A QoS-aware Process Calculus*

Rocco De Nicola\textsuperscript{1}, Gianluigi Ferrari\textsuperscript{2}, Ugo Montanari\textsuperscript{2}, Rosario Pugliese\textsuperscript{1}, and Emilio Tuosto\textsuperscript{2}

\textsuperscript{1} Dipartimento di Sistemi e Informatica, Università di Firenze, Via C. Lombroso 6/17, 50134 Firenze – Italy
email: \{denicola,pugliese\}@dsi.unifi.it
\textsuperscript{2} Dipartimento di Informatica, Largo Pontecorvo 1, 56127 Pisa – Italy
email: \{giangi,ugo,etuosto\}@di.unipi.it

Abstract. The definition of suitable abstractions and models for identifying, understanding and managing Quality of Service (QoS) constraints is a challenging issue of the Service Oriented Computing paradigm. In this paper we introduce a process calculus where QoS attributes are first class objects. We identify a minimal set of primitives that allow capturing in an abstract way the ability to control and coordinate services in presence of QoS constraints.

1 Introduction

Service Oriented Computing (SOC) \cite{13} has been proposed as an evolutionary paradigm to build wide area distributed systems and applications. In this paradigm, services are the basic building blocks of applications. Services can be thought of as heterogeneous software components which encapsulate resources and deliver functionalities. Services can be dynamically composed to provide new services, and their interaction is governed in accordance with programmable coordination policies. Examples of SOC architectures are provided by WEB services and GRID services.

The SOC paradigm has to face several challenges like service composition and adaptation, negotiation and agreement, monitoring and security. A key issue of the paradigm is that services must be delivered in accordance with contracts that specify both client requirements and service properties. These contracts are usually called Service Level Agreements (SLA). SLA contracts put special emphasis on Quality of Service (QoS) described as a set of non functional properties concerning issues like response time, availability, security, and so on.

The actual metric used for evaluating QoS parameters is heavily dependent on the chosen level of the abstraction. For instance, when designing a network infrastructure, performance (with some probabilistic guarantees) is the main

* Work partially supported by EU-FET Project Agile, EU-FET Project MIKADO, EU-FET Project PROFUNDIS, and MIUR project SP4 Architetture Software ad Alta Qualit’a di Servizio per Global Computing su Cooperative Wide Area Networks.
QoS metric. When describing multimedia application visual and audible qualities would be the crucial parameters. Instead, for final users, the perceived QoS is not just a matter of performance but also involves availability, security, usability of the required services. Moreover, the user would like to have a certain control on QoS parameters in order to customize the invoked services, while network providers would like to have a strict control over services. The resolution of this tension will be inherently dynamic depending on the run-time context.

In our view, it is of fundamental importance to develop formal machineries to describe, compose and relate the variety of QoS parameters. Indeed, the formal treatment of QoS parameters would contribute to the goal of devising robust programming mechanisms and the corresponding reasoning techniques that naturally support the SOC paradigm. In this paper we face this issue by introducing a basic process calculus where QoS parameters are first class objects. Namely, QoS parameters are used to control behaviours.

The goal of the present paper is to identify a minimal set of constructs that provide an abstract model to control and coordinate services in presence of QoS constraints. This differentiates our proposal from other approaches. In particular, process calculi have been designed to model QoS in terms of performance issues (an example is provided by the probabilistic π-calculus [14]). Other process calculi have addressed the issues of failures and failure detection [12]. Process calculi equipped with powerful type systems have also been put forward to describe the behavioral aspects of contracts [10, 9, 8].

Some preliminary results towards the direction of this paper can be found in [2, 3]. Cardelli and Davies [2] introduced a calculus which incorporates a notion of communication rate (bandwidth) together with some programming constructs. In [15, 7, 3] a (hyper)graph model to control explicitly QoS attributes has been introduced. The graphical semantics allows us to describes interactions in accordance with the agreed QoS level as optimal paths in the model thus creating a bridge between formal models and the protocols used in practice. Here, we elaborate on the approach of [3] with the aim of bridging further the gap between formal theories and the pragmatics of software development.

Fundamental to our approach is the notion of QoS parameter or cost; a QoS cost is a tuple of values and each component of the tuple indicates a QoS dimensions. The values of the fields can be of different kind, for instance, the cost along the latency dimension could be a numerical value but the security values could have the form of sets of capabilities indicating the permissions to perform some operations on given resources, e.g. read or write a file. Compositionality of QoS costs is therefore a key element of our approach: the composition of QoS costs will be a QoS cost as well. Indeed, one might want to build a cost based on latency, availability and access rights of a service. To guarantee compositionality of different kinds of costs, we shall require values to be elements of suitable algebraic structures, called constraint semirings (c-semirings, for short), consisting of a domain and two operations, the additive (+) and the multiplicative (·) operations, satisfying some properties. The basic idea is that the former is used
to select among values and the latter to combine values. The additive operation of the c-semiring induces a partial order as follows: \( a \leq b \) iff \( a + b = b \).

Different kind of semirings have been proposed to model QoS costs. For instance, general algorithms for computing shortest paths in a weighted directed graph are based on the structure of semirings \cite{11}. The modelling of trustness in an ad hoc network exploits the semiring structure \cite{?}. C-semirings are a specific kind of semirings and they were originally proposed to describe and program constraints problems \cite{1}. C-semiring based methods have a unique advantage when problems with multiple QoS dimensions must be tackled. In fact, it turns out that cartesian products, exponentials and power constructions of c-semirings are c-semirings.

Our process calculus, KoS builds on KLAIM \textit{(Kernel Language for Agent Interaction and Mobility)} \cite{4}. KLAIM is an experimental kernel programming language specifically designed to model and program wide area network applications with located services and mobility. KLAIM naturally supports a peer-to-peer programming model where interacting peers (nodes in KLAIM terminology) cooperate to provide common sets of services. KoS primitives handle QoS costs as first class entities. For instance, an overlay network is specified by creating nodes \( \text{node}_\kappa (t) \) and new links \( s \mathbin{\prec} t \) and indexing them with the cost \( \kappa \) of the operation. Thus, for instance the expression \( s \mathbin{\prec} t \) states that \( s \) and \( t \) are connected by a link whose QoS parameters are given by \( \kappa \).

The operational semantics of KoS ensures that the QoS costs are respected during system evolution. Suppose for example that node \( s \) would interact by an operation of cost \( \kappa' \) with node \( t \) along the link \( s \mathbin{\prec} t \). This interaction will be allowed provided that the SLA contract of the link is satisfied, namely, \( \kappa' \leq \kappa \).

We shall illustrate the expressiveness of the calculus through several examples. This can appear as an exercise in coding a series of linguistic primitives into our calculus notation, but it yields much more because the encodings offer a practical illustration of how to give a precise semantic interpretation of QoS management. Indeed, the main contribution of this paper is the careful investigation of a minimal conceptual model which provides the basis to design programming constructs for SOC. We focus on the precise semantic definition of the calculus because it is a fundamental step to design programming primitives together with methods supporting the correct usages of the primitives and the formal verification of the resulting applications.

The rest of the paper is organized as follows. In the next section we illustrate a motivating example and, in Section 3, we introduce syntax and semantic of our cost calculus KoS. In Section 4, we deal with expressivity issues and in the subsequent one we present a more complex scenario and show how it can be tackled by following our approach.

## 2 A motivating example

Before introducing the formal definition of KoS, we prefer to show its usefulness in modelling a realistic, but simplified, example. Our purpose here is to give a
flavour of the underlying programming paradigm. We consider a scenario where \( n \) servers provide services to \( m \) clients and we focus on balancing the load of servers. Clients and servers are located on different nodes; a generic client node has address \( c_i \) while a generic server node has address \( s_j \). Clients issue requests to servers by spawning process \( R \) from their node to a server node. For simplicity, we abstract from the actual structures of costs, and we assume that clients and servers “knows” each other and cannot be created dynamically. Adding dynamicity is straightforward.

A generic client node \( M_i \), for \( i \in \{1, \ldots, m\} \), is described by the following term:

\[
M_i \triangleq c_i :: \langle s_1, \kappa_1 \rangle | \cdots | \langle s_n, \kappa_n \rangle | !C_\delta.
\]

Intuitively, \( M_i \) represents a network component with address \( c_i \), containing tuples of the form \( \langle s_j, \kappa_j \rangle \), for \( j \in \{1, \ldots, n\} \), and running process \(!C_\delta\). Each tuple \( \langle s_j, \kappa_j \rangle \) represents the load \( \kappa_j \) of the server \( s_j \) that the client perceives, thus the whole set of tuples represents a sort of directory service containing the SLA contract with the available servers. Operator \(!\) is the (standard) replication operator of process calculi: \(!C_\delta\) represents an unbounded number of concurrent copies of process \( C_\delta \). Finally, process \( C_\delta \) specifies the behaviour of the client and is defined as follows:

\[
C_\delta \triangleq (\langle u, v \rangle). \varepsilon_v[R]\@u.conv_\delta(u).\langle u, v \cdot \delta \rangle
\]

Initially, the client selects a server by non-deterministically inputting a tuple by means of the operation \( (\langle u, v \rangle) \). Once the input is executed, variables \( u \) and \( v \) are instantiated with the server name and its load, respectively. Afterward, the client tries to spawn process \( R \) to the chosen server \( u \). Execution of \( \varepsilon_v[R]\@u.conv_\delta(u) \) takes place only if a “suitable” link toward \( u \) exists. What here is meant for “suitable” is that the load \( v \) of the client must not exceed the value on the link.

Then, since remote spawning consumes the links traversed during the migration, the client attempts to re-establish a connection with \( u \) by executing \( conv_\delta(u) \). Notice that the operation \( conv_\delta(u) \) is used by the client to ask for a link having a cost increased of a quantity \( \delta \). Once the connection has been established, the client updates its SLA view of the servers load by inserting tuple \( \langle u, v \cdot \delta \rangle \) into its local directory service.

A generic server \( N_j \), for \( j \in \{1, \ldots, n\} \), is described as follows:

\[
N_j \triangleq s_j :: \langle h \rangle | \langle c_1, \kappa'_1 \rangle | \cdots | \langle c_m, \kappa'_m \rangle | !\langle S c_1 s_j \rangle | \cdots | !\langle S c_m s_j \rangle
\]

Similarly to clients, \( N_j \) encapsulates a directory service containing SLA data about the clients. This directory service is formed by tuples of the form \( \langle c_i, \kappa'_i \rangle \), for \( i \in \{1, \ldots, m\} \), each recording the cost \( \kappa'_i \) assigned to the link towards node \( c_i \), and by the current load of the server, represented by a tuple containing a natural number \( \langle h \rangle \). For any client \( c_i \) there is a load manager \( S c_1 s_j \) which decides whether a link with \( c_i \) can be re-established or not. Process \( S c s \) is
written as follows:

\[
S \triangleq (\exists l. l). \text{if } l < \text{max then } (c, ?v). \text{acc}_{f(v, l)}(c). (c, f(v, l))
\]

The load manager repeatedly acquires the tuple \(h\) (current load) and compares it with \text{max}, the maximum admissible load. According to this check, the process decides whether to accept requests for new connections coming from the client: the link is created only when \(h\) is less than \text{max}. The cost of the new link is computed by a function \(f\) and depends on both the old cost and the current load.

Finally, we assume that process \(R\) representing clients service requests is a sequential process of the form

\[
R \triangleq (?x). (x + 1) \ldots \text{actual request} \ldots (?y). (y - 1),
\]

Namely, \(R\) has a prologue and an epilogue which respectively increments and decrements the counter that measures the server load.

3 The calculus

This section introduces \(KoS\) a calculus that provides a set of basic primitives for modelling and managing QoS costs. \(KoS\) inherits from \(KAOS\) \cite{3}, the idea of considering QoS costs as first class entities. However, \(KoS\) also reconsiders many solutions adopted in \(KAOS\). The former calculus have been modified and significantly simplified to develop a kernel foundational calculus.

A \(KoS\) term represents a net made of \textit{nodes} which model places where computations take place or where services can be allocated/accessed. We assume as given a set of nodes \(S\) (ranged over by \(s, t, \ldots\)) that are connected by \textit{links} representing the middleware infrastructure, i.e., the interactions between two nodes can take place only if they are connected by a sequence of links. Links are weighted by “measures” that represent the cost of using the connections along to specific QoS dimensions.

3.1 Costs as constraint semirings

We assume existence of a set of \textit{costs} \(\mathcal{C}\), ranged over by \(\kappa\), that is a constraint semiring \cite{1} (\textit{c-semiring}). Formally, we have the following definition.

\textbf{Definition 1 (C-semiring).} An algebraic structure \((A, +, \cdot, 0, 1)\) is a c-semiring if \(A\) is a set \((0, 1 \in A)\), and + and \cdot are binary operations on \(A\) that satisfy the following properties:

- \(+\) (additive operation) is commutative, associative, idempotent, \(0\) is its unit element and \(1\) is its absorbing element;
- \(\cdot\) (multiplicative operation) is commutative, associative, distributes over +, \(1\) is its unit element, and \(0\) is its absorbing element.
Operation $+$ induces a partial order on $A$ defined as $a \leq_A b \iff a + b = b$. The minimal element is thus $0$ and the maximal $1$. $a \leq_A b$ means that $a$ is more constrained than $b$.

An example of c-semiring is $\langle \omega, \min, +, +\infty, 0 \rangle$, where $\omega$ is the set of natural numbers, the minimum between natural numbers is the additive operation and the sum over natural numbers is the multiplicative operation. Notice that in this case the partial order induced by the additive operations is the inverse of the ordinary total order on natural numbers. Another example of c-semiring is $\langle \wp(\{A\}), \cup, \cap, \emptyset, A \rangle$, where $\wp(A)$ is the powerset of a set $A$, and $\cup$ and $\cap$ are the usual set union and intersection operations.

KoS does not take a definite standing on which of the many c-semiring structures to use. The appropriate c-semiring to work with should be chosen, from time to time, depending on the kind of QoS dimensions one intends to model. Below, we introduce some c-semiring structures together with the QoS dimension they handle:

- $\langle \{true, false\}, \lor, \land, false, true \rangle$ (boolean): Network and service availability.
- $\langle \Real^+, \min, +, +\infty, 0 \rangle$ (optimization): Price, propagation delay.
- $\langle \Real^+, max, min, 0, +\infty \rangle$ (max/min): Bandwidth.
- $\langle [0, 1], max, \cdot, 0, 1 \rangle$ (probabilistic): Performance and rates.
- $\langle [0, 1], max, min, 0, 1 \rangle$ (fuzzy): Performance and rates.
- $\langle 2^N, \cup, \cap, \emptyset, N \rangle$ (set-based, where $N$ is a set): Capabilities and access rights.

C-semiring based methods have a unique advantage when problems with multiple QoS criteria must be tackled. In fact, it turns out that cartesian products, exponentials and power constructions of c-semirings are c-semirings. Thus the same concepts and algorithms can be applied again and again. The Hoare Power Domain of a c-semiring is of special interest since it let us formalize multi-criteria optimization problems.

### 3.2 Syntax

The syntax of KoS is presented in Table 1. Other than existence of $C$, it is assumed existence of a set of names $\mathcal{N}$, ranged over by $r, s$ and $t$. First-class values, ranged over by $u$ and $v$, can be either costs or names.

The syntax for nets allows the (possibly empty) parallel composition of located processes and links. A located process $s :: P$ consists of a name $s$, called the address of $P$, and the process $P$ running at $s$. A link $s \overset{\kappa}{\leftrightarrow} t$ states that $s$ and $t$ are connected by a link of cost $\kappa$. The net $(\nu s)N$ is a net that declares $s$ as restricted in $N$, which is the scope of the restriction.

The syntax for processes is standard. The symbol $0$ overloads the symbol for empty nets; however, the contexts will clarify whether it refers to processes or nets. Prefixes $\gamma$ encompass actions for

- creating a node $(\text{node}_\kappa(t))$ or a connection to/from another node $(\text{con}_\kappa(t), \text{acc}_\kappa(t))$,
- exchanging tuples of values ($(T)$ and $(v_1, \ldots, v_n)$),


\[ N, M ::= \begin{array}{ll}
N & \text{Empty net} \\
| s :: P & \text{Located Process} \\
| s ≲ t & \text{Link} \\
| (\nu s)N & \text{Node restriction} \\
| N \parallel M & \text{Net composition}
\end{array} \]

\[ P, Q ::= \begin{array}{ll}
0 & \text{Null process} \\
| \gamma.P & \text{Action prefixing} \\
| (\nu s)P & \text{Restriction} \\
| P | Q & \text{Parallel process} \\
| !P & \text{Iteration}
\end{array} \]

\[ \gamma ::= \begin{array}{ll}
node_\alpha(t) & \text{Node creation} \\
con_\alpha(t) & \text{Connection request} \\
acc_\alpha(t) & \text{Connection acceptance} \\
(T) & \text{Input} \\
\langle v_1, \ldots, v_n \rangle & \text{Output} \\
\varepsilon_\alpha[P]@t & \text{Remote process spawning}
\end{array} \]

\[ T ::= \varepsilon | v | ?x | \neg v | T, T \]

Table 1. KoS Syntax

- remotely spawning a process \( (\varepsilon_\alpha[P]@t) \).

Links are oriented, indeed \( s ≲ t \) allows a process to be spawned from \( s \) to \( t \) but not the viceversa. The creation of new links is obtained by synchronising actions \( con_\alpha(t) \) and \( acc_\alpha(s) \) performed at \( s \) and \( t \), respectively.

Communication involves exchange of tuples (i.e. finite sequences) of values that are retrieved via pattern matching. Input prefixes use templates \( T \), namely finite sequences of values or placeholders (written as \(?x\)). Execution of an output prefix causes generation of a tuple of values \( v_1, \ldots, v_n \). \( \varepsilon \) denotes both the empty template and the empty tuple. Hereafter, we let \( t \) range over tuples of values and, given a template \( T \) and a tuple \( t \), we let \( T_i \) and \( t_i \) denote the \( i \)-th element of \( T \) and \( t \), respectively.

The placeholder \(?x\) binds the occurrences of \( x \) in the rest of the template, namely, in \(?x, T\), the scope of \(?x\) is \( T \). The set \( \text{bn}(T) \) collects the names bound in \( T \) while \( \text{fn}(T) \) denotes the names having free occurrences in \( T \). The definitions of \( \text{bn}(\_\_ \_{}) \) and \( \text{fn}(\_\_ \_{}) \) are given below:

\[
\text{fn}(T) = \begin{cases} 
\emptyset, & \text{if } T = \varepsilon \\
\{s\} \cup \text{fn}(T'), & \text{if } T = s, T' \\
\text{fn}(T') \setminus \{s\} & \text{if } T = \neg s, T'
\end{cases}
\]

\[
\text{bn}(T) = \begin{cases} 
\emptyset, & \text{if } T = \varepsilon \\
\text{bn}(T'), & \text{if } T = s, T' \\
\{s\} \cup \text{bn}(T') & \text{if } T = \neg s, T'
\end{cases}
\]

We consider as equivalent those templates that differ only for renaming of bound names. The template \( \neg v \) tests for inequality, namely, it requires the matching tuple to contain a value different from \( v \) (see Definition 7). The only binders of
the calculus are the placeholder \(?x\) and the node restriction \(\nu s\). Recall that node names might appear in the costs (e.g., for specifying access rights), hence, we write \(\text{fn}(\kappa)\) to denote the names appearing in \(\kappa\). Moreover, we require that costs do not bind node names, therefore, \(\text{bn}(\kappa)\) is empty, for any cost \(\kappa\). We formally define free and bound names of nets and processes as follows. In the following we write \(\text{fn}(\_\_\_)\) (resp. \(\text{bn}(\_\_\_)\)) as an abbreviation for \(\text{fn}(\_)\cup \text{fn}(\_)\) (\(\text{bn}(\_)\cup \text{bn}(\_)\), respectively).

**Definition 2 (Free and bound names).** The free names of prefix actions are defined as expected: \(\text{fn}(\gamma) = \text{fn}(\kappa) \cup \{s\}\), if \(\gamma \in \{\text{node}\_\kappa(s), \text{con}\_\kappa(s), \text{acc}\_\kappa(s)\}\), \(\text{fn}(T) = \text{fn}(T), \text{fn}(\langle v_1, \ldots, v_n \rangle) = \text{fn}(v_1) \cup \ldots \cup \text{fn}(v_n)\) and \(\text{fn}(\varepsilon\_\kappa[P]\@s) = \text{fn}(\kappa, P) \cup \{s\}\). Bound names of \(\gamma\) are defined similarly, e.g., \(\text{bn}(T) = \text{bn}(T)\) and \(\text{bn}(\varepsilon\_\kappa[P]\@s) = \text{bn}(P)\) (while in the remaining cases is the empty set).

The sets \(\text{fn}(\_)\) and \(\text{bn}(\_)\) of free and bound names of processes and nets are defined accordingly. The only non-standard case is that for links where we let \(\text{fn}(r \overset{\kappa}{\rightarrow} s) = \text{fn}(\kappa) \cup \{s, r\}\) and \(\text{bn}(r \overset{\kappa}{\rightarrow} s) = \emptyset\).

As usual, we consider as equivalent those processes or nets that are obtained by \(\alpha\)-converting bound names. Moreover, we assume the following structural congruence laws.

**Definition 3 (Structural congruence).** The relation \(\equiv_{P} \subseteq P \times P\) is the least equivalence relation on processes (containing \(\alpha\)-conversion and) satisfying the following axioms:

- \((P, |, 0)\) is a commutative monoid;
- \(!P \equiv_{P} P \ | \ !P\).

The relation \(\equiv_{N} \subseteq N \times N\) is the least equivalence relation on nets (containing \(\alpha\)-conversion and) satisfying the following axioms:

- \((N, ||, 0)\) is a commutative monoid;
- if \(P \equiv_{P} Q\) then \(s :: P \equiv s :: Q\);
- \(s :: P \ | \ Q \equiv s :: P \ | \ s :: Q\);
- \(s :: (\nu t)P \equiv (\nu t)(s :: P)\), if \(t \neq s\);
- \((\nu s)(N || M) \equiv N || (\nu s)M\), if \(s \notin \text{fn}(N)\);
- \((\nu s)(\nu t)N \equiv (\nu t)(\nu s)N\).

The last axiom of Definition 3 states that the order of the restrictions is irrelevant, hence we can write \((\nu s_1, \ldots, s_n)N\) instead of \((\nu s_1) \ldots (\nu s_n)N\).

### 3.3 Semantics

We define the operational semantics of KoS by means of a labelled transition system that describes the evolution of nets. In the semantic clauses, it is useful to define a function that, given a net \(N\), yields the names that are used as node addresses in the net.
Definition 4 (Addresses). Let \( \text{addr} \) be the function given by:

\[
\text{addr}(N) = \begin{cases} 
\emptyset, & N = 0 \lor N = s \not\sim t \\
\{s\}, & M = s :: P \\
\text{addr}(M) \setminus \{s\}, & N = (\nu s)M \\
\text{addr}(N_1) \cup \text{addr}(N_2), & N = N_1 \parallel N_2.
\end{cases}
\]

Notice that \( \text{addr}(N) \subseteq \text{fn}(N) \), but not necessarily \( \text{addr}(N) = \text{fn}(N) \), for instance if \( N = s :: \langle t \rangle 0 \) then \( \text{fn}(N) = \{s, t\} \) while \( \text{addr}(N) = \{s\} \). Basically, \( \text{addr}(N) \) collects those free names of \( N \) that effectively occur in \( N \) as address of some node.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(LINK)</td>
<td>( s \not\sim t \rightarrow s \overset{\kappa}{\rightarrow} t \rightarrow s \overset{\kappa}{\rightarrow} \emptyset )</td>
</tr>
<tr>
<td>(PREF)</td>
<td>( s :: \gamma.P \rightarrow s :: P ), ( \gamma \not\in {\text{node}<em>\kappa(t), \text{con}</em>\kappa(t), \text{acc}_\kappa(t)} )</td>
</tr>
<tr>
<td>(NODE)</td>
<td>( s :: \text{node}_\kappa(t).P \rightarrow s :: P \parallel s \overset{\kappa}{\rightarrow} t \parallel t :: 0, s \not= t )</td>
</tr>
<tr>
<td>(CON)</td>
<td>( \frac{N \rightarrow s \text{con}<em>\kappa(t) \rightarrow N' \quad M \rightarrow t \text{acc}</em>\kappa(s) \rightarrow M'}{N \parallel M \rightarrow s \overset{\kappa}{\rightarrow} t \parallel M' \parallel s \overset{\kappa}{\rightarrow} t} )</td>
</tr>
<tr>
<td>(LEVAL)</td>
<td>( s :: \varepsilon_\kappa[Q] \rightarrow s :: P \parallel s :: Q )</td>
</tr>
<tr>
<td>(ROUTE)</td>
<td>( \frac{N \rightarrow r \varepsilon_\kappa(P) \rightarrow t' \quad M \rightarrow r \text{link}<em>\kappa t' \rightarrow M' \quad \kappa' \parallel \kappa'' \leq \kappa}{N \parallel M \rightarrow r \varepsilon</em>\kappa(P) \rightarrow t'' \quad N' \parallel M'} )</td>
</tr>
<tr>
<td>(LAND)</td>
<td>( \frac{N \rightarrow r \varepsilon_\kappa(P) \rightarrow t' \quad M \rightarrow r \text{link}<em>\kappa t' \rightarrow M' \quad \kappa' \parallel \kappa'' \leq \kappa}{N \parallel M \rightarrow r \varepsilon</em>\kappa(P) \rightarrow t'' \quad N' \parallel M' \parallel t :: P} )</td>
</tr>
<tr>
<td>(COMM)</td>
<td>( \frac{N \rightarrow r(T) \rightarrow N' \quad M \rightarrow t \rightarrow M' \quad \triangledown(T, t) = \sigma}{N \parallel M \rightarrow N' \sigma \parallel M'} )</td>
</tr>
</tbody>
</table>

Table 2. Network semantics
Definition 5 (Localized Actions). Let $\gamma$ be a prefix, then the localized prefix $\gamma@s$ is defined as follows:

$$\gamma@s = \begin{cases} s \varepsilon_k^s(P)@t & \text{if } \gamma = \varepsilon_k^s(P)@t \\ s \gamma & \text{otherwise} \end{cases}$$

The syntax of localized actions $\alpha$ is given below:

$$\alpha ::= \gamma@s \mid s \text{ link } t \mid \tau$$

We let $\text{fn}(\gamma@s) = \text{fn}(\gamma) \cup \{s\}$ and $\text{bn}(\gamma@s) = \text{bn}(\gamma)$.

Definition 6 (Nets semantics). The operational semantics of nets is given by the relation $\rightarrow \subseteq N \times (\alpha \times C) \times N$. Relation $\rightarrow$ is defined by the rules in Table 2 and the following standard rules:

\[
\begin{align*}
\text{(res)} & \quad N \xrightarrow{\tau} M \\
\text{(par)} & \quad N \xrightarrow{\alpha} N' \parallel M \quad N \parallel M' \xrightarrow{\alpha} N \parallel M' \\
\text{(str)} & \quad N \equiv N' \xrightarrow{\alpha} M' \equiv M
\end{align*}
\]

Intuitively, $N \xrightarrow{\alpha} M$ states that the net $N$ can perform the transition $\alpha$ to $M$ at cost $\kappa$. By inspecting the rules in Table 2, it is evident that all local transitions (communications, node or link creations) have unitary cost, while the only non-trivial costs appear on the transitions that spawn processes or show the presence of links. Let us give more detailed comments on the rules in Table 2.

Rule (link) states that a link within a net disappears once it has been used (i.e, it has manifested itself). These transitions are used in the premises of rules (route) and (land) for establishing a path between two nodes such that a remote evaluation can take place.

Rule (pref) accounts for action prefixing; node creation, however, deserves a specific treatment that is defined in rule (node). The side condition of (pref) also states that no link from $s$ to itself can be created. Indeed, we assume that transitions that involve only the local node have unitary cost and are always enabled.

Rule (node) allows a process allocated at $s$ to use a name $t$ as the address of a new node and to create a new link from $s$ to $t$ with cost $\kappa$. The side condition of (par) prevents that new nodes (and links) are created by using addresses of existing nodes.

Rule (con) adds a new link between two existing addresses $s$ and $t$: the link is created only if the processes at $s$ and $t$ declare compatible costs. More precisely, the accepting node $t$ is willing to connect only to those nodes that declare a cost lower than $\kappa'$. If this condition hold, a new link is added to the net, such link has the cost proposed by $s$. One can think of $s$ as asking for the connection with at least some characteristics expressed by $\kappa$ and $t$ establishes the connection only when he can enforce the requirement of $s$, namely $\kappa \leq \kappa'$. 

10
Rule (LEVAL) states that the local spawning of a process is always enabled while rules (ROUTE) and (LAND) control process migration and require more detailed explanations. A remote spawning action $\varepsilon \cdot [P]st$ consists of the migrating process $P$, the arrival node $t$ and a cost $\kappa$ expressing that $P$ must be routed on a path costing $\kappa$ at most $\kappa$. Differently from the local spawning of processes, remote spawning is not always possible, it is indeed mandatory that the net contains a path costing $\kappa$ at most $\kappa$. The cost of the path between $s$ and $t$ must not exceed the “price” $\kappa$ that the spawner is willing to “pay”. Notice that this semantically describes the SLA agreement on the mobility of processes. This is formally achieved by rules (ROUTE) and (LAND). More specifically, rule (ROUTE) states that, if the migrating process can go through an intermediate node $r$ and a link from $r$ to a node $s'$ does not exist, the cost $\kappa'$ of the partial path from $s$ to $r$ composed with the cost $\kappa''$ of the link from $r$ to $s'$ must be lower than $\kappa$. If this is the case, a transition can be inferred stating that $P$, spawned from $s$, can go through $s'$ with cost $\kappa' \cdot \kappa''$. Rule (LAND) is similar to (ROUTE) but describes the last hop of $P$, namely when the target node $t$ is reached. In this case, $P$ is spawned at $t$, provided that the cost of the whole path that has been found is lower than $\kappa$.

Rule (COMM) establishes that a synchronization takes place provided that sender and receiver are allocated at the same node and that the template and the tuple match according to the definition below. Hereafter, we use $\sigma$ to denote a substitution, i.e. a map from names to names and costs, and $\sigma[\sigma']$ to denote the composition of substitutions, i.e. the substitution $\sigma''$ defined as follows: $\sigma''(x) = \sigma'(x)$ if $x \in \text{dom}(\sigma')$, $\sigma''(x) = \sigma(x)$ if $x \in \text{dom}(\sigma) - \text{dom}(\sigma')$.

**Definition 7 (Pattern matching).** A template $T$ and a tuple $t$ match when the following function is defined

$$\triangleright (T, t) = \begin{cases} \varepsilon & \text{if } (T = \varepsilon \land t = \varepsilon) \lor (T = v \land t = v) \\ \varepsilon & \text{if } T = \neg v \land t = v' \land v \neq v' \\ \{v/\varepsilon\} & \text{if } T = ?x \land t = v \\ \sigma[\sigma'] & \text{if } T = F, T' \land t = v, t' \land \triangleright (F, v) = \sigma \land \triangleright (T', t') = \sigma' \end{cases}$$

where the application of a substitution to a template, $T\sigma$, is defined as follows:

$$T\sigma = \begin{cases} \varepsilon & \text{if } T = \varepsilon \\ v, T\sigma & \text{if } T = x, T' \land \sigma(x) = v \\ x, T'\sigma & \text{if } T = x, T' \land x \notin \text{dom}(\sigma) \\ ?x, T'\sigma & \text{if } T = ?x, T' \land \sigma' = \sigma[x/\varepsilon]. \end{cases}$$

Under the conditions of (COMM), the substitution $\triangleright (T, t)$ is applied to the receiver. Note that $\triangleright$ may not be defined, for instance $\triangleright (\neg s, s)$ does not yield any substitution and, therefore, the match in such a case does not hold.

As a final remark, notice that “ill-formed” nets can arise during computations. For instance, consider $s :: (?x).\text{conv}\langle x \rangle \mid \langle k' \rangle$. After the communication,
the net becomes \( s \equiv \text{con}_s \langle \kappa' \rangle \) which has no meaning. We treat these cases as run-time errors and observe that they might be avoided by a suitable typing discipline.

4 Example

In this section we present some specification examples. To make the presentation more readable let us introduce some notational conventions. First, we avoid writing trailing \( 0 \) processes, second, we write \( \varepsilon[P]@r \) instead of \( \varepsilon_1[P]@r \) and similarly for \( \text{node}_1(t) \), \( \text{con}_1(t) \) and \( \text{acc}_1(t) \). The examples reported in the rest of this section will be exploited in the more significative example reported in the Appendix 6.

**Boolean expressions**  Booleans are encoded as processes that allocate a pair of names to a node:

\[
\begin{align*}
\text{True } r \triangleq & \ (\nu t)\varepsilon[\langle t, t \rangle]@r \\
\text{False } r \triangleq & \ (\nu f, f'\prime)\varepsilon[\langle f, f'\prime \rangle]@r.
\end{align*}
\]

The truth and the falsity are tested by checking that the names in a pair are equal or different, respectively. The following process tests for the equality of two names:

\[
\text{Test } x \ y \ r \triangleq \ (\nu t)(\text{node}_1(t)\varepsilon[\text{Eval } y \ r \ | \ \langle x \rangle]@t),
\]

where \( \text{Eval } y \ r \triangleq (y).\text{True } r \ | \ (\neg y).\text{False } r \). Process \( \text{Test} \) spawns the tuple \( \langle x \rangle \) and the \( \text{Eval} \) process onto a newly generated node so that the first or the second component of \( \text{Eval} \) have exclusive access to \( \langle x \rangle \). Notice that only one of the components can consume the tuple, indeed, either \( x = y \) (and only the pattern \( (y) \) matches \( \langle x \rangle \)) or \( x \neq y \) (and only the pattern \( (\neg y) \) matches \( \langle x \rangle \)). Finally, \( \text{True} \) or \( \text{False} \) allocates on node \( r \) the truth value corresponding to \( x = y \).

The encoding of boolean values is indeed an example of a standard programming metaphor for finding and handling services. For instance assume that we want to describe a look up mechanism for discovering distributed services. For instance, the web services technology allows deploying new services by gluing together those that have been published. The web service composition, however, requires a look up phase where the available service must be discovered. In the boolean example, processes \( \text{True} \) and \( \text{False} \) are the services that have been published and composed together to provide the \( \text{Test} \) service. Notice that the look up phase does not require the knowledge of the service name but only that of the “schema” of the service. For instance, whenever a new “true” service is published it suffice to generate a new name and use it for building the “schema” for the true service (i.e., a pair of two equal names).

Assuming the above encoding of booleans, we can represent standard control structures such as \text{if-then-else} \ and \text{while}.
Public, private, permanent and stable links Links in KoS are public entities: when available they can be exploited by all processes. Consider the following KoS net:

\[ N \triangleq s :: \varepsilon_3[P]at \parallel s \not\rightarrow r \parallel r :: \text{con}_2(t).\varepsilon_2[Q]at \parallel t :: \text{acc}_2(r), \]

where costs are the c-semiring of natural numbers. Net \( N \) has three nodes \( s, r \) and \( t \) and, initially, only \( s \) and \( r \) are connected by a link with cost 1. Node \( s \) is trying to spawn \( P \) on \( t \) which is not possible because there is no path from \( s \) to \( t \). Node \( r \) is willing to spawn a process \( Q \) on \( t \), as well; however, \( r \) is aware that a link must be first created. Node \( t \) simply accepts requests for establishing a link from \( r \). According to the semantics of KoS, initially, it is only possible to synchronize \( \text{con}_2(t) \) and \( \text{acc}_2(r) \) which, by applying rule \( \text{(con)} \) leads to

\[ N' \triangleq s :: \varepsilon_3[P]at \parallel s \not\rightarrow r \parallel r :: \varepsilon_2[Q]at \parallel r \not\rightarrow t \parallel t :: \emptyset. \]

Now, applying rules \( \text{(pref)}, \text{(link)} \) and \( \text{(land)} \) we derive

\[ N' \xrightarrow{\frac{\tau}{2}} s :: \varepsilon_3[P]at \parallel s \not\rightarrow r \parallel r :: \emptyset \parallel t :: Q. \]

Notice that the link between \( r \) and \( t \) is consumed by the migration of \( Q \) hence \( P \) cannot reach \( t \). However, \( N' \) can also evolve differently, in fact, both the two spawning actions are enabled, because the creation of the link between \( r \) and \( t \) has also provided a path from \( s \) to \( t \) costing 3. Hence, by rules \( \text{(pref)}, \text{(link)}, \text{(route)} \) and \( \text{(land)} \) we can also derive

\[ N' \xrightarrow{\frac{\tau}{3}} s :: \emptyset \parallel r :: \varepsilon_2[Q]at \parallel t :: P. \]

Noteworthy, the migration of \( P \) prevents \( Q \) to be spawned because the link created by \( r \) has been used by \( P \).

In general, this kind of interference should be avoided and this can be done in KoS by expressing private links which can be specified by exploiting the properties of c-semirings. The intuition is to allow the use of a link only whether the traversing process has the appropriate “rights”. If we represent access rights as sets of names, then a process must “know” all the names needed for traversing the link. For instance, consider the following net:

\[ s :: \varepsilon_{\{r\}}[P]at \parallel s \not\rightarrow s', \]

process \( P \) can traverse the link \( s \not\rightarrow s' \) because it “knows” \( r \), that is the only name required to traverse the link. Noteworthy, \( P \) could not traverse \( s \not\rightarrow s' \) because it does not expose name \( u \).

The c-semiring that allows us to represent access rights is the c-semiring \( \mathcal{R} \) given by \( \mathcal{R} = (\wp_{\text{fin}}(S) \cup \{S\}, \cap, \cup, S, 0) \). It is straightforward to prove that \( \mathcal{R} \) is a c-semiring; moreover, the order induced by the additive operation of \( \mathcal{R} \) is the inverse of the set inclusion (i.e., \( X \leq Y \iff Y \subseteq X \)).

13
Therefore, a private link between the nodes $s$ and $t$ can be specified as

$$(\nu p)(s :: P \parallel s \overset{\{p\}}{\to} t \parallel t :: Q),$$

where $p \not\in \text{fn}(P,Q) \cup \{s, t\}$. Indeed, in order to pass through link $s \overset{\{p\}}{\to} t$, a process must exhibit the “password” $p$. The knowledge of $p$ is handled by enlarging the scope of the restriction and communicating it.

We conclude by illustrating how one could implement permanent links, i.e. links that are always available, by exploiting replication:

$$s :: !\text{con}_\kappa(t) \parallel t :: !\text{acc}_\kappa(s)$$

A slight variation are stable links, which are links existing until a given condition is satisfied.

$$\text{Stable}_s G t \triangleq !\text{con}_\kappa(t) \mid \varepsilon[\text{While } G \text{ do } \text{acc}_\kappa(s) \text{ od } 0]$$

Cryptography Interestingly enough, by exploiting private links, KAOS can encode the encryption/decryption mechanisms usually adopted for expressing security protocols. Generally, process algebras need to be extended with ad-hoc mechanisms in order to express cryptography. In fact, principals can be represented as processes sharing secrets (the cryptographic keys). Following [?], many approaches have extended previously defined process algebras by enriching the language with cryptograms and with mechanisms for decrypting them.

Differently from the other approaches, KoS can encode such primitives. Consider the following net:

$$(\nu k, s_k)(i :: P \parallel i \overset{k}{\to} s_k \parallel s_k :: M \parallel s_k \overset{k}{\to} r \parallel r :: Q), \quad (1)$$

and assume that the only links from/to $s_k$ are those appearing in (1). Net (1) aims at representing the initiator $i$ and the responder $r$ of a protocol that share a key $k$. According to (1) a key is modelled by means of a pair made of a name and a node which roughly speaking contains those messages that are encrypted with $k$.

The intuition is that encrypting corresponds to allocating a message on $s_k$ while decrypting corresponds to the possibility of “jumping” on $s_k$ and reading a message or, in other words, to the knowledge of $k$ for traversing links $i \overset{k}{\to} s_k$ or $s_k \overset{k}{\to} r$.

5 Conclusion

We have formally defined KoS a process calculus that provides basic primitives to describe QoS requirements of distributed applications. We demonstrated the applicability of the approach by specifying some expressive case studies.
Our research program is to provide a solid foundation to drive the design of languages and middleware having application-oriented QoS mechanisms. The work reported here is a preliminary step in this direction. In terms of calculus design, the current definition of KoS assumes that links are the basic construct to manage QoS interactions and cooperation. This is a reasonable assumption for several cases. For instance, in this paper we handled the QoS composition between two different overlay networks (see Section 6) by suitable links. However, one could interpret QoS composition of overlay networks in a more general sense than adding suitable links. An interesting challenge for future research is to extend KoS with more general mechanisms for composing overlay networks than simple parallel composition via links.

There are a number of ways in which KoS must be extended. For instance, it would interesting to develop type systems which would allow determining QoS properties of processes. We plan to extend types for access control of [6, 5] to deal with QoS attributes. In particular, it would be interesting to exploit such types to capture the notion of contract.

Finally, the issue of verification should be also be considered. In particular, we can directly apply the techniques developed in [3] for generating and analyzing shortest-path problems of KoS specifications. Moreover, we plan to develop an observational semantics for KoS based on the idea of observing QoS costs.

References

9. N. Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In J. van Leeuwen, O. Watanabe, M. Hagiya,
6 Appendix

We consider a scenario where a service is replicated on different nodes and can be invoked through a unique handler $H$ that manages the requests of the clients. This kind of architectures is adopted from Internet Service Providers (ISP) that offer dial-up connection to end-users (EU). In this case a telecommunication company (TC) handles the phone overlay networks. The EU connects to the “nearest” ISP server by dialing a single (country-wide) number. The TC takes care of dispatching the call to the closest ISP server on the overlay network. There are (at least) two possible way of connecting the EU and the ISP server. Either the TC establishes a direct connection between the EU and the ISP, or the TC act as a gateway between the phone overlay network and the IP overlay network. Both solutions can be easily expressed in KoS.

Figure 6 illustrates the logical architecture of the system: the handler manages the requests (e.g., controls the access rights of the client), looks for a suitable server, and forwards the request, while trying to balance the load of any replica of the service. Hence, the request of a client $C$ might not be forwarded to the “best” server from the client’s point of view. In this case, $H$ provides another server to $C$, however, the client may or may not commit to use it.

The simplest way to model this composed overlay network is to assume that the link between $C$ and $H$ have costs expressing the access rights of $C$. When a server $s$ meeting both the request of $C$ and the load constraints is found, $H$ replies to $C$ and tells $s$ to accept a (private) link from $C$. Hereafter, we assume that $h$ is the node address of $H$. We detail the client first:

$$C \triangleright_{pr} C' = (\nu r)(\nu s'\nu p)(\nu c)(\nu r'\nu s')(\nu p')$$

If $c_{pr'} < c$ then
$$\exists_{s'} \exists_{p'} \exists_{p} \exists_{r'} \exists_{r} \exists_{s} \exists_{p} \exists_{r} \exists_{s}$$
then $con_{\{r,p\}}(s).e_{\{s,p\}}[R]@s$
else $con_{\{r,p\}}(s).e_{\{s,p\}}[\langle r, \text{“to-much”} \rangle]@s$. 

16
The process $C$ requests $H$ to find a server and waits for the response. The request contains $c$, the node address of $C$, and a private name $r$. The name $r$ can be thought of as the unique marker of the request so that only $C$ will acquire data corresponding to the request $r$. Possibly, $H$ returns a response (marked with $r$) containing the server address $s$, the price $pr'$, and the password $p$ for the private link. Finally, $C$ establishes a private link with $s$ and, depending on the price $pr'$ required by the server, either raises its request $R$ (‘then’ branch) or notifies $s$ that the service is too expensive (‘else’ branch).

The definition of $H$ requires the following auxiliary processes.

\[
\begin{align*}
Rd(T) & \triangleq (T).\{t_T\} \\
Lt_s r i & \triangleq Rd(r, ?j). If \ s, j \leq i \ then \ True \ else \ False
\end{align*}
\]

Process $Rd(T)$ looks for a tuple matching $T$ and immediately re-generates the consumed tuple; this is denoted by $t_T$ which is obtained from $T$ by removing all the ’?’ occurring in its placeholders. Then, process $Lt_s r i$, interpreting $(r, v)$ as a “cell” having address $r$ and containing value $v$, reads the value in $r$ and compares it with $i$. 

Fig. 1. Logical architecture
\[ H \triangleq ! (\text{"connect"}, ?x, ?r), (r, 1) \]

While \( H \) \( l \) \( n \)serv

do
\((r, ?i). \text{Rd} (\text{pref}(x, i), ?l, ?pr.)\)
\If \( H \) \( l \geq \max \)
\then \((r, i + 1)\)
\else
\((\nu p) (\epsilon[\text{"newlink"}, x, r, p] \circ \text{pref}(x, i)). \)
\( \epsilon[\langle r, \text{pref}(x, i), pr, p \rangle] \circ \text{ax} \).
\((x, ?v). \text{acc}_{(v, l)}(x). (x, f(v, l)). (\text{pref}(x, i), ?l, ?pr). (\text{pref}(x, i), l + 1, pr)) \)
\od \( \epsilon[\langle r, \text{"no-server-available"} \rangle] \circ \text{ax} \).

The process \( H \) is continuously listening for a connection request. Once such a request is issued, from a client at \( x \), \( H \) starts scanning the server list (\( n \)serv is the number of servers). For each server \( s \), node \( h \) contains a tuple \( \langle s, l, pr \rangle \) where \( l \) is an estimation of the load of \( s \) and \( pr \) is the price for using \( s \). Also, for each client \( x \), \( h \) maintains a tuple \( \langle x, \kappa \rangle \) that reports the connection between \( H \) and \( x \) (as done in Section 2). Moreover, \( H \) uses a function \( \text{pref} \) that, given the client address \( x \) and the index \( i \), yields the \( i \)-th server “preferred” by the client. At the \( i \)-th iteration of the while loop, \( H \) reads the information of the \( i \)-th server preferred by \( x \) and, if the load of such a server is too high, the cycle is repeated provided that more servers are left (‘then’ branch); otherwise, a password \( p \) for a private link is generated and communicated to both \( x \) and the selected server. The server will accept a private link creation from \( x \) so that the client owning the password \( p \) can perform a request at \( s \). Finally, \( H \) re-establish a link with \( x \) according to the new load of the servers by exploiting the function \( f \) and reflecting this changes in the tuple corresponding to \( x \) (i.e., \( \langle x, f(v, l) \rangle \)), as in Section 2. Indeed, the mechanism of load balancing is the one defined in Section 2, the only difference being that now \( H \) is the unique handler that manages the connections with the clients.

Given \( H \) and \( C \), the servers must simply wait for a connection request (issued from \( H \)) and establish the private connection with the client:

\[ S \triangleq ! (\text{"newlink"}, ?x, ?r, ?p). \text{acc}_{(r, p)}(x) \ldots \text{wait & execute} \ldots \]

Once the request has been served, \( S \) simply updates the load of \( s \).

A net where \( C, H \) and \( S \) work can be defined as follows.

\[
\begin{align*}
\parallel i = 1, \ldots, m \quad \mathbf{x}_i :: & C \quad \kappa_i \quad pr_i \quad x_i \quad \mid \text{acc}(h) \quad \mid \text{con}_{\kappa_i}(h) \\
\parallel i = 1, \ldots, m \quad h :: & \text{com}(x_i) \quad \parallel h :: H \quad \parallel \\
\parallel (\nu s_1, \ldots, s_n) (\parallel j = 1, \ldots, n \quad h :: \text{com}(s_j) \quad \mid \langle s_j, l_j, pr_j \rangle \mid s_j :: S) \end{align*}
\]

where \( \parallel i = 1, \ldots, m \quad N_i \) shortens \( N_1 \parallel \ldots \parallel N_n \).

The other kind solution discussed at the beginning of this section can be achieved by exploiting the possibility offered by \( \text{KoS} \) of “connecting” links to
form paths between nodes. More precisely, instead of connecting directly the client’s node \( x \) and (the node of) the selected server \( s \), we can connect \( h \) and \( s \) so that the client’s request at \( s \) is routed through \( h \). In this scenario, assuming that the link between the \( x \) and \( h \) is permanent, we have:

\[
C' \triangleright (\nu r)(\varepsilon_{k,1}[\langle \text{“connect”, } x, r \rangle]@h.
\]

\[
\]

\[
\text{If } x \text{ pr'} < pr \text{ then } \varepsilon_{k,\{p,r\}}[R]@s \text{ else } \varepsilon_{k,\{p,r\}}[(r, \text{“to-much”})]@s).
\]

Accordingly, \( H' \) can be simplified as follows.

\[
H' \triangleq !(\langle \text{“connect”, } ?x, ?r \rangle,\langle r, 1 \rangle)
\]

\[
\text{While}_h L_{t_h r nserv}
\]

\[
\text{do (r, ?i).Rd } (pref(x, i), ?l, ?pr).
\]

\[
\text{If } h \ l \geq \max
\]

\[
\text{then } (r, i + 1)
\]

\[
\text{else (r p)()}(\varepsilon_{k,\{\text{“newlink”, } r, p\}}@pref(x, i).
\]

\[
\varepsilon_{k,\{r, \text{pref}(x, i), pr, p\}}@x.
\]

\[
\text{con}_{k,\{r,p\}}(\text{pref}(x, i)).
\]

\[
\text{pref}(x, i), ?l, ?pr, \langle \text{pref}(x, i), l + 1, pr \rangle)
\]

\[
\text{od } \varepsilon_{k,\{r, \text{“no-server-available”}\}}@x).
\]

Notice that \( H' \) creates a private link with the selected server, say \( s \), after having communicated the choice to the client. Notice also that even though \( x \) and \( h \) are connected via a permanent link, clients at \( x \) hangs on the input in \( C' \) when the load of the server is too high. Indeed, \( H' \) will reply with the message “no-server-available” to the incoming requests. This avoids that new requests are spawned to the servers.

\( S' \) gets a “newlink” request and establishes a connection but, differently from \( S \), it connects to \( h \), hence it does not need to know \( x \):

\[
S' \triangleright !(\langle \text{“newlink”, } ?r, ?p \rangle, \text{acc}_{k,\{r,p\}}(h), \text{...wait & execute...}
\]

\[
\varepsilon_{k,\{s, l, ?pr\}, \langle s, l - 1, pr \rangle}@h).
\]