and commutative and 0 is its identity
(APP-s)	$\dfrac{A \equiv A'}{k[A]_\delta \equiv k[A']_\delta}$
(DELTA-s)	$\dfrac{id \notin a(A)}{k[A]_\delta \equiv k[\ A[id/id']\ ]_{\delta[id/id',\perp/id]}}$

Table 2: D_π^R Structural Congruence

3.3 The Operational Semantics

D_π^R operational semantics is defined by a reduction semantics that relies on structural congruence rules for processes, applications and systems which are defined in Table 2. Structural rules for processes, (MON-p), (ALPHA-p) and (BANG-p), are the standard rules for π-calculus like process algebras. Rule (PAR-a) states that order in which applications are put in parallel is not relevant. Note that applications do not have an identity element. Indeed, possibly candidates are applications of the form $id :: 0$, but, as we will see later, such applications must returns the residual credits that they did not use in their computation to their running location. Rules (MON-s) and (APP-s) are obvious. Rule (DELTA-s) states that application names are not relevant and can be changed provided that the renaming is reflected in the credit environment.

Definition 6 *A system M is well-formed if, for each $k[A]_\delta$ occurring in M, is $id \notin a(A)$ then $\delta(id) = \perp$.*

System well-formedness ensures that application renaming does not have undesidered credits assignments and that we may locally chose names for applications. We will show, reduction semantics of D_π^R preserves system well-formedness.

We introduce some auxiliary operations necessary for specifying the semantics.

Definition 7 *The function* $\gamma : \Delta \times \mathcal{I} \setminus \{\mathfrak{s}\} \to C$ *is the* garbage function *and is defined as*

$$\gamma(\delta, id) = \begin{cases} \delta, & \text{if } \delta(id) = \perp \\ \gamma(\delta', id), & \text{if } \delta(id) \neq \perp, \delta' = \delta[C'/\mathfrak{s}, C''/id] \end{cases}$$

where $C' = from\ \delta(id) \models\ r[C_1] \mid tt, \delta(\mathfrak{s}) \models r[C_2] \mid C_3\ select\ r[C_2 \oplus \Re(C_1)] \mid C_3$ *and* $C'' = from\ \delta(id) \models\ r[C_1] \mid C_2\ select\ C_2$*.*

Garbage function γ, when applied to δ and id, removes all the credits acquired by id and gives them back to the location. Another useful operation allows to add credits along a path of a credit:

Definition 8 *Given a path expression* S*, we let* $\oplus_S : C \times C \to C$ *be the function defined by*

$$C \oplus_S c = \begin{cases} r[c' + c], & \text{if } C = r[c'] \\ r[C' \oplus_{S'} c], & \text{if } C = r[C'] \wedge S = r.S' \\ r[C' \oplus_{S'} c] \mid C', & \text{if } C = r[C'] \mid T'' \wedge S = r.S' \end{cases}$$

Note that $C \oplus_S C'$ *is defined only if* $C \models S[tt]$ *and* C' *is an elementary credit.*

The reduction relation is defined in Table 3 and Table 4 together with the rules

$$\frac{M \to M'}{M|N \to M'|N} \qquad \frac{M \equiv N \quad N \to N' \quad N' \equiv M'}{M \to M'}$$

$$\frac{id \notin a(A)}{k[id :: 0 \mid A]_\delta \to k[A]_{\gamma(\delta, id)}}\ \textbf{Garbage}$$

$$\frac{\begin{array}{c}\delta(\mathfrak{s}) \models C \mid X \quad C \models S[c] \quad \delta(id_i) \models C_i \mid X_i \wedge C_i \models S[tt], i = 1,2 \\ 1\rho \leq C_1, \quad \omega \leq C_2 \\ \delta' = \delta[(C_1 \oplus_S \overline{1\omega}) \mid X_1/id_1][(C_2 \oplus_S \overline{1\rho}) \mid X_2/id_2][C \oplus_S (1\rho + 1\omega)/\mathfrak{s}]\end{array}}{k[id_1 :: S!\langle U\rangle.P_1 \mid id_2 :: S?(X : \zeta).P_2 \mid A]_\delta \to k[id_1 :: P_1 \mid id_2 :: P_2[U/X] \mid A]_{\delta'}}\ \textbf{Sync}$$

$$\frac{u = v}{k[id :: if\ u = v\ then\ P\ else\ Q \mid A]_\delta \to k[id :: P \mid A]_\delta}\ \textbf{Match}$$

$$\frac{u \neq v}{k[id :: if\ u = v\ then\ P\ else\ Q \mid A]_\delta \to k[id :: P \mid A]_\delta}\ \textbf{Mismatch}$$

Table 3: Reduction Semantics I

We comment on garbage and synchronization rules; conditional rules being obvious. **Garbage** rule handles credits of terminated applications. The hypothesis of the rule guarantees that id does not appear in A. The effect of the reduction is the re-assignment of (real part of) the credits of each resource of id to the credit environment of location k. Synchronization is not always possible: Indeed, **Sync** rule requires that the sender owns at least one credit for writing and the receiver has at least one credit for reading. After communication has taken place, the consumed credits are added to the location credit and, correspondingly, they are decremented from the partners of the synchronization.

$$\frac{id \in a(A) \qquad id' \notin a(A')}{k[(id :: go\, k'.P) \mid A]_\delta \mid k'[A']_{\delta'} \to k[A]_\delta \mid l[id' :: P \mid A']_{\delta'}}\ \textbf{Go1}$$

$$\frac{id \notin a(A) \qquad id' \notin a(A') \qquad \delta'_1 = \gamma(\delta, id)}{k[(id :: go\, k'.P) \mid A]_\delta \mid k'[A']_{\delta'} \to k[A]_\delta \mid l[id' :: P \mid A']_{\delta'_1}}\ \textbf{Go2}$$

$$\frac{\begin{array}{c} C_i \leq \Re(from\ \delta \models\ u_i[C] \mid tt\ select\ C) \quad i = 1, \ldots, n \qquad r' \text{ fresh} \\ \delta' = \delta[\prod_{i=1}^{n} u_i[C_i] \mid \delta(id)/id][\prod_{i=1}^{n} u_i[C_i \oplus \overline{C'_i}]/\mathfrak{s}][\prod_{i=1}^{n} u_i[\mathcal{Z}(C_i)]/r'] \end{array}}{k[id :: (r \leftarrow (u^1_{C_1}, \ldots, u^n_{C_n})).P \mid A]_\delta \to k[id :: P[r'/r] \mid A]_{\delta'}}\ \textbf{Bind}$$

$$\frac{\begin{array}{c} C_\mathfrak{s} = \Re(from\ \delta(\mathfrak{s}) \models\ u[C] \mid tt\ select\ C) \\ C_u = \Re(from\ \delta(id) \models\ u[C] \mid tt\ select\ C) \\ 1\rho \leq C \leq C_\mathfrak{s} \quad \wedge \quad C_m \leq C \oplus C_u \leq C_M \\ C_1 = from\ \delta(id) \models\ u[C'] \mid C''\ select\ u[(C' \oplus C) \otimes (C_M \oplus \overline{C} \oplus \overline{C_u})] \mid C'' \\ C_2 = from\ \delta(\mathfrak{s}) \models\ u[C'] \mid C''\ select\ u[C' \oplus \overline{C}] \mid C'' \\ \delta' = \delta[C_1/id][C_2/\mathfrak{s}] \end{array}}{k[(id :: \$^{C_M}_{C_m}\langle u \rangle.P \mid A]_\delta \to k[id :: P \mid A]_{\delta'}}\ \textbf{Acquire}$$

Table 4: Reduction Semantics II

Rules **Go1** and **Go2** regulates the migration of an application in two cases. In both cases, however, observing that the credit environment of the destination location k' does not change, and that, if the system is well formed, we conclude that after migration, the application has no credit on any of its resources. Rule **Go1** deals with the case when part of the application moves to k', while another part remains on the starting location k (condition $id \in a(A)$). In this case credits on k does not change. On the other hand, when the whole application moves to k' (condition $id \notin a(A)$, then id returns all its unused credits to the location k before migrating.

The bind prefix creates an aggregate of resources having some credits that are less, or equal, than the credits available on the location. After the transition, the credits that the location has on each resource are decremented of the amount of credits assigned to the aggregate.

An application id may dynamically requests new credits on its resources. This is done with the acquire prefix that allows id to specify a range of credits (C_m

and C_M). Let C_u the credit that id already owns on u (as specified in the hypothesis of **Acquire**). If the location may guarantee a credit C such that $C \oplus C_u$ is not greater than C_M, then C is summed to C_u (C_1) and subtracted from the credits of the location relative to u (C_2). Note also that, all the credits that k does not release, i.e. $C_M \oplus \overline{C} \oplus \overline{C_u}$, are added to the virtual credits of id on u.

3.4 The type system

We introduce a type system which ensures that migrating processes cannot consume more resource credits that those acquired through negotiations.

If a process asks for a resource binding it is fundamental to check consistency of the required resources: For instance, process $(r \leftarrow a_{c_a}, b_{c_b}).P$ requests the binding of a and b asking for credits c_a and c_b: We require that $c_a \leq c'_a$ and $c_b \leq c'_b$, where c'_a and c'_b are the credits defined by resource type definitions. Furthermore, P must be well-typed.

Definition 9 *A type environment, $\Gamma, \Gamma', \ldots$, is a function from a finite subset of location identifiers to location types. We adopt the usual notation of type theory and write $\Gamma = \{l_1 : L_1, ..., l_n : L_n\}$ to denote the function having $\{l_1, ..., l_n\}$ as domain and mapping each location l_i to L_i.*

Type environment aims at storing type information about resources allocated on locations. We use the notation $\Gamma(l)$ to refer the type of location l and $\Gamma(l).r$ to refer the type of the resource r on location l. For instance, given

$$\Gamma = \{l : [a : \langle\zeta\rangle, d : \langle\zeta'\rangle], \ \ ...\}$$

$\Gamma(l) = [a : \langle\zeta\rangle,\ d : \langle\zeta'\rangle]$ and $\Gamma(l).d = [\langle\zeta''\rangle]$.

We extended the above notation to name nesting by structural induction:

- $\Gamma(k).S = \Gamma(k).u$ if $S = u$
- $\Gamma(k).S = OPEN(\Gamma, k, u).S'$ if $S = u.S'$

where $OPEN(\Gamma, k, r)$ "transforms" resource types into location types. For instance, if $\Gamma(k).r = (a : T_{R_a}, b : T_{R_b})$ then the type of $r.a$ is obtained by opening the type of r on k with respect to Γ: $OPEN(\Gamma, k, r) = [a : T_{R_a}, b : T_{R_b}]$.

The set of names occurring in an environment Γ is indicated by $n(\Gamma)$ and the domain Γ is indicated by $dom(\Gamma)$. Moreover, $\Gamma,\ l : L$ is the extension of Γ with the association of l to the type L (provided that $l \notin dom(\Gamma)$) and $\Gamma,\ {}_l u : \zeta$ is the environment obtained extending in Γ the type associated to l with $u : \zeta$, assuming that $u \notin dom(\Gamma)$.

Example 4 *Let Γ be $\{l : [r : T_R, r_1 : T_{R_1}]\}$, then*

$$\Gamma,\ w : L_w = \{l : [r : T_R, r_2 : T_{R2}], w : L_w\}$$
$$\Gamma,\ {}_l r_2 : T_{R_2} = \{l : [r : R, r_2 : R_2, r_2 : R_2],\}$$

The type system is defined in Table 5 and consists of the following judgments:

- $\Gamma \models N$: the system N is well-typed with respect to Γ;
- $\Gamma \models_l id :: P$: application id is well-typed to be executed on the location l, with respect to Γ;
- $\Gamma \models_l S : T_R$: S is defined on l and has type T_R with respect to Γ.

In Table 5, $res(\delta)$ denotes the set of resources assigned by δ to applications and locations. We use $\Gamma \models M, N$ as a shorthand for $\Gamma \models M \wedge \Gamma \models N$ and similarly for the other judgments. Rule (s-null) states that the void system is well-typed

Systems

$$\text{(s-null)}\frac{}{\Gamma\models 0} \qquad \text{(s-comp)}\frac{\Gamma\models M,\ N}{\Gamma\models M|N}$$

$$\text{(s-run)}\frac{\Gamma\models l{:}L \quad \Gamma\models_l \widetilde{id{::}P} \quad res(\delta)\subseteq n(\Gamma)}{\Gamma\models l[\widetilde{id{::}P}]_\delta}$$

Applications

$$\text{(p-null)}\frac{l\in dom(\Gamma)}{\Gamma\models_l id{::}0} \qquad \text{(p-comp)}\frac{\Gamma\models_l id{::}P,\ id{::}Q \quad l\in dom(\Gamma)}{\Gamma\models_l id{::}P|Q}$$

$$\text{(p-move)}\frac{\Gamma\models_l id'{::}P}{\Gamma\models_k id{::}go\ l.P}\ id' \text{ fresh} \qquad \text{(p-bang)}\frac{\Gamma\models_l id{::}P}{\Gamma\models_l id{::}!P}$$

$$\text{(p-out)}\frac{\Gamma\models_l id{::}P,\ U{:}\alpha,\ S{:}\alpha}{\Gamma\models_l id{::}S!\langle U\rangle.P}$$

$$\text{(p-in)}\ \frac{\Gamma\models_l S{:}\alpha \quad \Gamma,\ {}_l X{:}\alpha\models_l id{::}P \quad X\notin dom(\Gamma)}{\Gamma\models_l id{::}S?(X{:}\alpha).P}$$

$$\text{(p-match}_l\text{)}\frac{\Gamma\models u{:}L,\ v{:}L \quad \Gamma\models_l id{::}Q, id{::}P}{\Gamma\models_l id{::}if\ \ u{=}v\ \ then\ \ P\ \ else\ \ Q}$$

$$\text{(p-match}_r\text{)}\frac{\Gamma\models_l u{:}T_R,\ v{:}T_R \quad \Gamma\models_l id{::}Q, id{::}P}{\Gamma\models_l id{::}if\ \ u{=}v\ \ then\ \ P\ \ else\ \ Q}$$

$$\text{(p-bind)}\frac{\forall\ i\ C_i\leq\mathcal{C}(\Gamma(l).u_i) \quad \Gamma,\ {}_l r{:}(u_1{:}\Gamma(l).u_1,\ldots,u_n{:}\Gamma(l).u_n)\models_l id{::}P}{\Gamma\models_l id{::}(r\leftarrow\widetilde{u_c}).P}$$

$$\text{(p-acq)}\frac{m\leq M \quad \Gamma\models_l id{::}P}{\Gamma\models_l id{::}\$^M_m(u).P}$$

Table 5: Type System for D^R_π

in any environment. Rule (s-comp) states that $M \mid N$ is well-typed in Γ, if both M and N are well-typed in Γ. Rule (s-run) states that a location l is well-typed provided that all applications are well-typed and can run on l according to an environment Γ which types l and each resource occurring in the allocation environment of l. According to (p-null), the void application can be executed in each location typed by Γ. Rule (p-comp) states that, if two separate parts of an application are well-typed in Γ, then their parallel composition is well-typed too. Rule (p-move) states that the continuation of a *go* prefix must be well-typed on the target location. Rule (p-bang) says that, whenever $id :: P$ is well-typed in Γ, then also the iterated application is well-typed in Γ and can be executed at l. Rule (p-out) details the typing of output prefix: if $id :: P$ can be executed at l, U has the same type of $leaf(S)$, then $id :: S!\langle U\rangle.P$ can be executed at l. Similarly, (p-in) prescribes that X must have the same type of $leaf(S)$ in Γ. Rules (p-match$_l$) and (p-match$_r$) are trivial. Rule (p-acq) states that an acquire prefix is well-typed provided that the location has enough resources.

Example 5 *Let P be the process $go\ l.\$^{1\rho}_{1\rho}(v).v?(z : INT).go\ k\ldots$ and let us consider the following system:*

$$M \stackrel{\text{def}}{=} l[\mathtt{S} :: ...]_\delta \mid k[\mathtt{C} :: P]_{\delta'}.$$

M is well-typed in any Γ such that $\Gamma \models_l v : INT, k : []$, namely, a type environment that gives the correct type to the channel and that "knows" k.

3.5 A hint on Subject Reduction and Type Safety

The main results about the type system of D^R_π are the *subject reduction* and the *type safety* propriety: Usually, the subject reduction is stated as follows:

If $\Gamma \models N$ and $N \to M$ then $\Gamma \models M$.

However, this formulation does not apply to D^R_π. Consider the system

$$N \stackrel{\text{def}}{=} l[id :: (r \leftarrow a_{c_a}, b_{c_b}).P]_\delta.$$

Then $N \to M$, where $M \stackrel{\text{def}}{=} l[id :: P]_{\delta'}$ and δ is extended to δ' with information about r as prescribed by the operational semantics. If $\Gamma \models N$, it is not possible that $\Gamma \models M$ because Γ does not contain any information about r that is bound in N. Hence, we need to define a *contextual subject reduction* in which dynamic creation of new resources is considered.

Given a system N, $\delta(N)$ denotes the set of credit environments equipping the locations appearing in N.

Theorem 1 *Contextual Subject Reduction*

(a) If $M \equiv N$ then $\Gamma \models M$ if, and only if, $\Gamma \models N$.

(b) If $\Gamma \models N$ and $N \to M$ is a reduction step that generates resources $r_1, \ldots, r_n \in res(\delta) \setminus n(\Gamma)$ allocated on $l_1, \ldots, l_n$ and typed by $T_{R_1}, \ldots, T_{R_n}$ then $\Gamma,\ {}_{l_1}r_1 : T_{R_1}, \ldots,\ {}_{l_n}r_n : T_{R_n} \models M$.

To prove type safety, we follow the same approach of [6]. We define a set of illegal actions that would generate failures. We prove that if a system is well-typed it cannot generate run-time failures. To this purpose, we extend D_{π}^{R} adding process tags that describe their knowledges on network resources. Processes become $\{id :: P\}_{\Gamma}$ where Γ is a type environment. During its evolution, $\{id :: P\}_{\Gamma}$ increases its knowledges and its private environment Γ is extended correspondingly.

An example of behaviour is given below:

$$\frac{\neg(C_2 \leq C_1) \ \vee \ \neg(C_1 \leq \mathcal{C}(\Gamma(l).u)) \ \vee \ u \notin \Gamma(l)}{l[\{id::\$_{C_1}^{C_2}(u).P\}_{\Gamma}|A]_{\delta} \stackrel{err}{\longmapsto}}$$

In the case of acquire, an error occurs ($\stackrel{err}{\longmapsto}$) if the resource is not known by the process or the credits are non-correctly required:

Theorem 2 *(Type Safety) If* $\Gamma \models \ M$ *then* $\neg(M \stackrel{err}{\longmapsto})$

References

[1] M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *CONCUR*, volume LNCS 2154, pages 102–120. Springer, 2001.

[2] L. Cardelli and G. Ghelli. A query language for semistructured data based on the ambient logic. In D. Sands, editor, *Programming Languages and Systems, ESOP2001*, volume 2028 of *LNCS*, pages 1–22. Springer, 2001. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS2001.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, Berlin, Germany, Mar. 1998. Full version TCS 240(1), 2000.

[4] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, June 2000.

[5] G. Ghelli, G. Castagna, and F. Zappa-Nardelli. Typing mobility in the seal calculus. In *CONCUR*, volume LNCS 2154, pages 82–101. Springer, 2001.

[6] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier Science Publishers, 1998. Full version as CogSci Report 2/98, University of Sussex, Brighton.

[7] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *27th International Colloquium on Automata, Languages and Programming (ICALP '2000)*, July 2000. A longer version

appeared as Computer Science Technical Report 2000:03, School of Cognitive and Computing Sciences, University of Sussex.

[8] R. Milner. *Communication and Concurrency.* Printice Hall, 1989.

[9] R. Milner, J. Parrow, and D. Walker. Modal Logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.

[10] L. Moreau and C. Queinnec. Design and semantics of quantum: a language to control resource consumption in distributed computing, 1997.