プログラミング 8−8
(1996. 8. 28)

Nepi^2: π 計算に基づくネットワーク・プログラミングのための 2 レベル計算体系

堀田 英一 (horita@progn.kecl.ntt.jp), 真野 健 (mano@progn.kecl.ntt.jp)
NTT コミュニケーション研究所

概要. π 計算に基づき, 分散処理記述言語 Nepi^2 を提案する. Nepi^2 の言語は 2 つのレベルで定義される. 第一レベルの言語は, 局所的なシステムを走行可能なプロセスとして表現するために用いられるプログラミング言語である. 第二レベルの言語は大域的なシステムの記述法を与えるものであり, このようなシステムの解析のために用いられる. 我々は π 計算に基づいて, Nepi^2 の操作的意味論を定義し, 最後に Unix の標準的な機能を用いて, 試験的な処理系を与える.

Nepi^2: a Two-Level Calculus for Network Programming Based on the π-Calculus

Eiichi Horita (horita@progn.kecl.ntt.jp) and Ken Mano (mano@progn.kecl.ntt.jp)
NTT Communication Science Laboratories

Abstract. We propose a calculus Nepi^2 for network programming, on the basis of the π-calculus. The language of Nepi^2 is defined in two levels: the first-level language, which is a programming language, is used for representing *local systems* as processes; the second-level language is used for describing and analyzing *global systems* consisting of processes represented in the first-level language. We define the operational semantics for Nepi^2 on the basis of the π-calculus, and give an experimental implementation on a network using standard facilities of Unix.

1 Introduction

We propose a two-level calculus Nepi^2 for network programming, on the basis of the π-calculus [11, 10]; the two-levels treats *local* (or *intra-process*) concurrency and *global* (or *inter-process*) concurrency, respectively. We discuss the operational semantics and an experimental implementation of this calculus.

Nepi^2 evolved from Nepi [5, 6, 7], which is a network programming language based on the π-calculus but does not distinguish between local and global concurrency. In Nepi, any two agents can communicate wtih each other only via a global communication manager (GCM); this method of inter-agent communication has two problems in performance and scalability: **(i)** for two agents residing within a local system, this method entails unnecessary global interactions; **(ii)** a substantial part of the load for inter-agent communication concentrates on the GCM, which can be a bottleneck when the overall system scales up. A major motive for introducing two levels in Nepi^2 is our desire to alleviate these problems.

We distinguish between local and global concurrency in Nepi^2 mainly for performance reasons. Form the viewpoint of programming and semantics, on the other hand, we try to keep the difference between the treatments of local and global concurrency as little as possible.

In Sect. 2, the language of Nepi^2 is defined in two levels: the first-level language, which is a programming language, defines the syntactic category of *processes* and is used for representing *local systems* as processes; the second-level language, which is defined on top of the first-level one, defines the syntactic category of *global systems*, and is used for analyzing global systems consisting of processes described in the first-level language. We define, in Sect. 3, the operational semantics of Nepi^2 via a Plotkin-style transition system [14]. Then we describe a distributed implementation of Nepi^2 in Sect. 4. First we introduce another calculus called the $\nu\pi$-calculus, which we believe to be equivalent to the π-calculus at an appropriate level of abstraction; the $\nu\pi$-calculus is more suited toward distributed implementation. Next, a programming system for Nepi^2 is provided by implementing the $\nu\pi$-calculus in Unix using standard facilities such as sockets and a thread library.

Pierce and Turner developed *Pict*, a language based on the π-calculus, and provided a uniprocessor implementation of it [13] ([15] also discusses implementation methods of a similar language). As pointed out in [13], multi-processor implementations of any language based on the π-calculus remain for future research. Our development of Nepi^2 is an effort in this direction. With Nepi^2, two programs invoked as different OS processes residing in different machines in a network can communicate with each other. This is why we call Nepi^2 a network programming language.

2 The Language

The underlying framework of Nepi is the π-calculus, which is an algebraic calculus rather than a programming language. To develop a programming system, we need to specify a concrete syntax, which we give along the lines of the S-expression syntax of Lisp. We define the language Nepi[2] in two levels as explained in the Introduction.

2.1 The First-Level Language

For the first-level language, we provide two syntaxes: an *abstract* one and a *concrete* one. The abstract syntax is used for defining the operational semantics of this language in Sect. 3, while the concrete syntax is used for writing programs and their compilation. Programs in the concrete syntax are transformed in a very straightforward way to corresponding expressions in the abstract syntax.

Preliminaries. The phrase "let $(x \in)\ X$ be ..." introduces a set X with a variable x ranging over X. The set of natural numbers is denoted by ω. For $n, m \in \omega$, let $[n..m] = \{i \in \omega |\ n \leq i \leq m\}$. The power set of a set X is denoted by $\wp(X)$. For a set A, we write A^* to denote the set of finite sequences of elements of A. We denote the syntactic identity between expressions by $\equiv$. The syntactic category *Id* of *identifiers* is defined as in C. In Nepi, there are four basic types: `int` (of *integers*), `str` (of *character strings*), **chan** (of *communication channels*), and **proc** (of *processes*).

Let $\mathcal{V}_{\text{int}}$ $\mathcal{V}_{\text{str}}$, $(\xi \in)\ \mathcal{V}_{\text{chan}}$, and $(A \in)\ \mathcal{V}_{\text{proc}}$ be the syntactic categories of variables of types `int`, `str`, **chan**, and **proc**, respectively. Each of these is defined as a subset of *Id*, by specifying the starting letter of the identifiers in it: elements of $\mathcal{V}_{\text{int}}$ (resp. $\mathcal{V}_{\text{str}}$, $\mathcal{V}_{\text{chan}}$ and $\mathcal{V}_{\text{pro}}$) are identifiers starting with `i` (resp. `w` , `c` and `p`). Let $(z \in)\ \mathcal{V}_{\text{data}} = \mathcal{V}_{\text{int}} \cup \mathcal{V}_{\text{str}}$ and $(x \in)\ \mathcal{V}_{\text{val}} = \mathcal{V}_{\text{data}} \cup \mathcal{V}_{\text{chan}}$.

Value Expressions. We call `int` and `str` *built-in data types* of Nepi, and put $(t \in)\ DT = \{\texttt{int}, \texttt{str}\}$.[1] A set *DO* of *built-in operators* on these types is also provided, together with a typing function $type : DO \to (DT^* \times DT)$. For each $t \in DT$, let $\mathcal{L}_t$ be the set of terms of type t constructed from the signature (DT, DO). We use $\mathbf{n}$ as a metavariable ranging over $\mathcal{L}_{\text{int}}$. Let $(d \in)\ \mathcal{L}_{\text{data}} = \bigcup\{\mathcal{L}_t |\ t \in DT\}$. We assume that for each $t \in DT$, a semantic domain $\mathbf{DV}_t$ is given as a subset of $\mathcal{L}_t$, and that for each operator $F \in DO$ with $type(F) = ((t_1, \ldots, t_r), \tilde{t})$, a function from $\mathbf{DV}_{t_1} \times \cdots \times \mathbf{DV}_{t_r}$ to $\mathbf{DV}_{\tilde{t}}$ is given as the predefined *interpretation* of F. We put $(\mathbf{d} \in)\ \mathbf{DV} = \bigcup\{\mathbf{DV}_t |\ t \in DT\}$. The *evaluation* $[\![d]\!] \in \mathbf{DV}$ of each *closed* data expression d is determined by the interpretations of the operators in DO.[2]

From $\mathcal{L}_{\text{str}}$, the syntactic category $(u \in)\ \mathcal{L}_{\text{chan}}$ of *channel expressions* is defined by $\mathcal{L}_{\text{chan}} = \mathcal{V}_{\text{chan}} \cup \{(\text{ch}\ d) |\ d \in \mathcal{L}_{\text{str}}\}$. Finally, we define the syntactic category $(v \in)\ \mathcal{L}_{\text{val}}$ of *value expressions* by $\mathcal{L}_{\text{val}} = \mathcal{L}_{\text{data}} \cup \mathcal{L}_{\text{chan}}$.

Abstract Syntax for Process Expressions. Process expressions of Nepi are constructed as algebraic terms from the following primitive constructs plus λ-notation (for function abstraction) and μ-notation (for recursion): **(1)** 'δ' for *inaction*; **(2)** '$\|$' for *parallel composition*; **(3)** 'ν' for generation of a fresh channel; **(4)** '!' for *local* (or *intra-process*) *output* and '!!' for *global* (or *inter-process*) *output*; **(5)** '?' for *local* (or *intra-process*) *input* and '??' for *global* (or *inter-process*) *input*; **(6)** '+' for *alternative choice*; **(7)** a set $(\mathbf{o} \in)$ **OP** of *output ports*; **(8)** a set $(\mathbf{i} \in)$ **IP** of *input ports*; and **(9)** '`if`' for *conditionals*. Formally, the syntactic category $(P, Q \in)\ \mathcal{L}_{\text{proc}}$ of *process expressions* is defined simultaneously with the three categories $(G \in)\ \mathcal{L}_{\text{pref}}$ (of *prefixed process expressions*), $(\Delta \in)\ \mathcal{DC}$ (of *declaration components*) and $(D \in)\ \mathcal{D}$ (of *declarations*), by the following grammar:

$$\begin{aligned} P ::= &\ \delta \mid (\|\ P_1\ P_2) \mid (\nu\ \xi\ P) && (1)\\ &\mid G \mid (+\ G_1\ \cdots\ G_n)\\ &\mid (\textbf{if}\ \mathbf{n}\ P_1\ P_2)\\ &\mid (\mathbf{o}\ d\ P) \mid (\mathbf{i}\ (\lambda\ z\ P))\\ &\mid ((\mu\ D\ A)\ \ v_1\ \cdots\ v_n),\\ G ::= &\ (!\ u\ v\ P) \mid (!!\ u\ v\ P) && (2)\\ &\mid (?\ u\ (\lambda\ x\ P)) \mid (??\ u\ (\lambda\ x\ P)),\\ \Delta ::= &\ (A\ (x_1\ \cdots\ x_n)\ P), && (3)\\ D ::= &\ (\Delta_1\ \cdots\ \Delta_k), && (4) \end{aligned}$$

where n and k range over ω.

An element $(A\ (x_1\ \cdots\ x_n)\ P)$ of $\mathcal{DC}$ declares the process A to have formal parameters $x_1\ \cdots\ x_n$ and body P. For the last term $((\mu\ D\ A)\ v_1\ \cdots\ v_n)$ of (1) with

$$\begin{cases} D = (\Delta_1\ \cdots\ \Delta_k), \\ \Delta_i = (A_i\ (x_1\ \ldots\ x_{n(i)})\ P_i) \quad (i \in [1..k]), \end{cases} \qquad (5)$$

there must exist $i \in [1..n]$ such that $A = A_i$ and $n = n(i)$. Regarding (4), we impose the constraint that for D of the form (5), $A_1, \ldots, A_k$ must be all distinct and any element of $\mathcal{V}_{\text{proc}}$ appearing in P_i must belong to $\{A_1, \ldots, A_k\}$ $(i \in [1..k])$.

For $P \in \mathcal{L}_{\text{proc}}$, let $fv(P)$ denote the set of *free variables* contained in P. Let

$$\mathcal{L}^{\emptyset}_{\text{proc}} = \{P \in \mathcal{L}_{\text{proc}} |\ fv(P) = \emptyset\}.$$

[1] We can introduce new data types and new operators on data by giving their definitions in C. For the syntax for doing this, see [5].

[2] An expression is said to be *closed* when it contains no free variables.

Concrete Syntax for Programs. Besides the abstract syntax above, we introduce a *concrete* syntax for convenience in writing Nepi programs and in their compilation. The concrete syntax is the same as the abstract syntax, except that we use the `labels` construct borrowed from Lisp to declare recursive processes in the concrete syntax instead of the μ-construct used in the abstract syntax.

More precisely, the set $\widehat{\mathcal{L}}_{\text{proc}}$ (of process expression) and the one $\widehat{D}$ (of declarations) in the concrete syntax are defined in the same way as in (1)–(4), except that we replace the term $((\mu\ D\ A)\ v_1\ \cdots\ v_n)$ in (1) by $(A\ v_1\ \cdots\ v_n)$. Each concrete program of Nepi consists of zero or more declarations of processes and a *main process expression.* From $(D\ \in)\ \widehat{\mathcal{D}}$ and $(P, Q\ \in)\ \widehat{\mathcal{L}}_{\text{proc}}$, we define the syntactic category $(R \in)\ \widehat{\mathcal{L}}_{\text{prog}}$ of *concrete programs* in Nepi[2] by

$$R ::= (\texttt{labels}\ D\ P),$$

where we use the *labels* construct for defining mutually recursive processes. For each concrete program $(\texttt{labels}\ D\ P) \in \widehat{\mathcal{L}}_{\text{prog}}$, we define its abstract form $\mathcal{A}((\texttt{labels}\ D\ P)) \in \mathcal{L}_{\text{proc}}$ as follows: For D having the form (5), we obtain $\mathcal{A}((\texttt{labels}\ D\ P))$ from P by replacing each occurrence of the form $(A_i\ v_1\ \cdots\ v_{n(i)})$ in P by $((\mu\ D\ A_i)\ v_1\ \cdots\ v_{n(i)})$ $(i \in [1..k])$. We note that the application of $\mathcal{A}$ to concrete programs is analogous to *flattening* of LOTOS [8].

2.2 The Second-Level Language

We use the symbol Π for the *global parallel composition.* Let S be a variable ranging over (global) *system expressions* which are combinations of (local) programs running in parallel. From $(P\ \in)\ \mathcal{L}_{\text{proc}}$, the syntactic category $\mathcal{L}_{\text{sys}}$ of *system expressions* is defined by the following grammar:

$$S ::= P \mid (\Pi\ S_1\ S_2) \mid (\nu\ \xi\ S). \qquad (6)$$

By this grammar we have

$$\mathcal{L}_{\text{proc}} \subseteq \mathcal{L}_{\text{sys}}, \qquad (7)$$

and we may consider the sort of process expressions as a *subsort* of that of system expressions (where we use the concept of subsort in the sense of [4]).

3 The Operational Semantics Based on the π-Calculus

In this section, we define the operational semantics of Nepi[2] via a Plotkin-style transition system [14]. As a preliminary to the definition, we first introduce the notion of *structural congruence.*

3.1 Structural Congruence

Notation 1 We denote α-convertibility between process expressions by $\equiv_\alpha$.

For $n \geq 1$ and $P_1, \ldots, P_n \in \tilde{\mathcal{L}}_{\text{proc}}$, we use the notation $(\|^*\ P_1, \ldots, P_n)$ as a shorthand for a process expression; what this shorthand stands for is inductively defined as follows: **(i)** For $n = 1$, the notation $(\|^*\ P_1)$ stands for P_1. **(ii)** For $n > 1$, the notation $(\|^*\ P_1\ \ldots\ P_n)$ stands for $(\|\ P_1\ \tilde{P})$, where $\tilde{P}$ is what the notation $(\|^*\ P_2\ \cdots\ P_n)$ stands for. ∎

Following [10, Sect. 2.3], we define the *structural congruence* $\hat{\equiv}$ over $\mathcal{L}_{\text{sys}}$, which contains $\mathcal{L}_{\text{proc}}$, as the smallest congruence relation satisfying the following laws.[3] First, two system expressions that are α-convertible with each other are congruent:

SC1²: $S_1 \equiv_\alpha S_2 \Rightarrow S_1 \hat{\equiv} S_2$.

We have the following Abelian semigroup laws for $\|$ and Π:

SC2: $(\|\ P_1\ (\|\ P_2\ P_3)) \hat{\equiv} (\|\ (\|\ P_1\ P_2)\ P_3)$.

SC2²: $(\Pi\ S_1\ (\Pi\ S_2\ S_3)) \hat{\equiv} (\Pi\ (\Pi\ S_1\ S_2)\ S_3)$.

SC3: $(\|\ P_1\ P_2) \hat{\equiv} (\|\ P_2\ P_1)$.

SC3²: $(\Pi\ S_1\ S_2) \hat{\equiv} (\Pi\ S_2\ S_1)$.

For process/system expressions of the form $(\nu\ \cdots)$, we have the following four laws:

SC4²: $(\nu\ \langle\xi,\zeta\rangle\ S) \hat{\equiv} (\nu\ \langle\zeta,\xi\rangle\ S)$.

SC5²: $\xi \notin fv(S) \Rightarrow (\nu\ \xi\ S) \hat{\equiv} S$.

SC6: $\xi \notin fv(P_2) \Rightarrow$
$(\|\ (\nu\ \xi\ P_1)\ P_2) \hat{\equiv} (\nu\ \xi\ (\|\ P_1\ P_2))$.

SC6²: $\xi \notin fv(S_2) \Rightarrow$
$(\Pi\ (\nu\ \xi\ S_1)\ S_2) \hat{\equiv} (\nu\ \xi\ (\Pi\ S_1\ S_2))$.

3.2 Transition Rules

For channel expressions u_1 and u_2, we write $u_1 \cong u_2$ to mean that both u_1 and u_2 are channel variables with $u_1 \equiv u_2$ or that both u_1 and u_2 are closed terms with $[\![u_1]\!] = [\![u_2]\!]$. For expressions $P, v_1, \ldots, v_r$, and distinct variables $x_1, \ldots, x_r$, we denote by $P[\langle v_1, \ldots, v_r\rangle / \langle x_1, \ldots, x_r\rangle]$ the result of the simultaneous substitution of $v_1, \ldots, v_r$ for $x_1, \ldots, x_r$ in P. We define the set $\mathbf{E}$ of *events* by $(e\ \in)\ \mathbf{E} = (\mathbf{OP} \times \mathbf{DV}) \cup (\mathbf{IP} \times \mathbf{DV})$. Let $(a\ \in)\ \mathbf{E}_\tau = \mathbf{E} \cup \{\tau\}$, where τ is a symbol representing an *internal* (or *unobservable*) action.

[3] Below, we tag rules for system expressions with a superscript ², as in SC1² below. We remark that SC1² implies that $\forall P_1, P_2 \in \mathcal{L}_{\text{proc}}[\ P_1 \equiv_\alpha P_1 \Rightarrow P_1 \hat{\equiv} P_2\ ]$, since each process expression is also a system expression by (7). The rules SC4² and SC5² below have similar implications obtained by replacing the variables S_1, S_2 ranging over $\mathcal{L}_{\text{sys}}$ with P_1, P_2 ranging over $\mathcal{L}_{\text{proc}}$.

The *transition relations* $\xrightarrow{a}$ ($a \in \mathbf{E}_\tau$) are defined as the smallest binary relations on $\mathcal{L}_{\mathrm{sys}}$ satisfying the following laws COM–STR^2.

COM: If $u_1 \cong u_2$ and $type(v) = type(x)$, then

$$(\| \ (+ \ \cdots \ (! \ u_1 \ v \ P) \ \cdots) \\ (+ \ \cdots \ (? \ u_2 \ (\lambda \ x \ Q)) \ \cdots)) \\ \xrightarrow{\tau} (\| \ P \ Q[v/x]).$$

COM′: If $u_1 \cong u_2$ and $type(v) = type(x)$, then

$$(\| \ (+ \ \cdots \ (!! \ u_1 \ v \ P) \ \cdots) \\ (+ \ \cdots \ (?? \ u_2 \ (\lambda \ x \ Q)) \ \cdots)) \\ \xrightarrow{\tau} (\| \ P \ Q[v/x]).$$

COM^2: If $u_1 \cong u_2$ and $type(v) = type(x)$, then

$$(\Pi \ (\|^* \ (+ \ \cdots \ (!! \ u_1 \ v \ P) \ \cdots) \ \cdots) \\ (\|^* \ (+ \ \cdots \ (?? \ u_2 \ (\lambda \ x \ Q)) \cdots) \ \cdots)) \\ \xrightarrow{\tau} (\Pi \ (\|^* \ P \ \cdots) \ (\|^* \ Q[v/x] \ \cdots)).$$

OUT: For each $\mathbf{o} \in \mathbf{OP}$, we have

$$(\mathbf{o} \ d \ P) \xrightarrow{(\mathbf{o}, [\![d]\!])} P.$$

IN: For each $\mathbf{i} \in \mathbf{IP}$, we have

$$(\mathbf{i} \ (\lambda \ z \ P)) \xrightarrow{(\mathbf{i}, \mathbf{d})} P[\mathbf{d}/z].$$

CND: If $[\![\mathbf{n}]\!] \neq 0$, then $(\mathtt{if} \ \mathbf{n} \ P_1 \ P_2) \xrightarrow{\tau} P_1$. Otherwise, $(\mathtt{if} \ \mathbf{n} \ P_1 \ P_2) \xrightarrow{\tau} P_2$.

REC: If $(A \ (x_1 \ \cdots \ x_r) \ P) \in D$, then

$$((\mu \ D \ A) \ v_1 \ \cdots \ v_r) \\ \xrightarrow{\tau} P[\langle v_1, \ldots, v_r \rangle / \langle x_1, \ldots, x_r \rangle].$$

PAR: $$\frac{P_1 \xrightarrow{a} P_1'}{(\| \ P_1 \ P_2) \xrightarrow{a} (\| \ P_1' \ P_2)}.$$

PAR^2: $$\frac{S_1 \xrightarrow{a} S_1'}{(\Pi \ S_1 \ S_2) \xrightarrow{a} (\Pi \ S_1' \ S_2)}.$$

RES^2: $$\frac{S \xrightarrow{a} S'}{(\nu \ \xi \ S) \xrightarrow{a} (\nu \ \xi \ S')}.$$

STR^2: $$\frac{S_1 \stackrel{.}{\equiv} S_1', \ \ S_1' \xrightarrow{a} S_2', \ \ S_2' \stackrel{.}{\equiv} S_2}{S_1 \xrightarrow{a} S_2}.$$

From $\xrightarrow{a}$ ($a \in \mathbf{E}_\tau$), we define so-called *weak* transition relations $\stackrel{s}{\Longrightarrow}$ ($s \in (\mathbf{E}_\tau)^*$), and thereby the concept of *weak bisimulation* as in [9].

The rule STR^2 is useful in simplifying the definition of the transition relation, but imposes difficulty in distributed implementation of the π-calculus. In the next section, we introduce another calculus named the $\nu\pi$-calculus, which is more suited to distributed implementation, and which we believe to be equivalent to the π-calculus at an appropriate level of abstraction.

4 The Implementation on a Network

In this section, we describe a distributed implementation of Nepi^2. First, in Sect. 4.1, we introduce another calculus called the $\nu\pi$-calculus. Next, in Sect. 4.2, a programming system for Nepi^2 is provided by implementing the $\nu\pi$-calculus.

4.1 The $\nu\pi$-Calculus: an Implementation Oriented Calculus

The transition relations of the $\nu\pi$-calculus are defined as binary relations between *system configurations*, which are pairs of a process expression and an integer (for a similar treatment of mobile processes in the framework *chemical abstract machines*, see [1, Sect. 5.2]). We define the transition relations between system configurations as in Sect. 3, except that we replace the rules RES^2 and STR^2 in Sect. 3 by the rules RES$_\nu$ and STR$^2_\nu$ given below.

To give the rule RES$_\nu$, we need to introduce distinct channel constants $\gamma_0, \gamma_1, \gamma_2, \ldots$ not appearing in $\mathcal{L}_{\mathrm{proc}}$. Let $\tilde{\mathcal{L}}_{\mathrm{proc}}$ be the set of process expressions constructed in the same way as in Sect. 2 except that the symbols γ_m may be used as channel constants ($m \in \omega$). Clearly, we have $\mathcal{L}_{\mathrm{proc}} \subseteq \tilde{\mathcal{L}}_{\mathrm{proc}}$. Let $\tilde{\mathcal{L}}^{\emptyset}_{\mathrm{proc}} = \{P \in \tilde{\mathcal{L}}_{\mathrm{proc}} | \ fv(P) = \emptyset\}$. A *system configuration* is formally defined to be a pair $\langle m, P \rangle \in \omega \times \tilde{\mathcal{L}}^{\emptyset}_{\mathrm{proc}}$, where m is used as the index of the next fresh channel. The rule RES$_\nu$ is given in terms of the channel constants γ_i as follows:

RES$_\nu$: $\langle m, (\nu \ \xi \ P) \rangle \xrightarrow{\tau} \langle m+1, P[\gamma_m/\xi] \rangle$.

To formulate the rule STR$^2_\nu$, we define a relation $\equiv_\nu$ to be the equivalence relation on $\tilde{\mathcal{L}}_{\mathrm{sys}}$ induced by the Abelian semigroup laws for $\|$ and Π. In terms of $\equiv_\nu$, the rule STR$^2_\nu$ is given as follows:

STR$^2_\nu$: $$\frac{S_1 \equiv_\nu S_2, \ \ \langle m, S_1 \rangle \xrightarrow{a} \langle \ell, S_1' \rangle}{\langle m, S_2 \rangle \xrightarrow{a} \langle \ell, S_1' \rangle}.$$

The idea underlying this rule is that two system expressions that consist of the same parallel components (but possibly differ in their textual representations) should be able to make the same transitions. We employ STR$^2_\nu$ only for convenience in formulating the transition rules of the $\pi\nu$-calculus. Indeed, without using STR$^2_\nu$, we can define essentially the same transition system, by extending the set of actions as in [9].

We believe that each system expression S in the π-calculus and the system configuration $\langle m, S \rangle$ in the $\nu\pi$-calculus are bisimilar in the sense of CCS [9, Sect. 5.1]. This property is formally stated by the next claim.

Claim 1 *There is a relation* $\approx \subseteq \mathcal{L}^{\emptyset}_{\mathrm{sys}} \times (\omega \times \tilde{\mathcal{L}}^{\emptyset}_{\mathrm{sys}})$ *satisfying the following two clauses (i), (ii).*

(i) $\forall S \in \mathcal{L}^{\emptyset}_{\mathrm{sys}}, \forall m \in \omega[\ S \approx \langle m, S \rangle \ ]$.

(ii) *For every* $S_1 \in \mathcal{L}^{\emptyset}_{\text{sys}}$ *and* $\langle m, S_2 \rangle \in \omega \times \tilde{\mathcal{L}}^{\emptyset}_{\text{sys}}$ *such that* $S_1 \approx \langle m, S_2 \rangle$, *the following two properties (8) and (9) hold for every* $s \in (\mathbf{E}_\tau)^*$:

$$\forall S_1' [\, S_1 \overset{s}{\Longrightarrow} S_1' \;\Rightarrow\; \exists \langle \ell, S_2' \rangle [\, \langle m, S_2 \rangle \overset{s}{\Longrightarrow} \langle \ell, S_2' \rangle \wedge S_1' \approx \langle \ell, S_2' \rangle \,]]. \tag{8}$$

$$\forall \langle \ell, S_2' \rangle [\, \langle m, S_2 \rangle \overset{s}{\Longrightarrow} \langle \ell, S_2' \rangle \;\Rightarrow\; \exists S_1' [\, S_1 \overset{s}{\Longrightarrow} S_1' \wedge S_1' \approx \langle \ell, S_2' \rangle \,]]. \tag{9}$$ ∎

We are working on the proof of this claim, expecting that this can be proved along the lines of the proof of [5, 6, 7, Theorem 1], a similar claim for Nepi from which Nepi² evolved (see [5] for a detailed proof of this theorem). Assuming that this claim holds, we developed a programming system for Nepi² by implementing the $\nu\pi$-calculus as described below.

4.2 The Two-Level Implementation Based on the $\nu\pi$-Calculus

On the basis of the $\nu\pi$-calculus, we developed an experimental two-level implementation of Nepi² on a network, using standard facilities of Unix. Figure 1 illustrates the implementation, where we use essentially the same method as that of [2] to implement a rendezvous-type inter-agent communication, also employing the concept of a *tuple space* of [3].

Units of concurrent execution corresponding to process expressions of Nepi² are called *agents*. A prefixed process expression G is called *local* (resp. *global*) when it is of the form $(!\ u\ v\ P')$ or $(?\ u\ (\lambda\ x\ P'))$ (resp. $(!!\ u\ v\ P'')$ or $(??\ u\ (\lambda\ x\ P''))$). For $G \in \mathcal{L}_{\text{pref}}$, we define *Type*($G$) as follows:

$$\mathit{Type}((!\ u\ v\ P')) = (!u, \mathit{type}(v), v),$$
$$\mathit{Type}((?\ u\ (\lambda\ x\ P'))) = (?u, \mathit{type}(x)),$$
$$\mathit{Type}((!!\ u\ v\ P')) = (!!u, \mathit{type}(v), v),$$
$$\mathit{Type}((??\ u\ (\lambda\ x\ P'))) = (??u, \mathit{type}(x)),$$

where $\mathit{type}(v)$ (resp. $\mathit{type}(x)$) is the type of the value expression v (resp. of the value variable x).

In order to manage the request of an agent P to execute the composition $(+\ G_1\ \cdots\ G_n)$, we have to distinguish three cases: **(i)** When each G_i is a local prefixed process expression, P submits the communication request $\langle \mathit{Type}(G_1), \ldots, \mathit{Type}(G_n) \rangle$ to the relevant *local communication manager* (LCM) and asks the LCM to decide whether there is any matching communication request. **(ii)** When each G_i is a global prefixed process expression, P submits the communication request to the relevant LCM, and the LCM just forwards the agent's request to the *global communication manager* (GCM) and asks the GCM to decide whether there is any matching communication request. **(iii)** Otherwise, P first submits the communication request to the relevant LCM, and then the LCM forwards the *global part* of the request to the GCM; the LCM and the GCM cooperate following a certain protocol (the *LCM-GCM protocol*) to achieve the communication requested by the agent. In the protocol, either of the LCM or the GCM needs to have precedence over the other to avoid the possibility of deadlock (here we omit describing the protocol for lack of space).

```
 1.(labels
 2. ((p_main ()
 3.   (+
 4.    (?? c0
 5.     (λ ξ
 6.      (‖ (p_main) (p_sub ξ)) ))
 7.    (? c1
 8.     (λ x
 9.      ⟨Handle the return code x⟩
10.      (p_main) ))))
11.  (p_sub (ξ)
12.   ⟨Manage the session with the client⟩
13.   (! c1 ⟨return code⟩ δ) ))
14. (p_main) )
```

Figure 2: A Concurrent Server

5 Example Program

Figure 5 gives an outline of a concurrent server of a client-server system, where pseudo-statements are surrounded by ⟨···⟩ . The main server (p_main) creates a subagent (p_sub) for each service request from a client. The subagent manages the session with the relevant client and sends a return code to p_main when the session ends. In this program, we use alternative choice between a *global input* (on line 4) and a *local input* (on line 7).

6 Concluding Remarks

We conclude this paper with several remarks on related work and on topics for future work.

In [12], a two-level calculus M-LOTOS based on LOTOS [8] and the π-calculus [11] is proposed. This calculus is for specification rather than for programming, and is much more complex than Nepi² partly because of the complexity of the base language LOTOS. The idea underlying the *multiple tuple spaces* of [3] is similar to the one underlying the design of Nepi².

The proof of Claim 1 is to be given, and we expect that there is no essential difficulty in achieving this along the lines of [5, 6, 7]. The correctness of the LCM-GCM protocol mentioned in Sect. 4.2 is also to be proved. It also remains for future work to support structured data for communication in Nepi², as mentioned in [6].

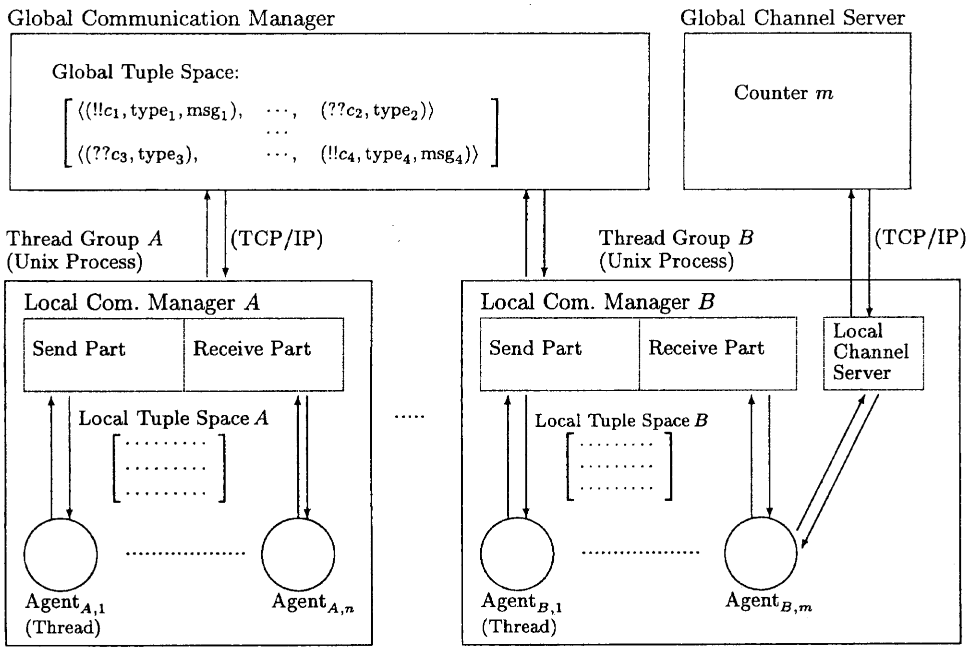

Figure 1: The Two-Level Implementation of Nepi[2]

References

[1] G. Berry and G. Boudol, The chemical abstract machine, *Theoretical Computer Science, Vol. 96*, 217–248, 1992.

[2] Luca Cardelli, An implementation model of rendezvous communication, in Lecture Notes in Computer Science, Vol. 197, pp. 449–457, 1984.

[3] D. Gelernter, Multiple tuple spaces in Linda, in *Proceedings of the PARLE'89 Conference*, Lecture Notes in Computer Science, Vol. 366, Springer, pp. 20–27, 1989.

[4] J.A. Goguen and G. Malcolm: *Algebraic Semantics of Imperative Programs*, MIT Press, 1996.

[5] E. Horita and K. Mano, *Nepi: A Network Programming Language Based on the π-Calculus*, ECL Technical Report, Vol. 11933, NTT Communication Science Laboratories, 1995.

[6] E. Horita and K. Mano: Nepi: a network programming language based on the π-calculus, *IPSJ SIG Notes, 95-PRO-2*, pp. 161–168, 1995.

[7] E. Horita and K. Mano: Nepi: a network programming language based on the π-calculus, in *Proceedings of the 1st International Conference on Coordination Models, Languages and Applications 1996*, Lecture Notes in Computer Science, Vol. 1061, Springer, pp. 424–427, 1996.

[8] ISO, *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, International Standard ISO 8807, ISO, 1989.

[9] R. Milner, *Communication and Concurrency*, Prentice Hall International, 1989.

[10] R. Milner, *The Polyadic π-Calculus: A Tutorial*, Technical Report ECS-LFCS-91-180, LFCS, Department of Computer Science, Univ. of Edinburgh, 1991.

[11] R. Milner, J. Parrow, and D. Walker, A calculus of mobile processes, I and II, *Information and Computation, Vol. 100*, pp. 1–40 and pp. 41–77, 1992.

[12] E. Najm, J.-B. Stefani, and A. Février, Mobile LOTOS, in *Working Draft on Enhancement to LOTOS* (ISO/IEC JTC1/SC21/WG1 N1349), ISO, 1994.

[13] B. C. Pierce and D. N. Turner, *Concurrent objects in a process calculus*, in Proceedings of International Workshop TPPP'94, in *Proceedings of Seminar on Concurrency*, Lecture Notes in Computer Science, Vol. 907, pp. 187–215, Springer, 1994.

[14] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Report DAIMI FN-19, Comp. Sci. Dept., Aarhus Univ., 1981.

[15] K. Takeuchi, K. Honda, and M. Kubo, An interaction-based language and its typing system, in *Proceedings of PARLE'94: Parallel Architectures and Language Europe*, Lecture Notes in Computer Science, Vol. 817, pp. 398–413.