Journal of Universal Computer Science, vol. 23, no. 9 (2017), 808-844
submitted: 3/5/17, accepted: 26/9/17, appeared: 28/9/17 © J.UCS

RESTful Services and Web-OS Middleware: a Formal Specification Approach

Mario Bravetti
(University of Bologna, Italy & FOCUS, INRIA, France
mario.bravetti@unibo.it)

Abstract: Web Operating Systems can be seen as an extension of traditional Operating Systems where the addresses used to manage files and execute programs (via the basic load/execution mechanism) are extended from local filesystem path-names to URLs. A first consequence is that, similarly as for traditional web technologies, executing a program at a given URL can be done in two modalities: either the execution is performed client-side at the invoking machine (and relative URL addressing in the executed program set to refer to the invoked URL) or it is performed server-side at the machine addressed by the invoked URL (as, e.g., for a web service). Moreover in this context, user identification for access to programs and files and workflow-based composition of service programs is naturally based on token/session-like mechanisms. We propose a middleware based on client-server protocols and on a set primitives for managing files/resources and executing programs (in the form of client-side/server-side components) in Web Operating Systems. The middleware is based on an extension of the REST architecture. In order to provide an unambiguous specification, we formally define the semantics of the proposed middleware by first introducing a process algebra for standard REST and then extending it to the whole middleware.

Key Words: Web Services, Web Operating Systems, Process Algebra
Category: D.3.1, D.3.3, F.3.1

1 Introduction

The widespread use of more and more powerful mobile devices, like tablets and smartphones, in addition to laptops, workstations, etc., has led to the need of exploiting the Internet as a repository for storing personal information and applications. The purpose is to be able to use them from any of these devices, not to lose them in the case one of these devices is destroyed or stolen and not have to re-install/re-configure them when such personal devices are changed: smartphones tend, e.g., to be changed much more frequently than laptops. These needs have led to the development of cloud computing which shifts all resource managing from local machines to a remote (set of) server(s) located somewhere in the Internet. Such a trend is, however, influencing much more deeply the way in which people use personal computers: browsers are, by far, the most used computer application and play, more and more, the role of operating systems, which allow the user to use applications and retrieve/store information. Another reason of this trend is related to the capability of (web) applications and information deployed in the Internet to be shared among several users, thus allowing for cooperation and enhanced communication. Examples are Google web applications and social networks like Facebook.

Being the computing experience and the evolution of computer languages/ technology more and more related to just the browser, this naturally leads to the idea of: from the one hand making its functionalities to become part of the operating system, from the other hand getting free from the more traditional (and heavy) way of installing and configuring applications. In essence, the shift from traditional operating systems to so-called Web Operating Systems (as e.g. [Google Chromium]) consists in changing from usage of local filesystem path-names to manage files and execute programs (via the basic load/execution mechanism) to usage of URLs. A first consequence is that, similarly as in traditional web technologies, executing a program at a given URL, can be done in two modalities: either the execution is performed client-side at the invoking machine (and relative URL addressing in the executed program set to refer to the invoked URL) or it is performed server-side at the machine addressed by the invoked URL (as, e.g., for a web service). From the viewpoint of application development and execution, Web-OS allows applications to be deployed anywhere in the Internet and used from any machine by exploiting a front-end/back-end philosophy typical of web applications. In Web-OSes a typical application will have a front-end consisting of several *front-end components*, i.e. a (graphical) user interface, which is executed client-side and a back-end which consists of several *back-end components* (services) remotely executed on the machine where the application is deployed (such a back-end may in turn exploit other resources like databases leading to the typical multi-tier architecture of classical web technologies). Notice that, thanks to the usage of relative URLs in *front-end components*, which are resolved relatively to the remote directory URL from which the component has been downloaded, they can access remote resources/services in their deployment environment independently from the location of their execution environment. For example a typical Web-OS application has two basic forms of file reading/saving: relative to the machine where it is deployed (default way of resolving relative addresses) or absolute/relative to the user machine.

In this paper we define the basic architecture and functionality of a middleware for a Web-OS which uses the mechanisms above as the “normal” way for executing and deploying programs (to be used by the local machine or by other machines over the Internet). Concerning the principles that guided the design of the Web-OS middleware, we have worked on detecting the basic mechanisms involved in the current technologies for front-end and back-end internet application development and on integrating them in a coherent set of primitives. In particular: (*i*) we implemented file/resource management with RESTful services [Fielding 2000] (the kind of services which is mostly used currently, for example in services exposed by Twitter and Facebook), (*ii*) we extended them with remote service execution by encompassing in a clean way (without breaking the REST philosophy) interface-based stateful services, that is services invoked by specifying an operation name and a parameter, similarly as for SOAP services [SOAP], and which make use of session and application data, (*iii*) we added a primitive for local execution based on downloading a front-end component and executing it at client-side (mechanism included in current Rich Internet Application technologies and client-side web technologies), and (*iv*) we endowed the middleware with mechanisms for managing tables of client sessions, similarly as done in a browser, and for storing session and application data, as done in a web server.

In order to unambiguously present the behavior/semantics of such a middleware and of applications it executes, we use a process algebraic [Milner 1999] approach to formally define the semantics of the middleware primitives and component deployment/execution. We do this in two steps. We first formalize standard RESTful services via process algebra: to the best of our knowledge the formalization that we present is the first one expressing in full the behavior of such services, in particular their service invocation mechanism based on URL pattern matching. We then extend such a process algebra to also include all the primitives and the mechanisms of the Web-OS middleware described above. The purpose of such a formal machinery is to define precisely and sistematically what has been explained and informally described in words in the paper. In particular, the formalization is detailed and realistic in expressing pattern based URL matching mechanisms (as, e.g., for Servlet invocation in a Tomcat server) and functioning of HTTP sessions. It can, thus, be exploited as an unambiguous reference for Web-OS middleware implementations. Moreover, it represents, to the best of our knowledge, the first attempt to formalize the execution model of an operating system with process algebra.

The paper is structured as follows. In Sect. 2 we introduce RESTful services and we present the syntax and semantics of the REST process algebra. In Sect. 3 we describe in detail the Web-OS middleware architecture and functionalities. In Sect. 4 we propose a realization of the Web-OS middleware that is based on an extension of RESTful services. In Sect. 5 we present a process algebra which extends the REST one by also including all such functionalities and we provide a modeling example. In Sect. 6 we make concluding remarks. This paper is an extended and revised version of [Bravetti 2014].

2 A Process Algebra for RESTful Services

The RESTful extension that we propose aims at adopting a uniform mechanism for managing both file/resources and interface-based services.

The needed mechanisms for dealing with files/resources are provided by so-called RESTful Web Services [Fielding 2000]. In RESTful Web Services the

HTTP protocol and HTTP request methods like GET, PUT and DELETE are used to manage the resource identified by the URL address of the request. In the simplest case the URL corresponds to a local directory/file in the destination machine (e.g. according to a mapping from the context part of the URL to a directory in the machine filesystem) and the invoked HTTP server performs the operation corresponding to the request method: read, write (create or modify) and delete.

However, in RESTful Web Services, URL addresses can be used to represent resources different from directory/files, e.g. to directly manage records of a database table. The idea is to manage addressed resources via a service which is associated to a *pattern* in the form "$\backslash pathname \backslash *$": all requests to URLs whose pathname matches the pattern (where "$*$" stands for any possible suffix) cause an execution of the associated service which possesses an operation for each request method.[1] The operation receives the "suffix", that in the invoked URL replaces "$*$", as a parameter and uses it as an identifier for the resource (e.g. a database record) on which it actually operates by executing resource specific access code. Notice that resource URLs managed by a service can be of two kind: the resource collection kind, in the case the URL ends with "$\backslash$", and the single resource kind, in the opposite case. On an URL of the former kind it is possible to also use the POST HTTP request method with the following intended behavior: a subresource (a resource whose address is "$URL\, id$" or "$URL\, id\backslash$" for some "id") is created whose content is passed as the body of the request (as for PUT) and where id is a fresh string identifier generated by the server [Fielding 2000].[2]

Example 1. The (relative) URL "$\backslash persons\backslash$", denoting a resource collection, may represent a database of persons managed by a service (which internally accesess a real database) associated with pattern "$\backslash persons\backslash *$". A POST call at URL "$\backslash persons\backslash$" is used to create a new person. An URL is associated to the created person in the form "$\backslash persons\backslash id$", with "id" being a fresh identifier (not yet created for the URL "$\backslash persons\backslash$") that is returned to the caller. PUT, GET and DELETE calls are then used on URLs of kind "$\backslash persons\backslash id$" (for some "$id$"), representing single resources, to modify, read or delete such a resource.

2.1 Matching-based Invocation

Before presenting the REST process algebra, we describe in more details the functioning of matching based invocation.

[1] For denotational convenience, in this paper we use backslashes in URLs instead of slashes, because we use slashes to represent replacements.

[2] Notice that, in RESTful Web Services, it is not mandatory for services to manage several resources: the pattern "$\backslash pathname\backslash name$" (or just "$\backslash name$"), not including "$*$", can alternatively be used, which matches just a single resource.

From the HTTP server side, when a request is received, the destination URL is checked for matching with patterns of deployed services and, in case of successful matching, the longest matching pattern is selected (deployment of multiple services with the same pattern is not allowed) and the requested operation (that corresponding to the HTTP request method) of the associated back-end component executed.[3]

In the case the request in *not matched* by any pattern a *default behaviour* is performed: for PUT, GET and DELETE HTTP requests the default behaviour is to perform the corresponding action on the local filesystem at the path corresponding to the resource URL (that is a direct creation/modification, read and deletion, respectively, of the file or directory at that path); for a POST HTTP request to the resource collection URL, the default behaviour is to generate a file (or directory) with a fresh name in the local filesystem directory corresponding to the destination URL containing the body of the request and responding with the generated name. Notice that, the default behavior of a PUT at a URL requires, in case that no resource exists at URL, that the resource at the parent URL exists; similarly, the default behavior of a DELETE at the URL of an existing resource requires that there are no resources existing at any child URL.

2.2 Syntax

We will now present the process algebra for REST, which is devised as an extension of pi-calculus [Milner et al. 1992, Milner 1999]. We use $x, y, \ldots$ to denote generic *names* over a set $\mathcal{N}$. Names are used both to represent channal based communication as in the pi-calculus and to build *pathnames* and *URLs*, e.g. names like $persons$ and id of Example 1 that are also assumed to be in $\mathcal{N}$. Moreover, we use l, l' to denote *locations* over a set $\mathcal{L}$, representing application contexts, i.e. a location l identifies both a server (IP+port) and one of its application contexts (the initial part of a url).

Names of $\mathcal{N}$ are used to build (relative) pathnames as follows. A *directory relative path*, ranged over by *drpath*, is taken to be:

$$drpath ::= x \backslash drpath \mid \varepsilon$$

where "ε" is the empty string. For example "$persons \backslash db \backslash$" is a *drpath*. Also "$\varepsilon$" is a *drpath* (being *drpath* a relative path, it is of course relative to a directory). Then, a generic *relative path*, ranged over by *rpath*, representing both collection and single resources, is given by:

$$rpath ::= drpath \mid drpath\ x \qquad .$$

For example both "$persons \backslash db \backslash$" and "$persons \backslash db \backslash id$" are a *rpath*. Finally, a *URL*, ranged over by *url*, is a context location l followed by a pathname. We consider contexts to have a special directory called **exec** that we will use to

[3] This is analogous to the mechanism for determining the servlet to be executed in a Tomcat server.

deploy services, i.e. their code and their associated info, as the pattern they manage (see Example 1). Formally, URLs are defined as:

$$url ::= l\backslash[\mathbf{exec}\backslash]rpath$$

where we use

$$[\mathbf{exec}\backslash] ::= \mathbf{exec}\backslash \mid \varepsilon$$

to denote the optional usage of the special **exec** context directory (we assume $\mathbf{exec} \notin \mathcal{N}$). For example both "$l\backslash persons\backslash db\backslash$" and "$l\backslash\mathbf{exec}\backslash n\backslash$" are a url.

The process algebra represents the Internet as a *network* N of resources R deployed at some URL url: "$[R]_{url}$". Formally, we have:

$$N ::= [R]_{url} \mid N \parallel N \mid (\nu x)N$$
$$R ::= v \mid P$$

Resources R can be either values v (representing typed files) or programs under execution P (representing processes/threads in memory). Running programs P are only present during execution and they need to be considered only when presenting the process algebra semantics (see following Section 2.3). The "$\parallel$" and "(νx)" parallel and restriction operator have the same meaning as in pi-calculus. In particular, (νx) is used to define the scope of a name, i.e. it encloses the (deployed) resources R that have access to it.

For a network to be *well-defined* the following condition must be satisfied: if it includes $[R]_{burl\backslash x}$ or $[R]_{burl\backslash x\backslash}$ for some name x and string $burl$ that is not simply in the form "l" or "$l\backslash\mathbf{exec}$", then it must also include some resource R' at the parent URL "$burl\backslash$", i.e. the collection resource $[R']_{burl\backslash}$. The constraints about PUT and DELETE HTTP methods, discussed at the end of Section 2.1 (existence of a parent resource and non-existance of a child resource, respectively) and enforced in the process algebra semantics, guarantee that well-definedness of networks is preserved during execution. In the following we will consider well-defined networks only.

A *system* is a (well-defined) network not including process/thread resources P. A system represents a network where resources have been deployed but there is no service in execution. Making an analogy with object oriented programming (with services being like classes): a system is an object oriented program; a generic network is the snapshot of a program in execution.

Part of a system specification is also a predicate, denoted by $cond(url)$, over directory URLs, i.e. the argument url is assumed to be in the form $burl\backslash$ for some string $burl$. Such a predicate establishes whether the default behaviour of a POST at a given directory URL creates subresources in the form of single or collection resources (i.e. directories ending with "$\backslash$"). For a given directory URL $burl\backslash$, if $cond(burl\backslash)$ is *false* then, whenever a POST on $burl\backslash$ is called, a single resource at URL $burl\backslash id$ is created, with id being a fresh name; if, instead, $cond(burl\backslash)$ is *true* then a collection resource at URL $burl\backslash id\backslash$ is created, with id fresh. $cond(url)$ is assumed to be any predicate over directory URLs such that, for all contexts l: $cond(l\backslash) = true$ and $cond(l\backslash\mathbf{exec}\backslash) = true$. This because: obviously we cannot see context locations as collections of single items; and we represent services deployed under the special **exec** directory as collection resources (as we will see, at run time we represent threads executed by services as their subresources).

2.2.1 Values

Values v are abstract representations of data/code: in a real system they can actually be contained (serialized) in typed files in the format corresponding to their type, e.g. XML or "class" java bytecode. We consider: the empty string ε, used, e.g., to represent resources that exist but do not contain any significant information (as for a collection resource $[]_{l\backslash persons\backslash db\backslash}$, whose members $[R]_{l\backslash persons\backslash db\backslash id}$ are allowed to be in a network N only if the collection resource itself is also in N, due to the network well-definedness condition above); primitive values $pval$, which should at least allow us to represent successful and erroneous response from a command request and numbers; names x; deployed services $\langle D\rangle^{pat}$, with D being a declaration of the code of their operations, and pat being the associated pattern they manage (see Example 1); and references ref (defined formally in the next Section 2.2.2) to URLs, e.g. by direct use of a url or by means of a relative path. Formally, we have:

$$v ::= \varepsilon \mid pval \mid x \mid ref \mid < v, \ldots, v > \mid \langle D\rangle^{pat}$$

$$pval ::= \mathbf{ok} \mid \mathbf{err} \mid num \mid \ldots$$

We now formally define deployed services $\langle D\rangle^{pat}$.

A service pattern, ranged over by pat, is taken to be:

$$pat ::= \backslash drpath\ x \mid \backslash drpath\ x\backslash *$$

For example both "$\backslash person$" and "$\backslash persons\backslash *$" are a pat (remember that $drpath$ can be ε).

Declaration D contains the code of service operations associated to REST commands. Services are guaranteed to possess a behaviour for all REST methods by assuming that, in the case a command is not explicitly defined in the declaration D, the *default behaviour* (described in Section 2.1 for the case of unmatched commands) is performed. A REST command, ranged over by **com**, is:

$$\mathbf{com} ::= \mathbf{put} \mid \mathbf{get} \mid \mathbf{delete} \mid \mathbf{post}$$

We use Com to denote the set of REST commands **com**. Formally, D is a partial function from Com to pairs composed by a formal parameter variable x and a term E representing the code of the operation. Definitions in D are represented as $\mathbf{com}(x) \stackrel{\Delta}{=} E$.

Moreover, we assume D to be such that, if $\mathbf{com}(x) \stackrel{\Delta}{=} E \in D$, for $\mathbf{com} \in \{\mathbf{get}, \mathbf{delete}\}$, then parameter x is a dummy one and it is *not used* by the code E: this because, as we will see, when **get** and **delete** commands are invoked no actual parameter is passed (due to their intended meaning). In general, in the following, we will just use $\mathbf{com} \overset{\Delta}{=} E$, to stand for a definition $\mathbf{com}(x) \overset{\Delta}{=} E$ where parameter x is not used by the code E.

2.2.2 Code of Operations

We now present the syntax of code E we use to define service operations. The basic elements used in code are REST command calls, which are composed by standard operators such as sequencing, choice and looping. Before detailing the syntax of terms E we need some preliminary definitions.

In service operation code we use relative paths that include the special keyword <**ipath**> to stand for the internal path, i.e. the URL suffix identifying the resource on which a service is called (replacing the "*" in the matching pattern). References ref represent the possible ways of expressing resource addresses in a REST command call: absolute, i.e. a URL; root-relative, i.e. starting with a backslash and referring to the application context location l (the root); or relative. Formally, we have:

$$ref ::= url \mid \backslash[\mathbf{exec}\backslash]rpath \mid rpath_s$$

where $rpath_s$ is defined by:

$$rpath_s ::= rpath \mid drpath\ \ \texttt{<}\mathbf{ipath}\texttt{>}\ \ rpath$$

We also need to introduce expressions. An expression e includes values v possibly combined with functional operators (that we will not explictly detail here) and returns a value v. Similarly, a boolean expression be includes values v possibly combined with functional operators and returns a boolean ($true$ or $false$). In the semantics we will denote evaluation of expressions e/be, such that all variables (unbound names) have been already instantiated, with $\mathcal{E}(e)/\mathcal{E}(be)$.

Every REST command call has a flag, called "in" that denotes whether it should just work internally, $in = \mathcal{I}$, or not, $in = \varepsilon$ (in this case the flag is just omitted when writing the command). Internal commands are used to explicitly invoke the default command behaviour (the behaviour it has when it is not matched, see Section 2.1), thus modeling a direct local resource access. In this case the command call refers to the resource with a relative address.

The syntax of terms E representing operation code is as follows.

$$\begin{aligned} E ::= {} & x = \mathbf{put}^{in}_{ref}\, e.E \mid x = \mathbf{get}^{in}_{ref}.E \mid x = \mathbf{delete}^{in}_{ref}.E \mid x = \mathbf{post}^{in}_{ref}\, e.E \mid \\ & x = e.E \mid \overline{x}\, e.E \mid x(y).E \mid (\nu x)\, E \mid \mathbf{spawn}\ E\,.E \mid \mathbf{if}\, be\ \ \mathbf{then}\, E\ \mathbf{else}\, E \mid \\ & recX.E \mid X \mid \underline{0} \mid \mathbf{return}\ e \end{aligned}$$

where, for command calls $x = \mathbf{com}^{in}_{ref} e.E$ (with e being omitted in the case of **com** being **get** or **delete**), we have that $in = \mathcal{I}$ implies $\exists\, rpath_s : ref = rpath_s$.

Besides command calls $x = \mathbf{com}^{in}_{ref} e.E$, with (freshly created) variable x being instanciated with the value returned by the command, we consider the following operators. $x = e.E$ denotes a (freshly created) variable x that is instanciated with the value computed by expression e. $\overline{x}\, e.E$ and $x(y).E$ are the pi-calculus x channel based output and input: they are used to represent blocking communication between threads, e.g., to model blocking behaviours based on locking mechanisms within the same server. $(\nu x)\, E$ is the pi-calculus (freshly created) name x binder that is used, e.g., for the generation of new channels. **spawn** $E_1\, .E_2$ represents spawning of E_1, i.e. generation of another thread (represented, as we will se, as a subresource of the thread in execution) dedicated to executing E_1, and immediate continuation with E_2 in the current thread. **if** *be* **then** E_1 **else** E_2 represents a choice based on a boolean condition. $recX.E$ and X are used to express recursion, i.e. looping behaviours (where $recX.E$ is repeated whenever X is reached inside E). $\underline{0}$ represents, as usual, a completed (empty) behaviour: it is used, e.g., to terminate a spawned thread. Finally, **return** e completes the execution by returning a value to the caller. We assume that in a term E recursion variables X always occur in the scope of a $recX$ operator binding them.

Example 2. By using **spawn** and a dedicated, freshly generated, pi-calculus channel we can model an asynchronous command invocation inside a term E as follows:

$$(\nu x)\,(\ \mathbf{spawn}\ (y\!=\!\mathbf{get}_{url}\, .\, \overline{x}\, y\, .\, \underline{0})\ .\, E'\ )$$

where E' performs some computation (which happens in parallel w.r.t. the **get** command execution) and then reads the value returned by **get** with a $x(z).E''$ operator.

Notice that operators $x\!=\!\mathbf{com}^{in}_{ref} e.E$, $x\!=\!e.E$, $y(x).E$, $(\nu x)\, E$ and declarations $\mathbf{com}(x) \triangleq E$ (that may occur inside values $\langle D\rangle^{pat}$ used by an expression e), they all act as *binders* for the name x of $\mathcal{N}$. We define the set of free $\mathcal{N}$ names of E, denoted by $fr(E)$, as the set of names x that occur in E and are not bound by one of such operators. Therefore, if E is the code of an operation with a formal parameter x, i.e. $\mathbf{com}(x) \triangleq E$, then E uses the parameter x whenever $x \in fr(E)$.

Example 3. Let us consider the REST interface to a database of persons presented in Example 1. The formal specification of such example is the system N defined by

$N = [\langle D_{init}\rangle^{\backslash init}]_{l_{init}\backslash\mathbf{exec}\backslash n\backslash} \parallel [\langle D\rangle^{\backslash persons\backslash *}]_{l\backslash\mathbf{exec}\backslash n\backslash} \parallel [\,]_{l\backslash persons\backslash} \parallel [\,]_{l\backslash deleted\backslash}$

where the database manager is represented by the declaration D associated, at location l, to the pattern $\backslash persons\backslash *$, while the declaration D_{init} associated, at location l_{init}, to the pattern $\backslash init$ is meant to include, in the definition of the **put** command, the code executed when the system is started: as we will see in the following Section 2.3 about semantics, the system is started by creating a resource at a given *url* with a **put** command. Here we take this *url* to be $l_{init}\backslash init$

and D_{init} (the declaration of the service code which matches the $l_{init}\backslash init$ URL) to just include the following definition of **put**:

$$\begin{aligned}\mathbf{put} \overset{\Delta}{=} & \, \mathbf{put}_{l\backslash persons\backslash} \,.\, x_1 = \mathbf{post}_{l\backslash persons\backslash} < John, Smith > \,. \\ & x_2 = \mathbf{post}_{l\backslash persons\backslash} < Mark, Johnson > \,.\, m = \mathbf{get}_{l\backslash persons\backslash x_1} \,. \\ & \mathbf{delete}_{l\backslash persons\backslash x_1} \,.\, \mathbf{post}_{l\backslash deleted\backslash} m \,.\, \mathbf{return}\end{aligned}$$

That is, we create the database of persons, we add two persons to the database, we then get back the data (value $< John, Smith >$) of the first person we added, we delete it from the database and we add the data of the deleted person in the collection resource $l\backslash deleted\backslash$ meant to collect deleted items. Notice that, when writing commands (e.g., for the first $\mathbf{put}_{l\backslash persons\backslash}$ command call above) we omit explicitly writing "ε" arguments (expressions composed of an "ε" value only) and "ε" superscripts (flag "*in*" being "ε"). Finally, the declaration D, containing the code of the person database manager is defined as follows. For each $\mathbf{com} \in Com$, D includes the following definition of $\mathbf{com}(x)$:

$$\mathbf{com}(x) \overset{\Delta}{=} \mathbf{com}^{\mathcal{I}}_{db\backslash <\mathbf{ipath}>} x.return$$

where x is omitted in the case of **com** being **get** or **delete**. This means that we abstractly represent database operations by commands on the internal collection URL $l\backslash persons\backslash db\backslash$.

Finally, we assume function *cond* (which is part of the system specification, see end of Section 2.2) to be such that $cond(l\backslash persons\backslash db\backslash) = false$ and $cond(l\backslash deleted\backslash) = false$, i.e. unmatched **post** commands at $l\backslash persons\backslash db\backslash$ and $l\backslash deleted\backslash$ generate subresources which are single items and not collections (i.e. their URL does not terminate with "\"). This is as expected since we are managing single persons.

According to the above system specification, at run time, the following network behavior occurs. When the database of persons is initially created by "$\mathbf{put}_{l\backslash persons\backslash}$", the command is matched by the person database manager declared by D, which correspondingly creates resource "$[\,]_{l\backslash persons\backslash db\backslash}$" in the network: in this case the value of "$< \mathbf{ipath} >$", i.e. the URL $l\backslash persons\backslash$ suffix replacing the "*" in the matching pattern, is "ε". Similarly the two subsequent "**post**" commands are matched, with "$< \mathbf{ipath} >$" being "ε" for both such matchings. Thus, according to the code in "D": two (sub)resources "$[< John, Smith >]_{l\backslash persons\backslash db\backslash id1}$" and "$[< Mark, Johnson >]_{l\backslash persons\backslash db\backslash id2}$", for some fresh names "*id*1" and "*id*2" (which are stored in variables x_1 and x_2, respectively), are created. The subsequent "**get**" and "**delete**" commands are again matched by the service declared by D, but this time with "<**ipath**>" being "*id*1" for both of them. They, thus, read the content of the resource at URL $l\backslash persons\backslash db\backslash id1$ (storing it in variable m that gets value "$< John, Smith >$") and, then, delete such a resource (which is removed from the network). Finally, the (unmatched) command "$\mathbf{post}_{l\backslash deleted\backslash} m$" creates a resource "$[< John, Smith >]_{l\backslash deleted\backslash id}$", for some fresh name "*id*": being it unmatched it just performs its default beaviour, i.e. it acts directly on the network (without causing code of a matching service to be executed).

The above described network behaviour will be defined formally in the following section about semantics.

2.3 Semantics

In order to present the semantics we need to preliminary define terms P representing programs (service operations) in execution. Syntax of terms P is defined as that of terms E with two restrictions: command subscripts ref are just url subscripts (due to syntactical relative URL resolution when operations are invoked) and the command **return** is not present (because **return** is encoded as communication on a private channel). Moreover we assume function $fr()$ over terms P (expressions e), yielding the set of free names of $\mathcal{N}$ in P (e, respectively), to have the same definition as for terms E (recall that an expression e can be simply a value v). Similarly, $fr()$ is also defined over networks N: it is defined as for terms E, where in addition we consider (νx) operators in the syntax of N as binders for names $x \in \mathcal{N}$. Consistently we assume $fr()$ over strings (e.g. part of URLs or patterns) to simply yield all the names of $\mathcal{N}$ included in the string.

Implicitly based on the above notion of free names, we will use the standard notation $P\{v/x\}$ to denote syntactical replacement of all free occurences of name x in term P with value v. As standard, we assume replacements to be performed in such a way that new names introduced by replacing elements are not "captured" by binders (by applying renaming to bound names if necessary). We also assume the replacement to yield $\underline{0}$ in the case it is not possible to perform it: this happens in case variable x is expected to contain a certain type of value, e.g. a name to be used as a pi-calculus channel (with an operator $\overline{x}\, e.P$ or $x(y).P$), but it is replaced by a value that is not of that type (in real programming languages this could be avoided by requiring code E of services to type check before putting them in execution). Finally, we also use the similar standard notation $P\{P'/X\}$ to denote replacement of all free occurrences of recursion variables X in P (i.e. variables X that are not in the scope of a $recX$ operator) with term P'.

Before presenting the semantics, we need to introduce some notation concerning parts of URLs that is convenient for expressing semantic rules. We use *path* to denote the path information that can occur in an *url* after a context l, i.e. $path$ ::= $\backslash$[**exec**$\backslash$]$rpath$, and *burl* (basic URL) to denote the part of a directory URLs obtained by omitting the final "\": that is a *burl* is a string such that *burl*\ is an url.

The semantics is defined in *reduction* style along the lines of that, for the pi-calculus, in [Milner 1999]. As in [Milner 1999] the semantics is based on a structural congruence relation over process algebra terms, i.e. networks N. Such a relation is defined by the set of laws presented in Table 1. Then, assumed

$\mathcal{R}$ (resource sets) to denote terms N that do not include $(\nu n)N$ subterms and $[R]_{url} \in \mathcal{R}$ to mean that $\mathcal{R}$ includes $[R]_{url}$ among its parallel terms, reduction transitions $\longrightarrow$ are defined by the rules in Tables 2, 3, 4, with the help of auxiliary transitions for commands, denoted by $\longrightarrow_c$. In the laws and rules we often need to put conditions concerning the set of free names of $\mathcal{N}$ used by a list of their elements (e.g. networks, URLs, terms E or P, ...): in order to not burden the notation we simply assume $f(el_1, \ldots, el_n)$ to stand for $f(el_1) \cup \cdots \cup f(el_n)$. In the following we present each table and we define the auxiliary functions it uses.

Table 2 includes the semantic rules defining transitions $\longrightarrow$ for all the process algebra operators: those for commands are defined in terms of transitions $\longrightarrow_c$ that are defined by the other two tables. We just comment the non standard ones. Rules for command calls establish, based on $\longrightarrow_c$ transitions, if the command can be called and in the negative case (non existance of a $\longrightarrow_c$ transition, denoted by $\not\longrightarrow_c$) return the special value **err**. Rule for **spawn** $P.Q$ starts the execution of P in parallel and continues immediately by executing Q.

Tables 3 and 4 include the semantic rules defining transitions $\longrightarrow_c$ in the case of: unmatched and internal commands (i.e. commands whose reference url does not match any pattern and commands with subscript $in = \mathcal{I}$); and matched ones, respectively.

Table 3 defines the behavior of **put**, **get**, **delete**, **post** as the standard one (see previous sections) unless $in = \mathcal{I}$ or url is matched by a pattern pat of some declaration $\langle D\rangle^{pat}$ (residing at the special **exec** directory of the context l of url) that redefines its behavior. The latter condition is checked by means of the auxiliary function $pats_{url}(\mathcal{R})$ that evaluates the set of patterns pat matching url that occur in service declarations $\langle D\rangle^{pat}$ contained in $\mathcal{R}$, i.e. it is defined by

$$pats_{l\,path}(\mathcal{R}) = \bigcup_{[\langle D\rangle^{pat}]_{l\backslash\mathbf{exec}\backslash m\backslash} \in \mathcal{R} \,\wedge\, path \in pat} \{pat\}$$

where $path \in pat$ stands for $\exists\, rpath : pat\{rpath/*\} = path$. Such a function is considered together with function max that evaluates the pattern pat obtained as the maximum of a set of patterns according to the ordering $\leq$ on patterns defined by: $pat \leq pat'$ if $pat\{s/*\} = pat'$ for some string s. Finally the condition makes use of function $coms(R)$ that yields: the set of commands **com** defined in D, if $R = \langle D\rangle^{pat}$ for some pat; $\emptyset$ otherwise. In the rules for **put**, **get**, **delete** and **post** the following auxiliary functions are also used. $urls(\mathcal{R})$ yields the set of URLs of $\mathcal{R}$ resources, i.e. $urls(\mathcal{R}) = \{url \mid [R]_{url} \in \mathcal{R}\}$. Function $d(url)$ returns the url' of the resource collection (directory) directly including the given resource, that is, such that for some n, url is $url'n$ or $url'n\backslash$ (it returns ε if url is in the form $l\backslash$). Function $id(url)$ returns the string obtained by removing the final "\" from url, in case it is present; it is also extended to set of url addresses, returning the set of strings obtained by applying $id(url)$ to every url. In the rule for **delete**, we make use of relation $url < url'$ that holds if url is strictly a prefix of url'; and, in the rule for **post**, of the notation $\{\backslash\}^{cond}$, with $cond$ being a boolean value, that is assumed: to yield "$\backslash$" if $cond$ is $true$; to yield "ε", i.e. an empty string, otherwise. Finally, we define sets of initial locations in which it is allowed to directly generate new subitems via **post** (In_g) and to create a subitem via **put** (In_p): $In_g = \{l\backslash, l\backslash\mathbf{exec}\backslash \mid l \in \mathcal{L}\}$, $In_p = \{\varepsilon, l\backslash \mid l \in \mathcal{L}\}$. In particular, the presence of ε in In_p allows for resources at new locations (URL $l\backslash$) to be created during execution.

Table 4 defines, instead, the behavior of a command **com** when its url is matched by a pattern pat of some declaration $\langle D\rangle^{pat}$ (residing at the **exec** directory of the context l of url) that redefines its behavior. Such a condition is checked, in the rule in Table 4, with the help of the auxiliary predicate $match(url, pat, \mathcal{R})$ that holds if url matches pat and there is no other service declaration in $\mathcal{R}$ associated with a strictly longer matching pattern, i.e. it is defined by

$$match(l\,path, pat, \mathcal{R}) = path \in pat \wedge pat \geq max\ pats_{l\,path}(\mathcal{R})$$

In the case the above matching condition holds, the rule in Table 4 performs the invocation of the service operation corresponding to the called command (the command parameter e is assumed to be ε in the case of **get** and **delete**): its code E contained in D is executed in a new, freshly generated, thread t. The operation call and its response are managed by two freshly generated pi-calculus channels and the needed replacements inside E are performed, i.e. relative addressing is resolved and the keyword "<**ipath**>" is substituted with the part of the URL invoked by the caller that matches $*$ in the pat pattern. In the rule we assume replacement $E\{\overline{z}/\mathbf{return}\}$ to mean that all occurrences of "$\mathbf{return}\,e$" terms in E (occurring inside any binder) are replaced by "$\overline{z}\,e.\,\underline{0}$". Similarly replacements θ_1 and θ_2 syntactically replace all occurrences of elements to the right of "/" (inside any binder) with the corresponding elements to the left. Moreover the rule also uses the following auxiliary functions. Similarly as for function $d(url)$ previously defined, when a pattern pat is considered $d(pat)$ returns the $path$ such that pat is $path\,*$ or $path\,n$ for some n. Function $url(l\,path\backslash, ref)$ performs the expected evaluation of an (absolute) URL url starting from the (absolute) URL $l\,path\backslash$ of a collection resource and the, possibly, relative URL expressed by ref, that is: either $url = ref$, if ref starts with some location l; or $url = l\ ref$, if ref starts with "$\backslash$"; or $url = l\,path\backslash ref$, otherwise. Finally, the expression $path - pat$ computes the $path$ suffix $rpath$ matching $*$ of pat (yields ε if there is no $*$) in the expected way, that is such that $path = pat\{rpath/*\}$.

Notice that the negative premise in Table 2 does not cause bad definedness of the operational semantics. This is because, in the rule for matched commands, the capability of a term of performing some auxiliary transition depends only

$$
\begin{array}{rcll}
N_1 \parallel N_2 & \equiv & N_2 \parallel N_1 & \\
(N_1 \parallel N_2) \parallel N_3 & \equiv & N_1 \parallel (N_2 \parallel N_3) & \\
((\nu x)N_1) \parallel N_2 & \equiv & (\nu x)(N_1 \parallel N_2) & x \notin fr(N_2) \\
{[(\nu x)P]}_{url} & \equiv & (\nu x)[P]_{url} & x \notin fr(url)
\end{array}
$$

Table 1: Congruence rules.

on the matchnig condition to be satisfied and *not* on the existence of some reduction transition $\longrightarrow$ (which could in turn depend on auxiliary transitions). This is because the term $N \parallel \mathcal{R}$ considered in the premise can always perform a reduction transition (i.e. the synchronization between $\overline{z}$ and z). Therefore, formally, operational semantics is well-defined in that the inference of transitions can be stratified (see, e.g., [Groote 1993]).

Definition 1. Let N be a well-defined system. We use $[\![N]\!]_{url}$ to denote the semantics of N when executed by creating, with a **put** command, an empty (ε) resource at *url*. $[\![N]\!]_{url}$ is defined as the $\longrightarrow$ transition system where the initial network is taken to be $N \parallel [P]_l$ with $P = \mathbf{put}_{url}$ and l being any location not occurring in N.

The definition above can be understood by considering again the analogy with object oriented programming: the code for **put**, declared by the service of N that (according to its associated pattern) matches *url*, acts as the body of the *main* method. For instance, system N of Example 3 is meant to be executed by taking *url* to be $l_{init}\backslash init$: it contains the deployed service $[\langle D_{init}\rangle^{\backslash init}]_{l_{init}}$ that matches such an *url* and whose declaration D_{init} includes the code for **put** that is meant to act as the main method. Such an execution is therefore formally represented by $[\![N]\!]_{l_{init}\backslash init}$.

Example 4. We now show how we formally represent the execution of system N of Example 3, that is its semantics $[\![N]\!]_{l_{init}\backslash init}$. According to Definition 1 the initial network is $N \parallel [P]_l$, with $P = \mathbf{put}_{l_{init}\backslash init}$, where l is any location not occurring in N.

An initial reduction is performed, according to the rule for matched commands in Table 4, representing the execution of the **put** command. Such a reduction leads to network $N \parallel N'$, where N' includes terms: $[P']_l$, where P' is just waiting for the called **put** to return on a private channel; and $[P'']_{l_{init}\backslash \mathbf{exec}\backslash n\backslash t\backslash}$ with t fresh, where P'' is the code of the **put** command as defined inside D_{init} (where relative and internal addressing has been resolved and the return command has been replaced by communication on the private channel above).

$$\frac{N_1 \equiv N \wedge N \longrightarrow N'}{N_1 \longrightarrow N'} \qquad \frac{N \longrightarrow N'}{(\nu x)N \longrightarrow N'}$$

$$\frac{[x = \mathbf{com}^{in}_{url}\hat{e}.P]_{url'} \parallel \mathcal{R} \longrightarrow_c \mathcal{R}'}{[x = \mathbf{com}^{in}_{url}\hat{e}.P]_{url'} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}$$

$$\frac{[x = \mathbf{com}^{in}_{url}\hat{e}.P]_{url'} \parallel \mathcal{R} \not\longrightarrow_c}{[x = \mathbf{com}^{in}_{url}\hat{e}.P]_{url'} \parallel \mathcal{R} \longrightarrow [P\{\mathbf{err}/x\}]_{url'} \parallel \mathcal{R}}$$

$$[\mathbf{spawn}\, P.Q]_{burl\backslash} \parallel \mathcal{R} \longrightarrow [Q]_{burl\backslash} \parallel ((\nu t)[P]_{burl\backslash t\backslash}) \parallel \mathcal{R} \qquad t \notin fr(P, burl)$$

$$[\mathbf{if}\, be\, \mathbf{then}\, P\, \mathbf{else}\, Q]_{url} \parallel \mathcal{R} \longrightarrow [P]_{url} \parallel \mathcal{R} \qquad \mathcal{E}(be) = true$$

$$[\mathbf{if}\, be\, \mathbf{then}\, P\, \mathbf{else}\, Q]_{url} \parallel \mathcal{R} \longrightarrow [Q]_{url} \parallel \mathcal{R} \qquad \mathcal{E}(be) = false$$

$$[recX.P]_{url} \parallel \mathcal{R} \longrightarrow [P\{recX.P/X\}]_{url} \parallel \mathcal{R}$$

$$[\overline{x}\, e.P]_{url} \parallel [x(y).Q]_{url'} \parallel \mathcal{R} \longrightarrow [P]_{url} \parallel [Q\{\mathcal{E}(e)/y\}]_{url'} \parallel \mathcal{R}$$

Table 2: Basic rules.

$[P'']_{l_{init}\backslash\mathbf{exec}\backslash n\backslash t\backslash}$ is thus a new thread dedicated to the execution of such code, which now is the only thread in execution.

In a subsequent reduction the command $\mathbf{put}_{l\backslash persons\backslash}$ is executed, which is matched by $\langle D\rangle^{\backslash persons\backslash *}$ and, similarly as for the execution of the previous **put** command above, it causes the creation of a new resource that starts executing the code of the **put** command as defined inside D. Being now the latter the only active thread, it causes the following two reductions: the creation of the new resource $[\,]_{l\backslash persons\backslash db\backslash}$ and the return via communication on a private channel that reactivates the code of the **put** of D_{init}.

With a similar matching scheme, resource $[< John, Smith >]_{l\backslash persons\backslash db\backslash n_1}$, with "$n_1$" being a fresh name, is then added and replacement n_1/x_1 is performed inside the code of the **put** of D_{init}. The last sequence is then repeated similarly with the creation of $[< Mark, Johnson >]_{l\backslash persons\backslash db\backslash n_2}$ and the replacement n_2/x_2. Subsequently, we have that **get** (apart from thread creation) just performs replacement $< John, Smith >/m$ and **delete** removes the resource $[< John, Smith >]_{l\backslash persons\backslash db\backslash n_1}$. Finally the last command of the code of the **put** of D_{init}, before the final return, causes the creation of resource $[< John, Smith >]_{l\backslash deleted\backslash n}$ for some fresh name n. The last network reduction is the execution of the mentioned return command which consists in a communication to the caller via a private channel.

$$\begin{array}{ll}
[x = \mathbf{put}^{in}_{url} e.P]_{url'} \parallel [v']_{url} \parallel \mathcal{R} & \\
\longrightarrow_c [P\{\mathbf{ok}/x\}]_{url'} \parallel [\mathcal{E}(e)]_{url} \parallel \mathcal{R} & \\
[x = \mathbf{put}^{in}_{url} e.P]_{url'} \parallel \mathcal{R} & d(url) \in urls(\mathcal{R}) \cup In_p \wedge \\
\longrightarrow_c [P\{\mathbf{ok}/x\}]_{url'} \parallel [\mathcal{E}(e)]_{url} \parallel \mathcal{R} & id(url) \notin id(urls(\mathcal{R})) \\
[x = \mathbf{get}^{in}_{url}.P]_{url'} \parallel [v]_{url} \parallel \mathcal{R} & \\
\longrightarrow_c [P\{v/x\}]_{url'} \parallel [v]_{url} \parallel \mathcal{R} & \\
[x = \mathbf{delete}^{in}_{url}.P]_{url'} \parallel [v]_{url} \parallel \mathcal{R} & \nexists url'' \in urls(\mathcal{R}) : url'' > url \\
\longrightarrow_c [P\{\mathbf{ok}/x\}]_{url'} \parallel \mathcal{R} & \\
[x = \mathbf{post}^{in}_{burl\backslash} e.P]_{url'} \parallel \mathcal{R} & burl\backslash \in urls(\mathcal{R}) \cup In_g \wedge \\
\longrightarrow_c (\nu n)([P\{n/x\}]_{url'} \parallel & n \notin fr(P, e, burl, url') \\
\qquad [\mathcal{E}(e)]_{burl\backslash n\{\backslash\}^{cond(burl\backslash)}}) \parallel \mathcal{R} &
\end{array}$$

additional condition for each rule:

$in = \mathcal{I} \vee \nexists [R]_{url''} \in \mathcal{R} : pats_{url}([R]_{url''}) = \{max\ pats_{url}(\mathcal{R})\} \wedge \mathbf{com} \in coms(R)$

where **com** is the rule command and $url = burl\backslash$ for the **post** rule

Table 3: Rule for auxiliary transitions of unmatched and internal commands.

3 Web-OS Middleware

We present, in Figure 1, the proposed Web-OS architecture. The upper part of the figure shows front-end and back-end components in execution, i.e. with operations actually under execution or waiting for operation calls. As explained in the introduction, each front-end component is associated to a "base URL" for relative addressing with respect to the URL it was downloaded from.

Concerning the lower part, the language/technology independent middleware receives (via language dependent APIs) request to execute commands from front-end and back-end components. Such commands are all invoked specifying a URL (relative or absolute) on which they must act and follow the request-response schema (as in HTTP), sending and receiving typed files, where, e.g., standard mime-types can be used. Such files can represent both transmission of parameters/data in a programming language independent manner (via e.g. XML or JSON) or transmission of actual files, as e.g. for file reading and writing via PUT and GET commands. Relative addresses are resolved differently depending on the invoking component: If it is a front-end component then the address is resolved against the *code-base URL*, i.e. the URL of the directory the front-end

$$\frac{N \parallel \mathcal{R} \longrightarrow \mathcal{R}' \wedge \mathbf{com}(y) \stackrel{\Delta}{=} E \in D \wedge match(l\,path, pat, \mathcal{R})}{[x = \mathbf{com}_{l\,path}\, e.P]_{url} \parallel [\langle D \rangle^{pat}]_{l \backslash \mathbf{exec} \backslash m \backslash} \parallel \mathcal{R} \longrightarrow_c \mathcal{R}'}$$

$$\begin{aligned} N \;=\; & ((\nu z)([\overline{z}\, e.z(x).P]_{url} \parallel \\ & (\nu t)[z(y).E\,\theta_1\theta_2\{\overline{z}/\mathbf{return}\}]_{l \backslash \mathbf{exec} \backslash m \backslash t \backslash})) \parallel [\langle D \rangle^{pat}]_{l \backslash \mathbf{exec} \backslash m \backslash} \\ & \qquad z \notin \{t, y\} \wedge \{z, t, y\} \cap fr(e, P, url, E, pat, path, m) = \emptyset \end{aligned}$$

$$\theta_1 = \{path - pat \;/ < \mathbf{ipath} >\}$$

$$\begin{aligned} \theta_2 = \; & \{x = \mathbf{com}^{\mathcal{I}}_{url(l\,d(pat), rpath)} e \,/\, x = \mathbf{com}^{\mathcal{I}}_{rpath} e \mid \mathbf{com} \in Com\} \\ & \{x = \mathbf{com}_{url(l\,d(pat), ref)} e \,/\, x = \mathbf{com}_{ref} e \mid \mathbf{com} \in Com\} \end{aligned}$$

Table 4: Rules for auxiliary transitions of matched commands.

component was downloaded from (just denoted by "base URL" in Figure 1 and in the following, whenever clear from the context) that is stored together with the front-end component when it is locally deployed. If, instead, it is a back-end component then the address is resolved against the *physical-base URL*, i.e. the URL of the directory at which the back-end component is executable.

The basic middleware commands are:

– **Manage a file/resource** at a specified URL: creating, modifying, reading or deleting; realized, e.g., by REST HTTP methods.

– **Remotely execute** an operation of a (back-end) component at the specified URL. It takes as parameter the operation name and the set of operation parameters and it returns the operation result. It is realized, e.g., by an extension of RESTful services (we will present in the following) that makes use of the POST HTTP method.

– **Locally execute** a front-end component by reading (downloading) its code from the specified URL and locally deploying it. It takes as parameter the set of parameters to be passed to the component execution/initialization and, if needed, local deployment infos and it returns the local reference (URL) to the deployed component. It is realized, e.g., as a GET HTTP method combined with mechanisms to locally deploy and execute the component.

Component deployment and **undeployment** are also basic mechanisms of the middleware. They are invoked directly, as commands, to deploy/undeploy back-end components or, internally, e.g. as part of the local execution command to locally deploy front-end components. They are realized, e.g., by performing POST/PUT and GET/DELETE of component code at special reserved URLs;

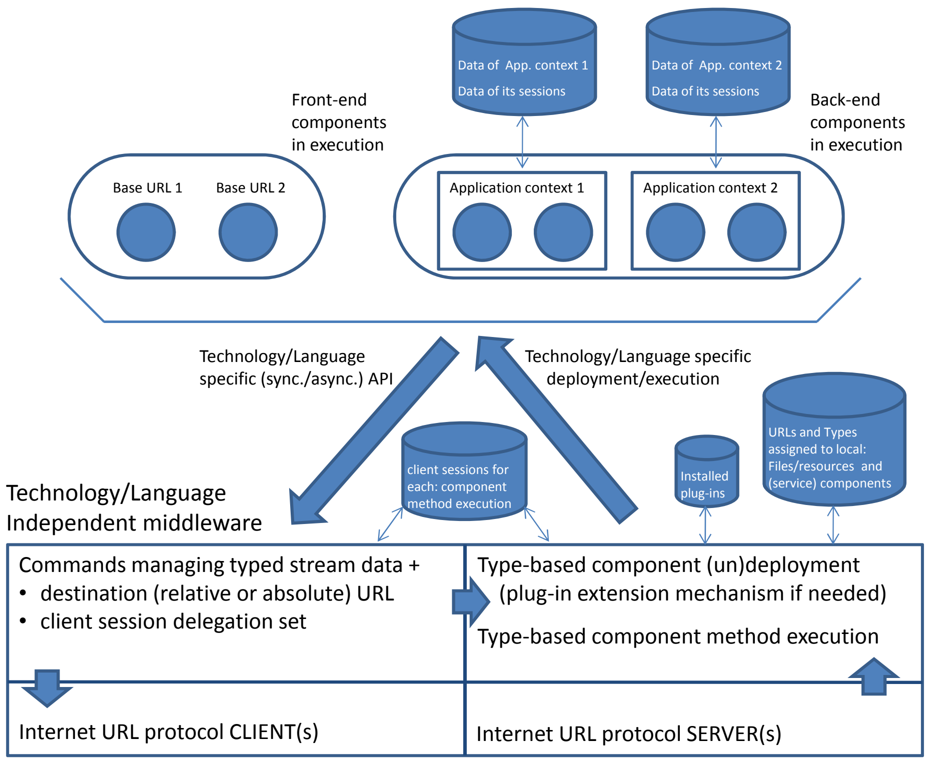
Data of App. context 1
Data of its sessions
Data of App. context 2
Data of its sessions
Front-end components in execution
Back-end components in execution
Base URL 1
Base URL 2
Application context 1
Application context 2
Technology/Language specific (sync./async.) API
Technology/Language specific deployment/execution
client sessions for each: component method execution
Installed plug-ins
URLs and Types assigned to local: Files/resources and (service) components
Technology/Language Independent middleware
Commands managing typed stream data +
• destination (relative or absolute) URL
• client session delegation set
Type-based component (un)deployment (plug-in extension mechanism if needed)
Type-based component method execution
Internet URL protocol CLIENT(s)
Internet URL protocol SERVER(s)

Figure 1: Basic Web-OS architecture

that is, in the case of RESTful, those (we denoted with prefix "$l\backslash$**exec**$\backslash$" in the REST process algebra) at which URL pattern-based matching mechanism takes place. In this way they are treated as special cases of file managing commands.

3.1 Application Contexts and Session Managing

As shown in the upper part of Figure 1, we assume back-end components and resources to be organized in *application contexts* each including a (web) application: in particular back-end components that are deployed/in execution and passive front-end components ready to be downloaded and deployed/executed. As standard, we will consider a *context URL* as the root of all application context resources, which are included in some subdirectory of the context URL. A related important notion that we will use is that of application and session data.

We assume that each application context uses sessions (session identifiers) to maintain data associated with client sessions via a technology dependent data structure (e.g. for Java based technologies, session attributes containing objects).

As quite usual, we consider each application context to independently manage sessions, which, hence, have application scope. From the client side, therefore, session information is collected as set of pairs composed of a *session id* and the *context URL* the id refers to. The middleware maintains such a set of client sessions when executing a (front-end or back-end) component operation.[4]

We also consider session delegation: all commands allow a session delegation set to be specified, i.e. the set of *context URLs* whose client sessions (if possessed by the component operation calling the command) are to be passed. When a command causes a component operation to be invoked, the latter will receive (if possessed by the caller) the client session id for its own context URL (standard HTTP session managing) and, possibly, delegated sessions (a set of pairs session id and associated context URL) that are added to the operation client sessions. The operation response will then, symmetrically, include a new status (created/removed session id) of the client session related to its own context URL (as standard) and of the delegated client sessions (which are delegated "back" at the response). Such information is then used to update the status of the set of client sessions possessed by the operation that invoked the command.

3.2 Middleware Behavior

We now describe in detail how middleware behaves when executing commands we previosly described. Concerning the left-hand side of Figure 1, command execution is based on using, as a client, the protocol specified in the URL address. Notice that, the technology/language dependent API, which invokes middleware commands, does not need to wait to receive the return value from the middleware before continuing with the execution of code: this asynchronous behaviour was, e.g., modeled by using the **spawn** primitive in the REST process algebra, see Example 2. Technically it can install an event listener (as done e.g. in AJAX) and interface with the middleware in such a way that the listener is executed upon completion of the command. Independently of the synchronous or asynchronous realization of the API for a particular technology, the middleware must be implemented so to be able to manage command requests in parallel in that: the client technology may use multithreading and, anyway, middleware command requests may come from operation executions of (different) components, which are run in parallel.

Concerning the right-hand side of Figure 1, the Web-OS middleware employs server(s) for protocols of URL addresses of files/resources and back-end components it exposes to the Internet. According to the client invocations we described

[4] Client sessions are therefore not shared between components or between different execution of operations in the same component, differently from what happens, e.g., in a browser.

above, such servers receive requests to read, write, delete files/resources, to deploy/undeploy back-end components and to remotely execute an operation of a back-end component. Regarding files/resources, we just assume that such servers perform the required operation by exploiting standard OS functionalities (e.g. of the underlying local filesystem manager), and, whenever a new file/resource is created the server records its URL and its type. The client session passed with the request can be exploited to check the identity of the client and its right to manage the file/resources. Back-end component (un)deployment is based on the type of the component to be (un)deployed (indicating the technology/language of the component): in the case of deployment, if the back-end component technology is still not supported by the Web-OS running on the local machine, a remote repository of plug-ins is queried to find a plug-in to install in the system so to support the back-end component technology. For supported technologies it uses the corresponding plug-in to perform the (un)deployment and it associates the deployment URL (in the configuration of the server of the protocol specified in such URL) and the type with the component. Concerning remote execution of an operation of a (back-end) component, the server exploits the middleware functionalities to execute the required operation via the plug-in corresponding to the component type. Before actual operation execution, the middleware associates a set of possessed client sessions to such an execution (which is initialized with delegated client sessions received with the request) and maintains it until the end of the execution.

Functionalities in the right-hand side of Figure 1 are invoked directly by the left-hand side in case middleware commands are executed that refer to local URLs (there is no need to actually perform a protocol client invocation and then receive the request from the corresponding local server) or in the case of the *local execution* command. In the latter case front-end component local execution is performed by doing, first, a deployment of the front-end component based on the type of the component to be deployed: as for back-end components, if the front-end component technology is still not supported by the Web-OS running on the local machine, a remote repository of plug-ins is queried to find a plug-in to install. Once the front-end component is deployed by using the plug-in corresponding to its type, the component is executed/initialized by exploiting the same plug-in.

Finally, concerning the interface of the middleware with the technology dependent part of the Web-OS, which is realized by an extensible set of back-end or front-end technology plug-ins, we have: (i) The (front-end or back-end) component code invokes the middleware commands via a technology/language dependent API which also performs data marshalling/unmarshalling; (ii) The middleware executes (called operations of) back-end or front-end components by performing technology/language dependent required mechanisms and also performing data marshalling/unmarshalling. The type of file used for data marshalling/unmarshlling (e.g. XML or JSON) depends on the plug-in associated with the component type and on its configuration.

4 Web-OS Middleware as an Extension of RESTful Services

We propose a realization of the Web-OS Middleware that is based on an extension of RESTful services. As we already explained in the introduction, such an extension aims at adopting a uniform mechanism for managing both file/resources and interface-based stateful services, i.e. services invoked by specifying an operation name and a parameter, similarly as for SOAP services, and which make use of session and application data.

In RESTful services only resource related commands, i.e. PUT, GET, POST, DELETE, can be used, with the pre-defined intended semantics described in Section 2, and their behaviour must be stateless (i.e. their behaviour is determined merely by the input data they receive) [Fielding 2000]. Deviating from this schema, for obtaining also stateful operations with a user-defined meaning, is commonly considered to be a needed practice (widely used, e.g., by Twitter and Facebook). In the following, we will show that it is possible to encompass in a clean way (without breaking the REST philosophy) interface-based stateful services. Moreover, we will present a realization of the Web-OS Middleware based on such an extension.

4.1 Encompassing Interface-based Services

In order to show that a conceptual extension of RESTful Services to encompass interface-based services can be done in a “clean” way we make a parallel with object oriented programming.

Resource “$\backslash persons\backslash$” of Example 1 could be seen as representing a class of person objects, and “*id*” in “$\backslash persons\backslash id$” as a reference to one of its objects. POST would then correspond to the constructor (which is a static method, i.e. called just referring to the class, and returns the reference “*id*” to a freshly allocated object), while PUT, GET and DELETE, respectively to putter, getter and destructor methods, which are non-static, i.e. called on a specific object of the class (determined by the reference “*id*”) to act on it. Notice that, by using a service associated to the “$\backslash persons\backslash *$” pattern, its PUT, GET and DELETE operations would receive the suffix “*id*” exactly as non-static methods of a class receive the reference of object in the variable “*this*”.

However, in object oriented programming, besides methods above dedicated to managing resources (object memory) with the given intended meaning, we can have also user-defined standard methods with no predefined meaning, that are naturally stateful. Similarly, such methods can be smoothly added also to the schema of RESTful services of [Fielding 2000] by using POST requests on single resources (which do not have a definition in the REST schema), e.g. a POST request at "$\backslash persons \backslash id$". Such a request would include, besides the URL of the resource on which the operation is expected to work (similarly as for "this" yielding the object on which the method was called, the POST operation of a service associated, e.g., to the "$\backslash persons \backslash *$" pattern would receive the suffix "*id*") the operation (method) name and the parameter data; while the response, the data returned by the operation. Therefore, for such operations, differently from those originally in the RESTful schema of [Fielding 2000], it makes sense to manage application and session (client) data in a stateful manner. For this reason we will also consider the client session and delegated sessions to be passed with operation requests.

4.2 Middleware Commands

Middleware commands, realizing those described in Section 3, are: **put**, **get** and **delete**, which work in the same way as for REST commands with the same name; and **rexec** and **lexec** that are described below.

The remote execution command **rexec** is realized as an extension of the functionalities of the **post** command of REST. As already explained, the idea is to make use of a smooth extension to the schema of RESTful services [Fielding 2000] by using POST requests on single resources, e.g., considering Example 1, a POST request at "$\backslash persons \backslash id$". Technically, such a request indicates the URL of the resource on which the operation is expected to work (the POST command definition of a service associated, e.g., to the "$\backslash persons \backslash *$" pattern receives, as usual, the suffix "*id*" in the $<$ **ipath** $>$ variable) and includes the operation name/parameter data in some interoperable format like XML or JSON, which is also used for the data returned by the operation in the response. Therefore, on single resources **rexec** allows for remote execution of an operation with a user defined name and meaning as for interface-based (e.g. SOAP) services: service declarations D are extended to include the code of such operations. On collection resources **rexec** preserves the same behaviour of subresource creation as in the REST **post** command.

The local execution command **lexec** is realized as follows. **lexec** at a *url* retrieves a passive front-end component, i.e. its code declaration $\langle D \rangle_{type}$ which also specifies the type/language of the code, via a **get** at *url*. Then it locally deploys it by creating, via **rexec**, a subresource at the location "*l*" corresponding to a plug-in which is able to execute such a code type: the precise deployment URL being a subresource of "$l\backslash$**exec**$\backslash$". In particular, it associates the deployed component code $\langle D \rangle_{type}$ with a single resource (not including "$*$") local execution pattern *pat* and with its code-base URL *baseurl* (the URL of the directory the component was downloaded from, i.e. such that $url = baseurl\, n$ for some n), denoted by $\langle D\rangle_{type}^{baseurl \to pat}$. Finally it runs it by performing a **put** at pat.

5 A Process Algebra for WebOS Middleware

5.1 Syntax

The definition of (well-defined) *networks* and *systems* is the same as for the process algebra for REST presented in Section 2. Here, though, we consider an extended syntax for terms $[R]_{url}$, that is, supposed $R ::= v \mid P$, for URLs url, values v and programs in execution P.

Concerning URLs, in a system they have the same syntax as for the REST process algebra. In the following section about semantics we will consider additional special URLs (that represent, for example, session managing with special resources) that can be present in a network during execution only. Since process/threads P are not present in a system (as for the REST process algebra they can only occur in a network during execution) they will also be detailed in the following section about semantics. Here we will, thus, focus just on describing values.

As for the REST process algebra, part of a system specification is also predicate $cond(url)$ that specifies, for directory URLs, whether the the default subresource creation behavior of the **rexec** command (**post** in the REST process algebra) creates single ($false$ case) or collection ($true$ case) resources. Predicate $cond(url)$ is assumed to be such that $cond(l\backslash\mathbf{exec}\backslash) = cond(l\backslash) = true$ as in the case of REST. Moreover, here we additionally consider, as part of a system specification, partial function $loc_{l,type}$. A pair $(l, type)$ belonging to the domain of function loc means that an **lexec** performed by code running at context location l is capable of executing the client-side technology of type $type$ (e.g. a plug-in for executing that kind of technology is available in the machine where context location l is deployed): in this case $loc_{l,type}$ yields the context l' at which the technology $type$ is executable (in the same machine where l is deployed).

5.1.1 Values

Concerning values v, we consider in addition to those considered for the REST process algebra: locations l, so to make it possible for expressions to compute a location (e.g. the location where to execute front-end components downloaded by **lexec**); pairs $x\!<\!v\!>$ used to represent combination of operation name x and parameter value v passed to a **rexec**; deployed front-end components $\langle D\rangle_{type}^{url^{\perp} \to pat}$; passive typed (where the type denotes their technology) front-end components $\langle D\rangle_{type}$ to be downloaded by **lexec**. The overall syntax of values v thus becomes

$$v ::= \varepsilon \mid pval \mid x \mid l \mid ref \mid x\!<\!v\!> \mid < v, \ldots, v > \mid \langle D\rangle_{type}^{url^{\perp} \to pat} \mid \langle D\rangle_{type}$$

$$pval ::= \mathbf{ok} \mid \mathbf{err} \mid num \mid \ldots$$

where $url^{\perp}$ stands for either url or $\perp$. The two cases arise depending whether the deployed component is a front-end component (in this case the url represents the code-base url) or not (in this case we will omit writing "$\perp \rightarrow$" in the following, thus getting the representation for deployed services used in the REST process algebra apart from the added type information). Notice that deployed front-end components should only be present during system execution, in that generated by the semantics of an **lexec** command.

Declarations D contain the code of component operations op: those that are user-defined and those associated to (resource related) commands as in the REST process algebra. Operations op are, thus, defined by:

$$op ::= \mathbf{com} \mid x \qquad\qquad \mathbf{com} ::= \mathbf{put} \mid \mathbf{get} \mid \mathbf{delete} \mid \mathbf{rexec}$$

Notice that **com** cannot be **lexec**, which is executed locally by running code and not by performing an invocation to a component. Formally, D is a partial function from operations op to pairs composed by a formal parameter variable x and a term E representing the code of the operation. Definitions in D are represented as $op(x) \overset{\Delta}{=} E$ ($op \overset{\Delta}{=} E$ if $x \notin fr(E)$ as it must be for $op \in \{\mathbf{get}, \mathbf{delete}\}$).

5.1.2 Code of Operations

We now present the syntax of terms E, i.e. code defining operations. We preliminary need some definitions.

Concerning references ref, used for expressing addresses in a middleware command calls, we introduce, with respect to the REST process algebra, the possibility of using symbolic addresses. In particular, we consider reference URLs ref starting with <**session**>, <**application**> and <**phbase**>. A ref starting with <**session**> is used in back-end components to manage session attributes of the current application context, e.g. $<\!\mathbf{session}\!>\backslash username$ can be used to manage the username of the invoker of the operation (identified by its session identifier that is automatically passed when the operation is invoked, as it happens, e.g., with HTTP sessions). Similarly, a ref starting with <**application**> is used in back-end components to manage application context attributes, e.g. $<\mathbf{application}>\backslash accounts\backslash$ can be used to manage the accounts of the current context users. Finally, a ref starting with <**phbase**> is used in front-end components to explicitly access the physical-base (given that for such components relative addressing is otherwise automatically resolved with respect to their code-base). The syntax of references ref is, thus, defined as for the REST process algebra with the additional production:

$$ref ::= \cdots \mid sym\backslash rpath$$

$$sym ::= <\!\mathbf{session}\!> \mid <\!\mathbf{application}\!> \mid <\!\mathbf{phbase}\!>$$

Moreover, we need to introduce session delegation sets "rs", which, attached (as a superscript) to middleware command calls, specify a set of contexts for which we want the client session to be delegated (and delegated back with any update when the command returns):

$$rs ::= \{rlist\} \mid \{\} \mid \mathcal{I} \qquad\qquad rlist ::= l, rlist \mid l \mid \epsilon$$

ϵ included in the list is a relative reference to the context of the base URL (codebase for front-end components, physical-base for back-end components). $rs = \{\}$ just denotes the absence of session delegation (in this case the "rs" superscript is just omitted when writing the command). Finally, the special case $\mathcal{I}$ of "rs", as for command flag "in" of the REST process algebra, denotes an internal command, i.e. the explicit invocation of the default command behaviour, thus modeling a direct local resource access.

The syntax of terms E representing operation code is as follows.

$$\begin{aligned} E ::= {} & x = \mathbf{put}^{rs}_{ref}\, e.E \mid x = \mathbf{get}^{rs}_{ref}.E \mid x = \mathbf{delete}^{rs}_{ref}.E \mid x = \mathbf{rexec}^{rs}_{ref}\, e.E \mid \\ & x = \mathbf{lexec}^{rs}_{ref}\, e.E \mid x = e.E \mid \overline{x}^{rs}\, e.E \mid x(y).E \mid (\nu x)\, E \mid \mathbf{spawn}\ E\,.E \mid \\ & \mathbf{if}\ be\ \ \mathbf{then}\, E\ \mathbf{else}\, E \mid \nu{<}\mathbf{session}{>}.E \mid \neg{<}\mathbf{session}{>}.E \mid recX.E \mid X \mid \\ & \underline{0} \mid \mathbf{return}\ e \end{aligned}$$

where: for commands $x = \mathbf{com}^{rs}_{ref} e.E$ (with e omitted in the case of **com** being **get** or **delete**), we have that $rs = \mathcal{I}$ implies $\exists\, rpath_s : ref = rpath_s$; while for command $x = \mathbf{lexec}^{rs}_{ref} e.E$ and for outputs $\overline{x}^{rs}\, e.E$ we have that $rs \neq \mathcal{I}$ (rs in outputs means that client sessions of specified contexts are simply transmitted).

Besides command calls and the operators we already considered for the REST process algebra, we here introduce $\nu{<}\,\mathbf{session}{>}.E$ and $\neg{<}\,\mathbf{session}{>}.E$ that perform session creation and destruction, respectively. ν <**session**> generates, in the current context (the context l that executes it), a fresh session id that is to be assigned to the invoker of the operation executing ν <**session**>: the operation (automatically) transmits such a session id when it returns (the invoker will then hold it as a client session for l and automatically send it back whenever invoking again some operation of l). Once created, the session is accessible by operation code via the symbolic URL < **session** >: as we already explained session attributes can be created/written or read by putting or getting values at URLs whose initial part is in the form <**session**>$\backslash$*attribute*. Simmetrically, $\neg$ <**session**> destroys the client session for the current context l possessed by the invoker of the operation executing it: the invoker and, if applicable, who delegated its client session for l to it, and so on ... in a chain, they all get their client session destroyed (an analogous chain effect happens also in the previous case of session creation). $\neg$ <**session**> requires all existing attributes of <**session**> to be preliminarily deleted in order to work.

5.2 Online Editor Application Example

We will now present an example of a system specified with the process algebra we presented above. We consider a server which has a user authentication mechanism (based on registration, login and logout) and a small editor application.

The latter has: a *front-end component* with capabilities for (i) creating a document at a user specified relative address *frp* (file relative path) on the local machine where it is executed, i.e. at its physical-base, and (ii) saving it on the server it was downloaded from, i.e. at its code-base; and a *back-end component* that receives the document to be saved on the server and saves it on a user-dependent directory, i.e. the directory $l\backslash editor\backslash userfiles\backslash user\backslash frp$, where *user* is the name of the logged user that is determined from session information.

The system specification is N defined by (unused type subscripts are omitted):

$[\langle D_{init}\rangle^{\backslash init}]_{l_{init}\backslash \mathbf{exec}\backslash n\backslash} \parallel [\langle D\rangle^{\backslash users}]_{l\backslash \mathbf{exec}\backslash n\backslash} \parallel [\,]_{l\backslash pwdDB\backslash} \parallel$
$[\langle D'\rangle^{\backslash editor\backslash userfiles\backslash *}]_{l\backslash \mathbf{exec}\backslash n'\backslash} \parallel [\,]_{l\backslash editor\backslash userfiles\backslash} \parallel [\langle D''\rangle_{edtype}]_{l\backslash editor\backslash editorapp}$
with "[ ]" denoting a contained ε (empty) resource.

As for Example 3, the system is meant to be executed by creating, with a *put* command, an empty resource at $l_{init}\backslash init$: the code of **put** declared inside D_{init} acts as the "main" method. Thus, we take D_{init} to just include the following definition of "**put**":

$\mathbf{put} \stackrel{\Delta}{=}$
$\mathbf{rexec}_{l\backslash users} register <username, pwd>. \mathbf{rexec}_{l\backslash users} login <username, pwd>.$
$x = \mathbf{lexec}_{l\backslash editor\backslash editorapp}\ frp\ .\ \mathbf{rexec}^{\{l\}}_{l'\backslash x\backslash editorapp}\ save <frp>\ .$
$\mathbf{rexec}_{l\backslash users} logout <>\ .\ \mathbf{return}$

The code above could represent the actions of a user or a workflow (executed by context l_{init}) on the server (context l) exposing the editor application (under the directory $l\backslash editor$): we take "*username*", "*pwd*" and "*frp*" to be any example string constants, with "*frp*" representing a file relative path and being an *rpath* that does not end with "\". The technology/language of the editor front-end component *editorapp* is *edtype* and we assume partial function *loc* to be such that $loc_{l_{init},edtype} = l'$, i.e. in the machine running context l_{init} we assume there is a plug-in at context l' running components whose technology is *edtype*. Concerning predicate $cond(url)$, which is also part of the system specification, we do not need to make any assumpion about it in that in this example we never use command **rexec** to create subresources.

We now describe the user workflow above, then we will formally define the remaining declarations D,D' and D''. The user first registers in the server (context l) by invoking operation *register* of the back-end component of l associated to the $\backslash users$ pattern (the *register* operation, as we will see, creates a new username item under $l\backslash pwdDB\backslash$ that represents the l password database). Then, the user logs in, by invoking a different operation of the same back-end component (which creates a session id for the user and sets its username as a "user" property of its session), and locally executes the front-end component *editorapp* of the editor application with **lexec**, i.e., downloading it from "$l\backslash editor\backslash editorapp$", deploying it in the local context $l' = loc_{l_{init}, edtype}$ able of executing the "*edtype*" technology, and executing it with parameter "*frp*" (which is received by the **put** operation of the front-end component that, as we will see, creates a new document and saves it locally at the "*frp*" relative path, i.e. considering it to be relative to its physical-base). The call to **lexec** returns the fresh name of the collection resource generated in context l' to locally deploy the front-end component *editorapp*: such a name is stored in variable x and is, then, used ($l' \backslash x \backslash$ being the physical base of front-end the component) to call its operation *save*. This could represent, e.g., the effect of the actions of the user on the GUI of the front-end component. When calling *save*, the client session associated to l is delegated (superscript $\{l\}$) so that the code of operation *save*, which runs on context l', accesses the l server of the editor application on user's behalf by passing its session id (in the previous case of **lexec** the l client session delegation is done automatically when it internally calls **put**, as we will detail in the next section about semantics). The *save* operation, as we will see, accesses (by using a simple relative path beginning with $userfiles\backslash$, which is referred, by default, to its code-base $l\backslash editor$) the editor application back-end component with pattern $\backslash editor \backslash userfiles \backslash *$ in order to save a copy on the server of the file whose local relative path is *frp* (under a server directory whose name is the username of the user, determined from the "user" property of its session, see details below). Finally, the user logs out: the "user" property of its session is deleted and its session id is destroyed.

Set D, defining the back-end component with pattern $\backslash users$, contains:

$$register(x) \overset{\Delta}{=} usr = x\hat{}1 \,.\, pwd = x\hat{}2 \,.\, \mathbf{put}_{pwdDB\backslash usr} pwd \,.\, \mathbf{return}$$

$$login(x) \overset{\Delta}{=} usr = x\hat{}1 \,.\, pwd = x\hat{}2 \,.\, y = \mathbf{get}_{pwdDB\backslash usr} \,.\, \mathbf{if}\; y == pwd \;\mathbf{then}$$
$$(\nu \!<\!\mathbf{session}\!>\, .\, \mathbf{put}_{<\mathbf{session}>\backslash user} usr \,.\, \mathbf{return\,ok})\; \mathbf{else\;return\,err}$$

$$logout(x) \overset{\Delta}{=} \mathbf{delete}_{<\mathbf{session}>\backslash user} \,.\, \neg\!<\!\mathbf{session}\!>\, .\, \mathbf{return}$$

where $x\hat{}1$ and $x\hat{}2$ stand for the first and second element of the list in x.

Set D', defining *editor back-end component* (with pattern $\backslash editor \backslash userfiles \backslash *$), contains:

$$\mathbf{put}(x) \overset{\Delta}{=} y = \mathbf{get}_{<\mathbf{session}>\backslash user} \,.\, \mathbf{put}^{\mathcal{I}}_{y\backslash <\mathbf{ipath}>} x \,.\, \mathbf{return}$$

Set D'', defining *editor front-end component*, contains:

$$\mathbf{put}(frp) \overset{\Delta}{=} \mathbf{put}_{<\mathbf{phbase}>\backslash frp} newdoc \,.\, \mathbf{return}$$

$$save(frp) \overset{\Delta}{=} x = \mathbf{get}_{<\mathbf{phbase}>\backslash frp} \,.\, \mathbf{put}_{userfiles\backslash frp} x \,.\, \mathbf{return}$$

Here *newdoc* is any predefined constant representing the value of created document. Notice that, as we already described above, the last *put* specifies an address which is relative to the front-end component code-base.

5.3 Semantics

In order to present the semantics we need to preliminary define the syntax of terms P representing programs/operations in execution.

We begin by extending the syntax of URLs so to represent, session and application attribute managing via special resources. First of all, we introduce metavariable $\mathcal{S}$ that ranges over client session identifiers (a name x or no session **ns**, in the case no session for the current context is detained by the client):

$$\mathcal{S} ::= \mathbf{ns} \mid x$$

The extended syntax of *url* that we consider is:

$$url ::= l\backslash[\mathbf{exec}\backslash]rpath \mid l\backslash special\backslash rpath$$
$$special ::= \mathbf{session} \mid \mathbf{session}\backslash\mathcal{S} \mid \mathbf{application}$$

Predicate $cond(url)$ is assumed to be consistently extended with respect to the one provided with the system specification in such a way that, for any context l, $cond(l\backslash\mathbf{session}\backslash) = true$.

Moreover, we need to introduce session-location delegation sets "*sls*", i.e. sets of pairs $l:\mathcal{S}$, which, attached (as a superscript) to middleware command calls, are transmitted to the command code (and transmitted back with any update to the involved sessions when the command is terminated). As we will see, they are determined according to session delegation sets rs specified in the static code (terms E). Their syntax is:

$$sls ::= \{sllist\} \mid \{\} \mid \mathcal{I} \qquad\qquad sllist ::= l:\mathcal{S}, sllist \mid l:\mathcal{S}$$

As for "rs" superscripts, $sls = \{\}$ just denotes the absence of session delegation (in this case the "*sls*" superscript is just omitted when writing the command) and $sls = \mathcal{I}$ denotes an internal command.

The syntax of terms P representing operations under execution is as follows.

$$\begin{aligned} P ::= {} & x = \mathbf{put}^{sls}_{surl:\mathcal{S}}\, e.P \mid x = \mathbf{get}^{sls}_{surl:\mathcal{S}}.P \mid x = \mathbf{delete}^{sls}_{surl:\mathcal{S}}.P \mid \\ & x = \mathbf{rexec}^{sls}_{surl:\mathcal{S}}\, e.P \mid x = \mathbf{lexec}^{sls}_{surl:\mathcal{S}}\, e.P \mid x = \ e.P \mid \overline{x}^{\,sls}\, e.P \mid \\ & x(y).P \mid (\nu x)\, P \mid \mathbf{spawn}\ P\,.P \mid \mathbf{if}\, be\ \mathbf{then}\, P\, \mathbf{else}\, P \mid \\ & \nu\ l\backslash\mathbf{session}\backslash\mathcal{S}.P \mid \neg\ l\backslash\mathbf{session}\backslash\mathcal{S}.P \mid recX.P \mid X \mid \underline{0} \end{aligned}$$

where a symbolic URL *surl* is a *url* where a name $x \in \mathcal{N}$ may occur in the place of its location l; that is, *surl* is such that, for some x, $surl\{l/x\}$ is a *url* (this obviusly holds in the case *surl* is a *url*). Symbolic URLs are needed to express run-time behaviours where the location l of a URL depends on the content of a variable x, as in the semantics of the **lexec** command that, as we will see, determines the local deployment location based on predifined function *loc*. Moreover, with respect to static code, every *surl* is followed by a ":$\mathcal{S}$" session information, which: if *surl* is a *url* denotes the client session id possessed for the context l of the *url* (which is automatically transmitted when the command is called); otherwise we always have $\mathcal{S} = \mathbf{ns}$ (no client session id is possessed). Finally session creation and destruction are expressed in terms of actions on $l\backslash\mathbf{session}\backslash\mathcal{S}$ that abstractly identifies resources of the current client session $\mathcal{S}$.

We assume function $fr()$, yielding the set of free names of $\mathcal{N}$ of a (list of) syntactic element(s), replacement of a name $x \in \mathcal{N}$ of P with value v, i.e. $P\{v/x\}$, and replacement of a recursion variable X of P with a term P', i.e. $P\{P'/X\}$, to be defined as in the process algebra for REST.

We are now in a position to present the actual semantics. As for the REST process algebra, we make use, for notational convenience, of paths $path$ representing the path information that can occur in an url (with the extended syntax considered above) after a context l, i.e. a $path$ is a string such that $l\,path$ is a url, and basic urls $burl$ as a directory url where the final $\backslash$ is omitted, i.e. a $burl$ is a string such that $burl\backslash$ is a url. We will use $Path$ to denote the set of all $path$ strings above. Moreover, Com is taken to be the set of commands **com** (commands that can be defined in a declaration D), while $ACom$ is taken to be the set of commands **acom**, with **acom** ranging over all middleware commands:

$$\mathbf{acom} ::= \mathbf{com} \mid \mathbf{lexec}$$

The semantics, in reduction style, relies on a structural congruence relation defined by the same laws in Table 1 considered for the REST process algebra. Moreover, assumed $\mathcal{R}$ (resource sets) to denote terms N that do not include $(\nu n)N$ subterms and $[R]_{url} \in \mathcal{R}$ to mean that $\mathcal{R}$ includes $[R]_{url}$ among its parallel terms, reduction transitions $\longrightarrow$ are defined, with the help of auxiliary transitions $\longrightarrow_c$ for commands **com**, by:

– the same rules in Tables 2 and 3 considered for the REST process algebra, apart from the last rule of Table 2 (communication rule) which must be replaced by the rule in Table 5 (which accounts for transmission of the client session and for session delegation) and from the fact that in Table 3 **post** is to be replaced by **rexec**, in is to be replaced by sls, and "$:\mathcal{S}$" has to be added at the end of the url of every command.

– the rules in Table 6 (which replace the rule for command match presented in Table 4 for the REST process algebra), Table 7 and 8.

In the following we present each table and we define the auxiliary functions it uses.

Table 3 (with the above modifications) and Table 6 include the semantic rules defining auxiliary transitions $\longrightarrow_c$ for commands **com**. Concerning **put**, **get** and **delete** the behavior is the same as in the REST process algebra. The same holds true for **rexec** on collection resources, that works as **post**. In particular, when they are not redefined by matching, they all ignoring session information, including delegation, passed with the request. Concerning a **rexec** *op*<*v*> command on a single resource URL, in case of matching with a back-end component (which does not redefine command **rexec** itself), it invokes operation "*op*" defined in component declaration D.

Table 3 deals with the behaviour of unmatched and internal commands. All auxiliary functions used in the table are defined as for the REST process algebra (with $pats_{url}$ and $coms$ considering $R = \langle D\rangle_{type}^{url^{\perp}\rightarrow pat}$ for any $url^{\perp}$ and $type$, instead of $R = \langle D\rangle^{pat}$), except sets of locations in which it is allowed to directly generate new subitems via **rexec** (In_g) and to create a subitem via **put** (In_p), which need to be both extended. They are now defined by $In_g = \{l\backslash, l\backslash\mathbf{session}\backslash, l\backslash\mathbf{exec}\backslash \mid l \in \mathcal{L}\}$ and $In_p = \{\varepsilon, l\backslash, l\backslash\mathbf{application}\backslash \mid l \in \mathcal{L}\}$: i.e. session ids are generated as fresh subresources of $l\backslash\mathbf{session}\backslash$ that are collections (where session attributes of that session id will be put) and it is possible to create application attributes under $l\backslash\mathbf{application}\backslash$.

Table 6 defines the behavior of a command **com** when the URL it addresses is matched by a pattern pat of some declaration $\langle D\rangle_{type}^{url^{\perp}\rightarrow pat}$ (residing at the **exec** directory of the context l of the addressed URL) that redefines its behavior, i.e., in the case of **com** being **rexec**: either D includes a definition of **rexec** (in this case the first rule of Table 6, which works exactly like the rule for matching of REST commands, is applied); or D does not include it, but it includes the definition of operation op with $op{<}e{>}$ being the parameter of **rexec**, for some expression e (in this case the second rule of Table 6 is applied). The two rules in Table 6 check pattern matching with the help of the auxiliary predicate $match(url, pat, \mathcal{R})$ that is defined as for the REST process algebra and realize the operation call by means of pi-calculus channel based communication. The only significant novelty here is the management of symbolic references, which are replaced with corresponding (special) URLs, and session delegation in pi-calculus channel based communication both when the operation is called (including both explicitly delegated client sessions and the the client session for location l of the URL addressed by the command, which is automatically transmitted) and when the operation returns (transmitting back the status of explicitely delegated and l sessions). In this table we need to consider a couple of new auxiliary functions. $\hookrightarrow$ yields the argument on its left if it is not $\perp$, otherwise it yields the argument on its right. It is used for selecting between code-base (for front-end components) and physical-base (for back-end components). Function $url(url, ref)$ performs relative ref addressing resolution and is defined as for the REST process algebra. Function $l(l\,optpath, optl)$, with $optpath \in Path \cup \{\varepsilon\}$ and $optl$ being either ϵ or a location l', operates similarly considering locations only: it returns l if $optl$ is ϵ (relative addressing resolution), l' otherwise.

Table 7 defines the behaviour of the **lexec** command. An important role is played by the predefined loc partial function, which, based on the $type$ of the front-end component at the URL (denoted by $l\,path\backslash n$ in the table) referred by **lexec** and the location (l' in the table) executing the **lexec** command, determines the local deployment location for the component (such a location is a server context residing in the same machine as l' which is capable of executing the technology of the front-end component). In particular, if partial function loc is undefined it means that such a technology is unsupported (the machine where l' resides is uncapable of executing technology $type$). This is expressed in Table 7 by using some auxiliary functions. $type(v)$ determines the type of value v in the case it is a passive typed front end component. More precisely, we have that $type(v)$ yields: $type$, if $v = \langle D\rangle_{type}$; ε, otherwise. The latter case is used to detect that v is not a passive typed front end component (we assume that, obviously, ε is not a $type$): if such a v is at the URL referred to by an **lexec** it fails. $supported_{l,type}$ is assumed to be a predicate that is $true$ whenever $(l, type)$ belongs to the domain of predefined function $loc_{l,type}$. Notice that, in the case $type$ is ε, this implies that $supported_{l,type}$ is $false$. The behaviour of **lexec** defined in the rule of Table 7 is the following. First the front-end component is downloaded from the reference URL $l\,path\backslash n$, then, in the case its technology $type$ is supported for location l' running **lexec** (and it is indeed a front-end component), it is deployed at the **exec** directory of the location loc given by $loc_{l',type}$. In order to do this an execution pattern $loc\backslash n'\backslash n$ (where n' is a freshly generated collection resource and n is the component name, i.e. the last name in the **lexec** reference URL) is assigned to the deployed component, constituting its physical-base (the directory $loc\backslash n'$). The code-base of the component (the directory $l\,path\backslash$ of the **lexec** reference URL) is also recorded together with the deployment descriptor, as explained before. Finally the front-end component is executed by performing a **put** command (its "constructor") at $loc\backslash n'\backslash n$ and passing it the argument e of **lexec**. Session delegation prescribed by the sls superscript of **lexec** is done when calling the **put** command, which also delegates the l client session, so that the front-end component can access back-end components at its code-base location l on behalf of the thread running **lexec**. **lexec** returns the freshly generated directory name n' which can be then used (under location loc) to execute operations on the locally deployed front-end component.

Table 8 defines session id creation and destruction for the current client (the invoker of the operation code). Recall that $l\backslash\mathbf{session}\backslash\mathcal{S}$ abstractly identifies resources of the current client session, $\mathcal{S}$ being the session id or **ns** (meaning that no session is detained). Thus, in the case of creation, nothing has to be done if $\mathcal{S} \neq \mathbf{ns}$ and, similarly, for destruction if $\mathcal{S} = \mathbf{ns}$ (first rules for session creation and destruction in Table 8). In the case of creation with $\mathcal{S} = \mathbf{ns}$, a new fresh collection resource is created under $l\backslash\mathbf{session}\backslash$ whose name becomes the current session id (this is enacted by syntactical replacement in the code continuation). In the case of destruction with $\mathcal{S} \neq \mathbf{ns}$, the collection resource $l\backslash\mathbf{session}\backslash\mathcal{S}$ is deleted (which requires all session attributes for session id $\mathcal{S}$ to have been previously deleted) and the current client session is set to the **ns** value (this is, again, enacted by syntactical replacement in the code continuation).

Notice that, similarly as for the semantics of the REST process algebra (Ta-

$$[\overline{x}^{\,sls} e.P]_{url} \parallel [x(y).Q]_{url'} \parallel \mathcal{R} \longrightarrow [P]_{url} \parallel [Q\{\mathcal{E}(e)/y\}\theta_1\theta_2]_{url'} \parallel \mathcal{R}$$
$$\theta_1 = \{l\backslash\mathbf{session}\backslash\mathcal{S}' \;/\; l\backslash\mathbf{session}\backslash\mathcal{S} \mid\; l : \mathcal{S}' \in sls\}$$
$$\theta_2 = \{l\ optpath : \mathcal{S}' \,/\, l\ optpath : \mathcal{S} \mid optpath \in Path \cup \{\varepsilon\} \wedge\ l : \mathcal{S}' \in sls\}$$

Table 5: New communication rule.

ble 4): replacement $\{\overline{z}^{sls\cup\{l:\mathcal{S}\}}/\mathbf{return}\}$ in Table 6 means that all occurrences of "$\mathbf{return}\, e$" terms in E (inside any binder) are replaced by "$\overline{z}^{sls\cup\{l:\mathcal{S}\}}\, e.\, \underline{0}$"; and replacements θ_1 and θ_2 in Tables 5, 6 and 8 syntactically replace all occurrences of elements to the right of "/" (inside any binder) with the corresponding elements to the left.

We now define semantics of systems following the same approach adopted for the REST process algebra.

Definition 2. Let N be a well-defined system. We use $[\![N]\!]_{url}$ to denote the semantics of N when executed by creating, with a **put** command, an empty (ε) resource at url. $[\![N]\!]_{url}$ is defined as as in Definition 1 where we take $P = \mathbf{put}_{url:\mathbf{ns}}$.

For example the semantics of the online editor application example presented in Section 5.2 is obtained by considering $[\![N]\!]_{l_{init}\backslash init}$.

5.4 Java Based Implementations

Simple Java based implementations of our integrated version of interface-based and RESTful based web services (on which WebOS middleware primitives are based) were experimented in the context of several Master's (and one Bachelor's) Theses that we supervised, see e.g. [Gregori 2007, Guidi 2008, Taioli 2008, Giovannini 2010, Torelli 2014]. In particular, we created a small Java library capable of performing the involved middleware commands by storing client sessions and resolving relative url addressing [Guidi 2008]. We used such a library in applets (front-end components downloaded with the local execution mechanism) and servlets (back-end components). We studied the effectiveness of our middleware primitives by considering several example applications. In particular, we considered comet applications (application that send events in real time to the front-end interface) and we tested several solutions based on services keeping responses permanently open and performing long polling requests (where a new request is performed immediately after a comet message is retrieved) [Gregori 2007, Taioli 2008, Giovannini 2010, Torelli 2014]. We also noticed that the usage of a session-based push service made it possible to avoid using pipe-based internal communication between servlets: in this way application internal

$$\frac{N \parallel \mathcal{R} \longrightarrow \mathcal{R}' \wedge \mathbf{com}(y) \stackrel{\Delta}{=} E \in D \wedge match(l\,path, pat, \mathcal{R})}{[x = \mathbf{com}^{sls}_{l\,path:S}\, e.P]_{url'} \parallel [\langle D\rangle^{url^{\perp}\rightarrow pat}_{type}]_{l\backslash\mathbf{exec}\backslash m\backslash} \parallel \mathcal{R} \longrightarrow_c \mathcal{R}'}$$

$$\frac{N \parallel \mathcal{R} \longrightarrow \mathcal{R}' \wedge op(y) \stackrel{\Delta}{=} E \in D \wedge match(l\,path, pat, \mathcal{R})}{[x{=}\mathbf{rexec}^{sls}_{l\,path:\mathcal{S}} op{<}e{>}.P]_{url'} \| [\langle D\rangle^{url^{\perp}\rightarrow pat}_{type}]_{l\backslash\mathbf{exec}\backslash m\backslash} \| \mathcal{R} {\longrightarrow_c} \mathcal{R}'} \quad \begin{array}{l} op \notin Com \wedge \\ \mathbf{rexec} {\notin} dom(D) \wedge \\ path = path'\backslash n \end{array}$$

$$\begin{array}{rcl} N & = & ((\nu z)([\overline{z}^{\,sls\cup\{l:\mathcal{S}\}} e.z(x).P]_{url'} \parallel \\ & & (\nu t)[z(y).E\,\theta_1\theta_2\{\overline{z}^{sls\cup\{l:\mathcal{S}\}}/\mathbf{return}\}]_{l\backslash\mathbf{exec}\backslash m\backslash t\backslash})) \parallel [\langle D\rangle^{url\rightarrow pat}_{type}]_{l\backslash\mathbf{exec}\backslash m\backslash} \end{array}$$

$$z \notin \{t, y\} \wedge \{z, t, y\} \cap fr(sls, \mathcal{S}, e, P, url', E, url, pat, path, m) = \emptyset$$

$$\begin{array}{l} \theta_1 = \{l\backslash\mathbf{session}\backslash\mathbf{ns}\ /\ {<}\mathbf{session}{>}\}\{l\backslash\mathbf{application}\ /\ {<}\mathbf{application}{>}\} \\ \qquad \{l\ id(d(pat))\ /\ {<}\mathbf{phbase}{>}\}\{path - pat\ /\ {<}\mathbf{ipath}{>}\} \end{array}$$

$$\begin{array}{l} \theta_2 = \{x{=}\mathbf{acom}^{\mathcal{I}}_{url(l\,d(pat),rpath):\mathbf{ns}} e\ /\ x{=}\mathbf{acom}^{\mathcal{I}}_{rpath} e \mid \mathbf{acom} \in ACom\} \\ \qquad \{x{=}\mathbf{acom}^{\{l(url^{\perp}\hookrightarrow l,r):\mathbf{ns} \mid r\in rs\}}_{url(url^{\perp}\hookrightarrow l\,d(pat),ref):\mathbf{ns}} e\ /\ x{=}\mathbf{acom}^{rs}_{ref} e \mid \mathbf{acom} {\in} ACom \wedge rs {\neq} \mathcal{I}\} \\ \qquad \{\overline{x}^{\,\{l(url^{\perp}\hookrightarrow l,r):\mathbf{ns} \mid r\in rs\}} e\ /\ \overline{x}^{\,rs} e\} \end{array}$$

Table 6: New rules for auxiliary transitions of matched commands.

$$\frac{\begin{array}{l} [x = \mathbf{get}_{l\,path\backslash n\,:\mathcal{S}}\,.\,\mathbf{if}\ supported_{l',type(x)}\ \mathbf{then}\ \big(loc = loc_{l',type(x)}\,. \\ n' {=} \mathbf{rexec}_{loc\backslash\,:\mathbf{ns}}.\,\mathbf{rexec}_{loc\backslash\mathbf{exec}\backslash:\mathbf{ns}} x^{l\,path\backslash\rightarrow\backslash n'\backslash n}.\,\mathbf{put}^{sls\cup\{l:\mathcal{S}\}}_{loc\backslash n'\backslash n\,:\mathbf{ns}} e. \\ P\{n'/y\}\big)\ \mathbf{else}\ P\{\mathbf{err}/y\}]_{l'\,path'} \parallel \mathcal{R} \longrightarrow \mathcal{R}' \end{array}}{[y = \mathbf{lexec}^{sls}_{l\,path\backslash n:\mathcal{S}} e.P]_{l'\,path'} \parallel \mathcal{R} \longrightarrow \mathcal{R}'} \quad \begin{array}{c} \{x, loc, n'\} \cap \\ fr(P) {=} \emptyset \end{array}$$

Table 7: Rules for local execution.

communication could be merely expressed in terms of service-based invocations (that is of the middleware primitives) [Giovannini 2010]. In general our experiments showed the easy implementability of the middleware and its wide applicability to several execution environments and web technologies.

$$[\nu\, l\backslash\mathbf{session}\backslash\mathcal{S}.P]_{url} \parallel \mathcal{R} \longrightarrow [P]_{url} \parallel \mathcal{R} \qquad \mathcal{S} \neq \mathbf{ns}$$

$$\frac{[x = \mathbf{rexec}_{l\backslash\mathbf{session}\backslash:\mathbf{ns}}.P\theta_1\theta_2]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}{[\nu\, l\backslash\mathbf{session}\backslash\mathbf{ns}.P]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}\; x \notin fr(P)$$

$\theta_1 = \{l\backslash\mathbf{session}\backslash x \;/\; l\backslash\mathbf{session}\backslash\mathcal{S} \mid \mathcal{S} \in \mathcal{N} \cup \{\mathbf{ns}\}\}$
$\theta_2 = \{l\; optpath : x \;/\; l\; optpath : \mathcal{S} \mid optpath \in Path \cup \{\varepsilon\} \wedge \mathcal{S} \in \mathcal{N} \cup \{\mathbf{ns}\}\}$

$$[\neg\, l\backslash\mathbf{session}\backslash\mathbf{ns}.P]_{url} \parallel \mathcal{R} \longrightarrow [P]_{url} \parallel \mathcal{R}$$

$$\frac{[\mathbf{delete}_{l\backslash\mathbf{session}\backslash\mathcal{S}\backslash:\mathbf{ns}}.P\theta_1\theta_2]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}{[\neg\, l\backslash\mathbf{session}\backslash\mathcal{S}.P]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}\; \mathcal{S} \neq \mathbf{ns}$$

$\theta_1 = \{l\backslash\mathbf{session}\backslash\mathbf{ns} \;/\; l\backslash\mathbf{session}\backslash\mathcal{S} \mid \mathcal{S} \in \mathcal{N} \cup \{\mathbf{ns}\}\}$
$\theta_2 = \{l\; optpath : \mathbf{ns} \;/\; l\; optpath : \mathcal{S} \mid optpath \in Path \cup \{\varepsilon\} \wedge \mathcal{S} \in \mathcal{N} \cup \{\mathbf{ns}\}\}$

Table 8: Rules for session creation and destruction.

6 Conclusion

Concerning related work, the process algebra that we presented is, to the best of our knowledge: the first attempt to formalize the execution model of an operating system with process algebra; the first one expressing, via explicit representation of URLs and URL matching, the behavior of RESTful Services (furthermore extending it); and, independently of the adoption of services of the RESTful kind, the first one managing sessions (and session delegation) in the form of pairs session identifier and context/application it refers to as it happen in practice, e.g. with java based web technologies. In particular, concerning the formalization of RESTful services, [Zuzak et al. 2011] focuses on the abstract representation of resources as elements of a space and the modification to such a space caused by the HTTP methods, [Hernández and Moreno García 2010] on transition systems where transitions abstractly represent the execution of an HTTP method and the consequent state modification in the client, finally [Wu et al. 2013] concentrates on the formalization of REST service constraints (in order to check if they are stateless, cacheable, etc.).

Regarding locations, our approach is similar to [Riely and Hennessy 1998] for the tree structure of process locations: we however express the tree structure in a location name (URLs) and we are able to match and reuse part of it. Moreover we make use of the tree structure directly in the communication which is based, at the level of service invocation, on pattern maching on location names, and is encoded as inter-location pi-calculus communication. On the contrary in [Riely and Hennessy 1998] the communication is intra-location only.

Regarding session treatment, our approach differs from [Lapadula et al. 2007] because it represents the behavior of sessions via fresh session identifiers (and not as correlation data). Moreover it differs from that in [Boreale et al. 2006, Cruz-Filipe et al. 2013, Boreale et al. 2008, Vieira et al. 2008] where sessions are represented via fresh identifiers and managed implicitly at service invocation as in our approach, but are not associated to application contexts and data. More precisely, in such approaches all communications to be performed inside the session must be expressed jointly with the service invocation that creates the session in the form of a conversation. In our approach instead session identifiers are associated to locations l (contexts) and, when a session identifier is created by a service invocation (it is created server-side by the context l as in a Tomcat server) it is stored, together with the associated context l, in the client session table at client-side (as it actually happens in a browser). Every future invocation to that context will implicitly use again that session identifier until some service-side code will explicitly cancel it with a session cancellation command. Since session identifiers are associated to contexts, it possible to express low level mechanisms such as client communication inside different sessions in an interleaved way (by calling services belonging to different contexts) and session delegation (by specifying the context for which to delegate the client session) without having to manage explicitly session identifiers (they are implicitly managed when a service is called as in [Boreale et al. 2006, Cruz-Filipe et al. 2013, Boreale et al. 2008, Vieira et al. 2008]). On the other hand our approach does not include any high-level mechanism for dealing with data streams as in [Cruz-Filipe et al. 2013], pipes as in [Boreale et al. 2008] or context-sensitive message passing as in [Vieira et al. 2008].

Concerning future work, the develompent of a detailed, non ambiguous and executable formal description of REST services and WebOS middleware via process algebra opens the possibility for the automatic analysis of such systems with, e.g., model checking techniques. Moreover, the presented formal modeling, being it detailed and realistic, e.g., in expressing pattern based URL matching mechanisms and functioning of HTTP sessions, puts the basis for REST clients/services and WebOS middleware implementations (that can differ for each of the involved endpoint locations) to be checked for correctness. This could be enacted both with static code checking, e.g. based on so-called behavioural types, and with testing or monitoring techniques. The formal development of these techniques and of related software tools automatizing them is left for future work.

References

[Boreale et al. 2006] Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V. T., Zavattaro, G.: "SCC: A Service Centered Calculus"; Proc. of Web Services and Formal Methods, Third International Workshop (WS-FM 2006); Volume 4184 of LNCS, 2006, pp 38-57.

[Boreale et al. 2008] Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: "Sessions and Pipelines for Structured Service Programming"; Proc. of Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference (FMOODS 2008); Volume 5051 of LNCS, 2008, pp 19-38.

[Bravetti 2014] Bravetti, M.: "Formalizing RESTful Services and Web-OS Middleware"; Proc. of Web Services and Formal Methods, 10th International Workshop; Volume 8379 of LNCS, 2014, pp 48-68.

[Google Chromium] "Google Chromium OS Project", http://www.chromium.org/chromium-os

[Cruz-Filipe et al. 2013] Cruz-Filipe, L., Lanese, I., Martins, F., Ravara, A., Vasconcelos, V. T.: "The Stream-based Service-Centered Calculus: a Foundation for Service-Oriented Programming"; Formal Aspects of Computing, 2013.

[Fielding 2000] Fielding, R. T.: "Architectural styles and the design of network-based software architectures"; PhD Thesis, University of California, Irvine, 2000 (Chapter 5 Representational State Transfer (REST)).

[Giovannini 2010] Giovannini, P.: "Development of Rich Internet Applications with Asynchronous Communication in Pure Service-Oriented Paradigm", Bachelor's thesis (Supervisor M. Bravetti), University of Bologna, Italy, 2010.

[Gregori 2007] Gregori, C.: "A Comet Based Methodology for the Development of Java Applets with Asynchronous Communication and its Application to a Shared Whiteboard System", Master's thesis (Supervisor M. Bravetti), University of Bologna, Italy, 2007.

[Groote 1993] Groote, J.F.: "Transition system specifications with negative premises"; Theoretical Computer Science, 118(2):263-299, 1993

[Guidi 2008] Guidi, A.: "A Methodology Based on Java, Services and HTTP Remote Filesystem for the Development of Universally Executable Applications that Preserve Configuration and Data", Master's thesis (Supervisor M. Bravetti), University of Bologna, Italy, 2008.

[Hernández and Moreno García 2010] Hernández, A. G., Moreno García, M. N. "A formal definition of RESTful semantic web services"; Proc. of First International Workshop on RESTful Design (WS-REST 2010); ACM, 2010, pp 39-45.

[Lapadula et al. 2007] Lapadula, A., Pugliese, R., Tiezzi, F.: "Calculus for Orchestration of Web Services"; Proc. of 16th European Symposium on Programming (ESOP 2007); Volume 4421 of LNCS, 2007, pp 33-47.

[Milner 1999] Milner, R.: "Communicating and Mobile Systems: the Pi-Calculus", Cambridge Univ. Press, 1999.

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

[Riely and Hennessy 1998] Riely, J., Hennessy, M.: "A Typed Language for Distributed Mobile Processes (Extended Abstract)"; Proc. of Principles of Programming Languages (POPL 1998), pp 378-390.

[SOAP] World Wide Web Consortium, "SOAP Protocol Version 1.2", http://www.w3.org/TR/soap/

[Taioli 2008] Taioli, F.: "A Methodology Based on Comet Long-Polling for the Development, by means of Web Services, of Java Applets with Asynchronous Communication", Master's thesis (Supervisor M. Bravetti), University of Bologna, Italy, 2008.

[Torelli 2014] Torelli, M.: "Push Notifications in Android Apps: Use of GCM and an Alternative Solution Based on HTTP Long Polling", Master's thesis (Supervisor M. Bravetti), University of Bologna, Italy, 2014.

[Vieira et al. 2008] Vieira, H. T., Caires, L., Seco, J. C.: "The conversation calculus: a model of service-oriented computation"; Proc. of 17th European Symposium on Programming (ESOP 2008); Volume 4960 of LNCS, 2008, pp 269-283.

[Wu et al. 2013] Wu, X., Zhang, Y., Zhu, H., Zhao, Y., Sun, Y., Liu, P.: "Formal Modeling and Analysis of the REST Architecture Using CSP"; Proc. of Web Services and Formal Methods, 9th International Workshop (WS-FM 2012); Volume 7843 of LNCS, 2013, pp. 87102.

[Zuzak et al. 2011] Zuzak, I., Budiselic, I., Delac, G.: "Formal Modeling of RESTful Systems Using Finite-State Machines"; Proc. of Web Engineering - 11th Int. Conference (ICWE 2011); Volume 6757 of LNCS, 2011, pp 346-360.