File Managing and Program Execution in Web Operating Systems[*]

Mario Bravetti

Department of Computer Science, University of Bologna, Italy
E-mail: bravetti@cs.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 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). 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/services) in Web Operating Systems. We formally define the semantics of such middleware via a process algebraic approach.

1 Introduction

The widespread use of more and more powerful mobile devices, like smartphones, in addition to personal laptops, laptops used at work, etc... has led to the need of exploiting the Internet as a repository for storying personal information and applications/programs so to be able to use them from any of these devices, not to loose them in the case one of these devices is destroyed/stolen and not have to re-install/re-configure them when such personal devices are changed: personal cellphones/smartphones tend, e.g., to be changed much more frequently than laptops. These needs have led to the recent 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 operative systems, which allow the user to use application/programs 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 (Google docs) 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 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 *application components*, i.e. a (graphical) user interface, which is executed client-side and a back-end which consists of several *service components* 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 *application components*, which are resolved relatively to the romote directory URL from which the component has been downloaded, they can access remote resources/service components in their delployment 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 machine where the user is using it.

The aim of this paper is to propose an architecture and a design/implementation 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). The idea has been to perform, apart from taking some concepts and basic mechanisms from traditional web technologies, a complete fresh re-start in developing such a WEB-OS architecture. In particular, while mechanisms typical of a browser (and a server) are present, we abandon the usage of HTML as the "central" format (often also a bottleneck) for user-interfacing at the client-side, around which the client-side technologies must operate. Notice that presence of HTML in browsers is one of the main reason that led to the current success of interpretation/javascript based client-side web technologies over the virtual machine based ones like, e.g., java applets. In a WEB-OS, where execution/development of web applications, becomes the "normal" way of

executing/developing applications, we believe that interpreted technologies and HTML are no longer adequate as the only user-interface technologies. Front-end of applications are more naturally implemented by virtual-machine based graphical user interfaces, so to have interfaces with the same quality as the traditional standalone OSes applications. Such a belief is also along the lines of current technology trends: application development in smartphones (which use graphical interfaces) and development of rich Internet application development where execution outside the browser is often considered a preferred feature (e.g. in Java by using the Java Web Start feature).

A confirmation that the current trend is bringing towards the development of WEB-OSes and, more in general, platforms for executing installation-free applications deployed in the Internet which remotely preserve configuration and data, is given by the very recent presentation of the Google Chromium OS [2], which has been developed concurrently with the ideas/implementations presented in this paper. From the available information it seems however that such an OS does not follow the "radical" approach taken in this paper and somehow resambles browser functioning also on usage of HTML pages. A consequence is that it considers usage of interpreted/javascript technologies as the client-side application front-ends (as in Google docs).

The WEB-OS architecture presented in this paper, while being freely designed from skratch, allows for any current client-side/server-side web technology/languages to be used in developing front-end/back-end of programs (both interpreted and virtual-machine based for the front-end, of arbitrary nature for the back-end) and any possible format for data exchanged (XML, Java object streaming, Javascript JSON, etc...). The idea is to have a language/technology and data format independent middleware and an extensible set of client/server plug-ins one for each of the supported technologies, which are used to interface the middleware to application/service components developed with such a technology. We propose a model of such a middleware based on: a set of primitives for managing files and executing programs in WEB-OSes; and mechanisms for application/service component (un)deployment and execution.

In such a middleware, an important role is played by session managing as a mean for user authentication and for the realization of complex workflow/business process patterns [5]. To this end we will also consider session delegation as a basic mechanism of the WEB-OS middleware.

After presenting the basic architecture of the middleware we will propose a service-based implementation for it which uniformly combines file/resource managing with service component remote execution. We will do this by developing an extension (both at the conceptual and at the technical level) of restful Web-Services [3] which use HTTP and its request methods as the communication protocol. The adequateness of this implementation is confirmed by the current success and widespread use of restful Web-Services and of HTTP POST for interface-based service components combined with session-like mechanisms (based on session id or authentication token). Facebook and Twitter are, e.g., endowed with a wide set of services that are in this form.

In order to unambiguously present the behaviour/semantics of such a middleware and of applications it executes, we use a process algebraic [6,1,4] approach to formally define the semantics of the middleware primitives and component deployment/execution. The process algebra that we present is, to the best of our knowledge, the first one representing (via URL managing) the behaviour of restful Web Services (furthermore it extends such behaviour) and the first one managing (via primitives which do not deal explicitly with session identifiers as data) sessions in the form of pairs: session identifier and context/application it refers to.

The paper is structured as follows. In Sect. 2 we present the basic WEB-OS architecture, in Sect. 3 we present the service-based implementation, in Sect. 4 we present the formalization with the process algebra. In Sect. 5 we make some concluding remark.

2 Basic Architecture

We now need to introduce some terminology on URLs that we will use to present the basic architecture of the WEB-OS.

URLs can be of two kind: URLs ending with "\", [1] representing resource collections (directories) and URLs ending with an identifier, representing a single resource (e.g. a file). A URL of the first kind can be used as a "*base URL*" that includes several resources whose "URLs" have such a base URL as a syntactical prefix. A particular case of base URL is a "*context URL*" that is used to denote all the resources and services belonging to an entire (web) Application. We will also use the intuitive (even if a little improper) terminology of "*relative URL*" as: a pathname (an alternated sequence of names and "\" characters) starting with a "\" or not. In the former case the relative URL is called, more specifically, a "*root relative URL*": when the former is *resolved against a base URL*, we get an absolute (i.e. non-relative) URL by replacing the (non-context) pathname part of the base URL with that of the root relative URL; when, instead, the latter is resolved against a base URL, we get an absolute URL by just concatenating the two (or by performing the preliminar name eliminations in the case ".." are used). These notions will be formalized in Section 4.

Another important notion that we will use is that of session. We will assume that each (web) Application 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 will consider each Application (identified by a context URL) to independently manage sessions, which, hence, have application scope. From the client side, therefore, session information will be collected as set of pairs composed of a *session id* and the *context URL* the id refers to.

Figure 1 shows the WEB-OS architecture. The upper part of the figure shows deployed/active, i.e. under execution or waiting for method calls, application

---

[1] For denotational convenience, in this paper we use backslashes in URLs instaed of slashes, because we use slashes to represent replacements.
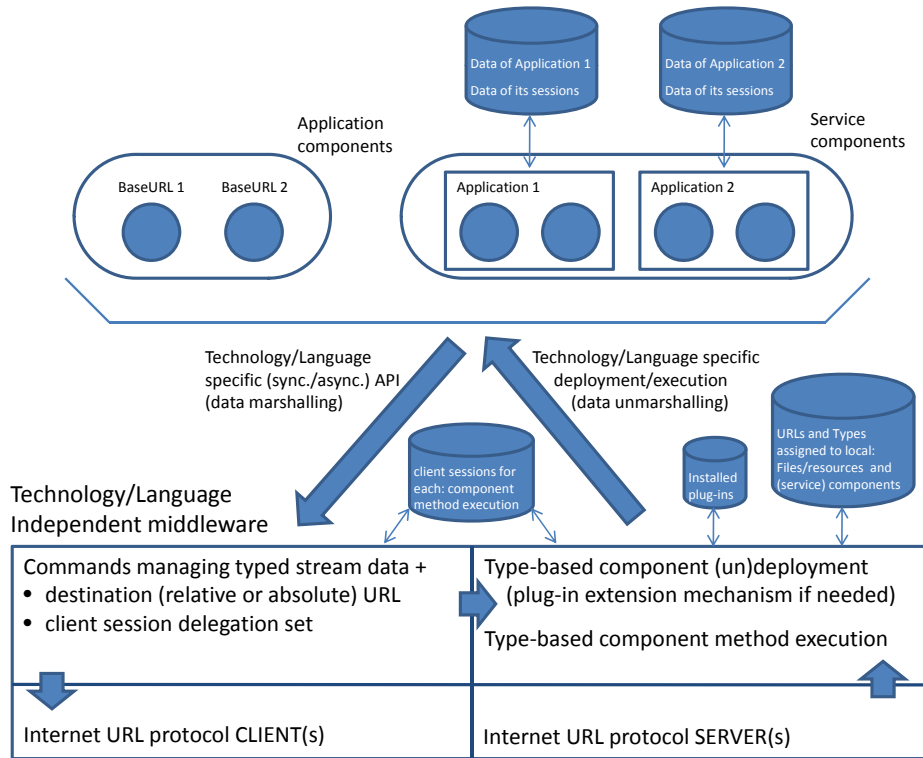
Fig. 1. Basic WEB-OS architecture

and service components (assuming that application components, that interact with the user by, e.g., some graphical user interface, have been installed by downloading them from some "Base URL").

Concerning the lower part, the language/technology independent middleware receives (via API) request to execute commands from application and service components. All such commands (that we will list in the following) receive as argument a (possibly empty) typed data stream (file) and return a (possibly empty) typed data stream (file). The file type determine the format of its content and can be implemented, e.g., by using the mime-type standard. Additional arguments of such commands are the command destination (absolute or relative) URL and session delegation set. The latter is a set of *context URLs* whose client sessions (if detained by the component execution calling the command) are to

be passed, besides the client session related to the destination URL (standard session managing), to a component which is, possibly, executed as the effect of the command. As we will see, the middleware maintains a set of detained client sessions (pairs session id and related context URL) for each execution of an (application or service) component. Client sessions are therefore not shared between components or between different execution of methods in the same service component, differently from what happens, e.g., in a browser.

Relative addresses are resolved differently depending on the invoking component: If it is an application component than the address is resolved against the code-base URL (i.e. the URL of the directory the application component was downloaded from) which is stored together with the application component when it is locally installed (denoted by "Base URL" in the figure). If, instead, it is a service component then the address is resolved against the physical-base URL i.e. the directory containing the URL used to execute the service component.

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. 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 method executions of (different) components, which are run in parallel.

Middleware basic commands (managing typed stream data) are the following ones:

– **Read** (inputstream), **write** (outputstream) or **delete** file/resources with the specified URL: writing creates or modifies and has a typed file as its parameter, while a successful reading returns a typed file.
– **Remotely execute** an operation of a service component with the specified URL: it takes a typed file as the parameter to be passed to the operation and returns a typed file as the operation result.
– **Deploy** or **undeploy** service components at the specified local URL: deployment takes a typed file containing the component code and, if needed, deployment infos as the parameter. Undeployment returns a typed file in the same form, so to allow for component migration.
– **Locally execute** an application component by reading the file with the specified URL which contains its code: it takes a typed file containing the parameter to be passed to the component execution/initialization and, if needed, deployment infos and it returns a typed file containing a local reference to the deployed component.

Optionally, the architecture could also include commands for execution of application component methods (besides direct interaction of the user with the application component user interface). In this case the reference returned by the local execution command (and which is then used as a destination address to execute

methods) must be a "typed" reference, i.e. a reference which makes it possible to determine the type of the component. Moreover, deployment/undeployment commands could be extended to work on any URL (and not just local ones) so to realize romote mechanisms typical, e.g., of a CLOUD system. However it is maybe more realistic to think about romete (un)deployment of components as a higher level mechanism that is not directly implemented by the middleware via a unique command and that could be technology dependent in the use of the related Internet protocol(s) (it could involve invocation of serveral middleware commands in a technology dependent way).

Command execution is based on direct usage (as a client) of the protocol specified in the URL address and/or, in the case of service components (un)deployment and application component local execution (and, if considered, of subsequent calls to its methods), of the middleware functionalities presented in the right-hand side of figure 1.

Concerning client URL protocol invocation, the command performs a request containing the typed file parameter, if any, and (by analyzing the client sessions detained by the method that invoked the command): the session id of the client session (if any) associated with the destination URL and the session ids of the client sessions (if any) associated with the session delegation URL contexts (the latter information is passed as pairs session id and related context URL). The response, symmetrically, is expected to include, besides a possible typed file or error message, a possible new status of the client session related to the destination URL (as standard) and of the delegated client sessions (which are delegated "back" at the response). The former is returned to the caller of the command, while the latter is used to update the status of the set of client sessions detained by the method that invoked the command.

Concerning the right-hand side of figure 1, service 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 service 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 service 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. Similarly, application component local execution is performed by doing, at first, a deployment of the application component based on the type of the component to be deployed: as for service components, if the application 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 application component is deployed by using the plug-in corresponding to its type, the component is executed/initialized by exploiting the same plug-in. Before actual application component execution, the middleware associates a set of detained client sessions to such an execution (which is initialized with standard and delegated client sessions passed with the command) and maintains it until the end of

the execution. If subsequent invocations of the application component methods are possible, the typed reference identifying the component will cause again the same plug-in to be exploited.

As we already mentioned, the WEB-OS middleware is also endowed with server(s) for protocols of URL addresses of files/resources and service components it exposes to the Internet. According to the client invocations we described above, such server(s) receive requests to read, write, delete files/resources and to remotely execute a method of a service component. Concerning files/resources, we just assume that such server(s) 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. Concerning service components, the server exploits the middleware functionalities to execute the required method via the plug-in corresponding to the component type. Similarly as for application components, before actual method execution, the middleware associates a set of detained client sessions to such a method execution (which is initialized with standard and delegated client sessions received with the request) and maintains it until the end of the execution.

Notice that, in the case middleware commands are used 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: in the case the addressed URL is local we can avoid invoking the protocol client and, instead, directly perform operations on the local resources/filesystem and/or invoke directly the right-hand middleware functionalities.

Finally, concerning the interface of the middleware with the technology/language dependent part of the WEB-OS, which is realized by an extensible set of service or application technology plug-ins:

- The (service or application) component code invokes the middleware commands via a technology/language dependent API which also performs data marshalling.
- The middleware executes (methods of) service or application components by performing technology/language dependent required mechanisms and also performing data unmarshalling.

The type of file used for data marshalling (e.g. XML or JSON for Javascript) and unmarshlling depends on the plug-in associated with the component type and on its configuration.

3   Service-based Implementation

We propose a service-based implementation for the basic architecture presented in the previous section. In particular the implementation that we propose aims at adopting a uniform mechanism for managing both file/resources and service

components. In the following we will call "operation" a method of a service component.

The needed mechanisms for dealing with files/resources are provided by so-called restful Web Services [3]. In Restful Web Services the HTTP protocol and HTTP request methods GET,PUT and DELETE are used to manage with the resource identified by the URL address of the request.

In the most simple 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), 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 component which is associated to a *pattern* in the form "$\backslash pathname \backslash *$": all requests to URLs whose (non-context) pathname matches the pattern (where "$*$" stands for any possible suffix) cause an execution of the associated service component which possesses an operation for each request method (for simplicity we assume the executed operation to have the same name of the corresponding request method). 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 component can be of two kind: the resource collection kind (the URL ends with "$\backslash$") and the single resource kind (the URL ends with a name). On an URL of the former kind it is possible to also use the POST HTTP request method with the following intended behaviour: a subresource (a resource whose address is "$URL\ name$" or "$URL\ name\backslash$" for some "$name$") is created whose content is passed as the body of the request (as for PUT) and whose name is a "new" (fresh) name established by the server.

Notice that, in restful Web Services, it is not mandatory for service components to manage several resources: the pattern "$\backslash pathname \backslash name$" (or just "$\backslash name$") can alternatively be used, which matches just a single resource. In this case operations of the service component do not receive a resource identifier, but just manage the single resource represented by the only matching URL.

The interpretation of service components as manager of resources adopted in restful Web Services leads to service components endowed with an operation for each HTTP method. This approach can be smoothly extended so to encompass also interface-based service components, i.e. those of the kind we considered in the previous section, which: possess any number of operations with arbitrary names, are invoked by specifying the name of the operation and do not possess a pre-defined intended semantics. Moreover, this can be done by preserving the interpretation of service components as manager of resources. To better understand this, we make a parallel with object oriented programming. The resources managed by a service component can be seen as objects of a class (in this case the resource is the memory occupied by object's fields) and service component

operations as methods defined in the class. In this view, URL suffix identifying the resource plays the role of the object reference which is passed to service component operations similarly as the reference "this" is passed to class methods. Moreover operations with the name (and meaning) of the HTTP request methods correspond to getter, putter, constructor and destructor methods, i.e. methods which in a class/component have a special pre-defined meaning. The presence of such special (related to object resource managing) methods in object classes, does not prevent them to also possess (non static) method with arbitrary names which act on the object resources in an arbitrarily-defined manner.

Technically, such user defined operations can be added to the restful Web Service paradigm by exploiting POST requests on single resource URLs: the body of the request contains the invoked operation name and the parameter typed file; the body of the response the typed file with the return value. By following this approach and by continuing the parallel with object oriented programming we have that:

– Statics of a class correspond to collection resources managed by a service component, which besides being themselves managed as resources have the ability to construct new subresources via POST (as for the static constructor method) and return a reference to them.
– Non-statics of a class correspond to single resources managed by a service component, which besides being themselves managed as resources can be manipulated by arbitrarily-defined operations invoked via POST.

Summing up, the service-based implementation is based on the HTTP protocol. The middleware commands described in the previous section are implemented as follows. We have GET, PUT, DELETE commands which cause HTTP client requests with the corresponding method to be performed and the REXEC (remote execution) command that cause a POST HTTP client request to be performed (and the parameter file sent as the request body). LEXEC command retrieves the typed file with the application component with a GET request. From the HTTP server side, when a request is received, the destination URL is checked for matching with patterns of deployed components and, in case of successful matching, the longest matching pattern is selected (deployment of multiple components with the same pattern is not allowed) and the requested operation (that corresponding to the HTTP request method) of the associated service component executed. [2]

Notice that, the extension of restful WS we are considering, which allows arbitrarily-defined operations to be invoked via POST, makes it meaningful to use sessions and session delegation. This because, while in restful WS [3], since the only allowed operations on resources are construction, destruction and getter, putter methods (i.e. those that also in object oriented programming are "special" methods with a predefined intended semantics and which provide the fundamental mechanisms to manage resources in terms of object memory), it

---

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

does not makes sense to make their behaviour/semantics to depend on sessions (i.e. on data recorded by previous operation invocations by the same client); in our extended setting, instead, since we are considering also arbitrarily defined additional methods, it is correct to have their semantics to possibly depend on data recorded by previous invocations.

We guarantee that service components possess an operation for each request method by defining default behaviours (which are performed in the case no operation is explicitly defined). In the case of GET, PUT, DELETE requests the default behaviour is to perform the corresponding action on the local filesystem at the corresponding location (according to a mapping of the URL context prefix into the local filesystem) and ignoring session information, including delegation, passed with the request. Concerning a POST request the default behaviour of the invoked component REXEC operation is the following. If the destination URL is a resource collection it generates a typed file (or directory) with a fresh name in the local filesystem directory corresponding to the destination URL containing the body of the request and it responds with the generated name (ignoring session information, including delegation, passed with the request); if, instead, it is a single resource, it expects the body to be a combined file (e.g. a combined mime-type) containing an operation name "*op*" and a parameter file, it invokes operation "*op*" of the service component and it responds with the returned value.

When defining GET, PUT, DELETE, REXEC operations in service components it is possible to invoke default versions of commands and explicitly pass the URL suffix to them, that identifies the resource they must manage. A typical use of this pattern is to use the received session to check the identity of the client and then to invoke the default behaviour.

Notice that, it also makes perfectly sense to add arbitrary defined operations, besides construction af a new subresource, also to collection resources (which would correspond to static methods different from the constructor in object oriented programming). This is realizable by just modifying the default behavior of REXEC so to manage combined "*op*"-parameter file requests also on collection resources.

Concerning service component deployment and undeployment commands, they must now associate a URL pattern (and not just a single URL) to deployed component.

Some considerations are now in order: the adoption of a resource based approach (originating from restful Web Services) allows us to manage general resources via URL (e.g. databases) and not just files. This is made possible by association of patterns to service component identifying the resources that it manages, i.e. service components are not identified by URLs which instead *just represent resources*: they are invoked by performing an action on one of the resources that they manage. In the case of a single resource pattern the action must be performed on the unique resource that they manage and, only in this case, we have a simple scenario where they are accessed by a unique URL. Effectively dealing with general resources is also made possible by the restful

Web Service mechanism of creation of fresh subresources, i.e. the intended behaviour of HTTP POST on collection resources. The extension we propose of usage of HTTP POST (REXEC) on single resources makes it possible to integrate resource managing and service component operation invocation (that looked like being dealt with separately in the architecture of the previous section) in a uniform resource-based mechanism. Finally, the capability of managing client session and client session delegation (introduced in the previous section) makes service component capable to implement complex workflows/businness processes. On the other hand practical applications (like facebook and twitter services) have shown the importance of user authentication/session managing via access tokens/session ids.

4 Formal Representation of Middleware Behaviour

We will now present the process algebra formalization for WEB-OS application behaviour which encompasses formalization of the middleware primitives and API invocation mechanisms.

In order to be able to cope with all the mechanisms we will make the following assumptions that allow for a more uniform treatment:

– Application components are deployed in the same way as service components: their reference is single pattern URL that can be used to invoke their methods via REXEC. Such methods can be used to abstractly represent interaction of the user with the components.
– Deployment/undeployment is performed by simple PUT and DELETE commands at special (local or non-local) URLs dedicated to delpoyment of application and service components (that act as manager of the resources located at the associated URL pattern). GET can be used for component migration.
– Commands in the process algebra are prefixes that wait for completion before performing the continuation code. This is meant to represent the behaviour of commands at the middleware level. If we are representing a technology/language with an asynchronous API we need to explicitly express in the process algebraic specification the asynchronous behaviour of the API with: multithreading and communication related to notification of command completion.

4.1 Syntax

We use $x, y, \ldots$ to denote generic names (identifiers) over a set $\mathcal{N}$. Moreover we use l, l' to denote locations overa set $\mathcal{L}$. Locations represent application contexts, i.e. a location l identifies both a server (IP address + port) and one of its application contexts.

The process algebra is an extension of pi-calculus [6] that represents the Internet as a network N of resources R deployed at some URL url: "$[R]_{url}$".

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

The restriction (νx) is used as in pi-calculus to represent the scope of a name (i.e. encloses the program/resources that have access to it).

URLS url are defined as pathnames starting with a context location l. We consider a special context directory called **exec** that we will use to deploy components, i.e. their code and their associated info, as the URL pattern they manage.

$$url ::= burl \backslash x \mid burl \backslash$$

$$burl ::= l \mid l \backslash \mathbf{exec} \mid burl \backslash x$$

We now also introduce relative pathes $rpath$ that will be used in the following, e.g., to represent "URL suffixes" which identify a resource belonging to a given pattern

$$rpath ::= bpath \mid bpath\ x$$

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

and patterns pat , that will be associated to components

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

Resources R can be either values v (typed files) or programs under execution P.

$$R ::= P \mid v$$

We will see the syntax of programs P in the subsection about semantics. Concerning values v we consider primitive values $pval$ which should at least allow us to represent successful and erroneous response from a request an numbers, names x, passive typed (where the type denotes their technology) application components $\langle D \rangle_{type}$ to be downloaded by LEXEC, deployed components $\langle D \rangle_{type}^{url^\perp \to pat}$, pairs $x < v >$ used to represent combination of operation/method name x and parameter value x passed to a REXEC, references ref that we will define in the following as URLs or relative pathes, etc..

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

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

where $url^\perp$ stands for either url or $\perp$. The two cases arise depending if the deployed component is an application component (in this case the url represents the codebase url) or not.

D is a declaration of a set of operations op, which are user-defined or commands, defined by

$$op ::= \mathbf{com} \mid x$$

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

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 $op(x) \stackrel{\Delta}{=} E$. 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 $x \notin fr(E)$ [3], i.e. no value is received by the $\mathbf{get}$ and $\mathbf{delete}$ operation definitions. In general, in the following, we will just use $op \stackrel{\Delta}{=} E$, to stand for a definition $op(x) \stackrel{\Delta}{=} E$ such that $x \notin fr(E)$.

We are now in a position to represent the syntax of terms E, i.e. code defining operations. We preliminary need some definitions

In method code we use urls starting with $<\mathbf{session}>$, $<\mathbf{application}>$ and $<\mathbf{phbase}>$ to manage session attributes of the current application and application attributes (for service components) and access the physical base (for application components).

We also use relative pathes starting with $<\mathbf{ipath}>$ to access the internal path, i.e. the URL suffix identifying the resource on which a service component is called.

References ref are possible way of expressing addresses (relative, root-relative or absolute) in a middleware command.

$$ref ::= url_s \mid rpath_s \mid \backslash rpath \mid \backslash \mathbf{exec} \backslash rpath$$

where url_s is defined as for url except that $burl_s$ replaces $burl$ and $rpath_s$ is defined as for $rpath$ except that $bpath_s$ replaces $bpath$. The syntax of $burl_s$ is as that of $burl$ with the additional production

$$burl_s ::= \cdots \mid < symurl >$$

where

$$symurl ::= \mathbf{session} \mid \mathbf{application} \mid \mathbf{phbase} \mid x$$

The syntax of $bpath_s$ is as that of $bpath$ with the additional production

$$bpath_s ::= \cdots \mid < \mathbf{ipath} >$$

.

We also need to introduce expressions. An expression e includes v possibly combined with operators and returns a v. A boolean expression be includes v possibly combined with 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 $\mathcal{E}(e)/\mathcal{E}(be)$.

Finally, we need to introduce session delegation sets rs, which specify a set of contexts for which we want the client session to be delegated. ε included in the list is a relative reference to the context of the base URL (codebase for application components, physical base for service components).

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

---

[3] We use $fr(E)$ to denote free names included in E.

Notice that the spacial case $\mathcal{I}$ of rs denotes an internal command, i.e. a usage of a default command behaviour inside a user defined command/operation. In this case the command expects a relative url identifying the resource it must work with.

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

where, for commands $x = \mathbf{com}^{rs}_{ref}e.E$ occurring in E, we have that $rs = \mathcal{I}$ implies $\exists rpath_s : ref = rpath_s$.

4.2 Semantics

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

In order to do this we need to extend the syntax of urls (terms $burl$), as defined in the previous section, so to represent session managing via special resources.

$$
burl ::= \cdots \mid l \backslash extrapath \mid x
$$

$$
extrapath ::= \mathbf{session} \mid \mathbf{session} \backslash \mathbf{ns} \mid \mathbf{application}
$$

S is a metavariable that can stand for a session identifier (a name x or no session ns, in the case no session is detained).

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

We need also to introduce session delegation pairs sls that are transmitted inside a service request.

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

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

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

Before presenting the semantics, we need to introduce pathes *path* as the path information that can occur in an url after the context.

$$path ::= \backslash rpath \mid \mathbf{exec} \backslash rpath \mid \backslash extrapath \backslash rpath \mid \varepsilon$$

In the following we will use $\mathcal{R}$ (resource collections) to denote terms N that do not include $(\nu n)N$ subterms.

The semantics is defined in tables 1, 2, 3, 4, 5 and 6. In the tables we assume the following definitions.

- $Int_g = \{l\backslash, l\backslash\mathbf{session}\backslash, l\backslash\mathbf{exec}\backslash \mid l \in \mathcal{L}\}$
- $Int_d = \{l\backslash, l\backslash\mathbf{application}\backslash \mid l \in \mathcal{L}\}$
- $maxpat(\mathcal{R}, path) = max \{pat \in pats(\mathcal{R}) \mid path \in pat\}$ [4]
  where $pats(\mathcal{R}) = \bigcup_{[R]_{url} \in \mathcal{R}} pat([R]_{url})$. $pat([R]_{url}) = \{pat\}$ if $R = \langle D \rangle^{cbase \to pat}_{type}$ and $url = l\backslash\mathbf{exec}\backslash m\backslash$ for any D, $cbase$, pat, l and m; $pat([R]_{url}) = \emptyset$ otherwise.
- $match(l\,path, pat, \mathcal{R}) = l\,path \in urls(\mathcal{R}) \wedge path \in pat \wedge pat \geq maxpat(\mathcal{R}_l, path)$

Moreover $d(pat)$ returns the path corresponding to the directory of the *pat* (after that $*$ has been removed), $\hookrightarrow$ returns the argument on the left if it is not $\bot$, otherwise it returns the argument to the right, $url(\_,\_)$ and $l(\_,\_)$ perform the expected evaluation of an (absolute) *url* and a context l starting from the absolute base *url* on the left and the relative url on the right. $path-pat$ computes the *path* suffix matching $*$ (or ε if there is no $*$) in the expected way.

Finally, the boolean function *cond* is assumed to be any predefined function such that, for any context l: $cond(l\backslash\mathbf{session}\backslash) = cond(l\backslash\mathbf{exec}\backslash) = cond(l\backslash) = true$. Moreover, $loc_{l,type}$ is assumed to be a predefined function that returns a context l' located at the same IP (in the same machine) as l, such that l' is capable to execute the client-side technology of type *type*. $type(x)$ determines

---

[4] We assume that ε is returned (the smallest element in the prefix relation) if we have an emptyest.

the type of value x: recall that values represent (content of) typed files. Hence we have $type(\langle D \rangle_{type}) = type$.

Notice that the negative premise in Table 2 does not cause bad definedness of the operational semantics. This because, in the rules for captured commands, the capability of a term of performing some auxiliary transition is **not** conditioned on the existence of some reduction transition $\longrightarrow$ (which could in turn depend on auxiliary transitions), this because the term $N \parallel \mathcal{R}$ considered in the premise can always perform a reduction transition (i.e. the synchronization between $\overline{op}$ and op).

Table 1. Congruence rules.

$$N_1 \parallel N_2 \equiv N_2 \parallel N_1$$

$$(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$$

$$((\nu x)N_1) \parallel N_2 \equiv (\nu x)(N_1 \parallel N_2) \ \ x \notin fr(N_2)$$

$$[(\nu x)P]_{url} \equiv (\nu x)[P]_{url}$$

Table 2. Basic rules.

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

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

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

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

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

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

$$[\overline{op}^{\,sls}e.P]_{url} \parallel [op(x).Q]_{url'} \parallel \mathcal{R} \longrightarrow$$
$$[P]_{url} \parallel [Q\{\mathcal{E}(e)/x\}\theta_1\theta_2]_{url'} \parallel \mathcal{R}$$

$$\theta_1 = \{l\, path : \mathcal{S}' \,/\, l\, path : \mathcal{S} \mid path \in Path \wedge\ l : \mathcal{S}' \in sls\}$$

$$\theta_2 = \{l\backslash\mathbf{session}\backslash\mathcal{S}' \,/\, l\backslash\mathbf{session}\backslash\mathcal{S} \mid\ l : \mathcal{S}' \in sls\}$$

Table 3. Rules for auxiliary transitions of uncaptured commands.

$$[x = \mathbf{put}^{sls}_{url:\mathcal{S}} e.P]_{url'} \parallel [v']_{url} \parallel \mathcal{R}$$
$$\longrightarrow_c [P\{\mathbf{ok}/x\}]_{url'} \parallel [\mathcal{E}(e)]_{url} \parallel \mathcal{R}$$

$$[x = \mathbf{put}^{sls}_{url:\mathcal{S}} e.P]_{url'} \parallel \mathcal{R} \qquad\qquad d(url) \in urls(\mathcal{R}) \cup Int_d \wedge$$
$$\longrightarrow_c [P\{\mathbf{ok}/x\}]_{url'} \parallel [\mathcal{E}(e)]_{url} \parallel \mathcal{R} \; id(url) \notin id(urls(\mathcal{R}))$$

$$[x = \mathbf{get}^{sls}_{url:\mathcal{S}}.P]_{url'} \parallel [v]_{url} \parallel \mathcal{R}$$
$$\longrightarrow_c [P\{v/x\}]_{url'} \parallel [v]_{url} \parallel \mathcal{R}$$

$$[x = \mathbf{delete}^{sls}_{url:\mathcal{S}}.P]_{url'} \parallel [v]_{url} \parallel \mathcal{R} \;\; \nexists url'' \in urls(\mathcal{R}) : url'' > url$$
$$\longrightarrow_c [P\{\mathbf{ok}/x\}]_{url'} \parallel \mathcal{R}$$

$$[x = \mathbf{rexec}^{sls}_{burl\backslash:\mathcal{S}} e.P]_{url'} \parallel \mathcal{R} \qquad burl\backslash \in urls(\mathcal{R}) \cup Int_g \wedge$$
$$\longrightarrow_c (\nu n)([P\{n/x\}]_{url'} \parallel \qquad\qquad n \notin fr(\{P,e,burl,url'\})$$
$$[\mathcal{E}(e)]_{burl\backslash n\{\backslash\}^{cond(burl\backslash)}}) \parallel \mathcal{R}$$

additional condition for each rule:

$$sls = \mathcal{I} \; \vee \; \nexists r \in \mathcal{R} : pat(r) = \{maxpat(\mathcal{R}, url)\} \wedge \mathbf{com} \in ops(r)$$

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

Table 4. Rules for auxiliary transitions of captured commands.

$$\frac{N \parallel \mathcal{R} \longrightarrow \mathcal{R}' \wedge op(y) \triangleq Q \in D \wedge match(l\,path, pat, \mathcal{R})}{[x = op^{sls}_{lpath:\mathcal{S}} e.P]_{url'} \parallel [\langle D \rangle^{cbase \rightarrow pat}_{type}]_{l\backslash\mathbf{exec}\backslash m\backslash} \parallel \mathcal{R} \longrightarrow_c \mathcal{R}'} \quad op \in Com$$

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

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

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

$$\theta_2 = \{x = \mathbf{com}^{\mathcal{I}}_{url(pat,rpath):\mathbf{ns}} e.P/x = \mathbf{com}^{\mathcal{I}}_{rpath} e.P \mid x = \mathbf{com}^{\mathcal{I}}_{rpath} e.P \in \mathcal{P}\}$$
$$\{x = \mathbf{com}^{\{l(cbase \hookrightarrow l,r):\mathbf{ns}|r \in rs\}}_{url(cbase \hookrightarrow l\,d(pat),ref):\mathbf{ns}} e.P/x = \mathbf{com}^{rs}_{ref} e.P \mid x = \mathbf{com}^{rs}_{ref} e.P \in \mathcal{P}\}$$

Table 5. Rules for sessions.

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

$$\frac{[x = \textbf{rexec}_{l\backslash\textbf{session}\backslash:\textbf{ns}}.P\theta_1\theta_2]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}{[\nu\, l\backslash\textbf{session}\backslash\textbf{ns}.P]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'} \quad x \notin fr(P)$$
$$\theta_1 = \{l\backslash\textbf{session}\backslash x \ / \ l\backslash\textbf{session}\backslash\mathcal{S} \mid \mathcal{S} \in \mathcal{N} \cup \{\textbf{ns}\}\}$$
$$\theta_2 = \{l\, path : x \ / \ l\, path : \mathcal{S} \mid path \in Path \wedge \mathcal{S} \in \mathcal{N} \cup \{\textbf{ns}\}\}$$

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

$$\frac{[\textbf{delete}_{l\backslash\textbf{session}\backslash\mathcal{S}\backslash:\textbf{ns}}.P\theta_1\theta_2]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}{[\neg\, l\backslash\textbf{session}\backslash\mathcal{S}.P]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'} \quad \mathcal{S} \neq \textbf{ns}$$
$$\theta_1 = \{l\backslash\textbf{session}\backslash\textbf{ns} \ / \ l\backslash\textbf{session}\backslash\mathcal{S} \mid \mathcal{S} \in \mathcal{N} \cup \{\textbf{ns}\}\}$$
$$\theta_2 = \{l\, path : \textbf{ns} \ / \ l\, path : \mathcal{S} \mid path \in Path \wedge \mathcal{S} \in \mathcal{N} \cup \{\textbf{ns}\}\}$$

Table 6. Rules for local execution.

$$[x = \textbf{get}_{l\, path\backslash n\, :\mathcal{S}}.l' = loc_{l,type(x)}.$$
$$n' = \textbf{rexec}_{l'\backslash\, :\textbf{ns}}.\textbf{rexec}_{l'\backslash\textbf{exec}\backslash:\textbf{ns}}x^{l\, path\backslash \to l'\backslash n'\backslash n}.$$
$$\frac{\textbf{put}^{sls\cup\{l:\mathcal{S}\}}_{l'\backslash n'\backslash n\,:\textbf{ns}}e.P\{l'\backslash n'\, /y\}]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}{[y = \textbf{lexec}^{sls}_{l\, path\backslash n:\mathcal{S}}e.P]_{url} \parallel \mathcal{R} \longrightarrow \mathcal{R}'}$$

---

5 Conclusion

We experimented a Java based partial implementation of the middleware, implementing our integrated version of interface-based and restful based web services, with comet related applications (application that send events in real time to the front-end interface) and we tested several solutions base on services keeping responses permanently open and long polling solutions. Moreover we experimented with the performance of several data binding methods.

References

1. Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, Gianluigi Zavattaro. SCC: A Service Centered Calculus. WS-FM 2006: 38-57.
2. Google Chromium OS Project, http://www.chromium.org/chromium-os
3. Roy T. Fielding. Architectural styles and the design of network-based software architectures. PhD Thesis, University of California, Irvine, 2000 (Chapter 5 Representational State Transfer (REST)).
4. A. Lapadula, R. Pugliese, and F. Tiezzi. Calculus for Orchestration of Web Services. In Proc. of ESOP07, volume 4421 of LNCS, pages 3347, 2007.
5. Web Services Business Process Execution Language Version 2.0 OASIS Standard, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
6. Robin Milner: Communicating and Mobile Systems: the Pi-Calculus, Cambridge Univ. Press, 1999
7. World Wide Web Consortium. SOAP Protocol Version 1.2, http://www.w3.org/TR/soap/