Customized Remote Execution of Web Agents

Titos SARIDAKIS Valérie ISSARNY Christophe BIDAN
IRISA / INRIA
Campus de Beaulieu
35042 Rennes Cedex, France
{saridaki,issarny,bidan}@irisa.fr

Abstract

Agent-based solutions are promising for ameliorating Web services, since they promote the modular construction of Web servers, relieve the network from transferring useless data, support user mobility, etc. However, existing Web servers do not favor the hosting of agents. In this paper, we propose a description of agent behaviors in terms of its requirements regarding resource utilization (e.g. memory, and disk space), functional services (e.g. system calls), and Quality of Services (e.g. degree of replication, and access control). These requirements, when formally expressed, can be used in an automated decision process which employs software specification matching techniques. On the acceptance of an agent, the host can use the agent's requirements to construct an environment customized to its execution. We discuss the benefits of this approach, and how it can be used to promote existing agent-based solutions in the Web framework.

1. Introduction

The Web architecture, being initially designed for the transfer of hypertext documents, has been stretched to its limits by applications from the fields of electronic commerce and distributed computing. Some solutions to consequent problems, primarily due to network saturation, have been based on *software agents*, a concept introduced to distributed systems from the field of artificial intelligence [1]. A software agent (or agent for short) is an autonomous piece of code with mobile characteristics that can be executed remotely. Agent-based solutions are attractive because they decrease network bandwidth utilization by migrating computations close to data, they provide for the construction of modular servers, and they promote the distribution of workload among the components of a distributed system. In the Web framework however, agent-based solutions raise some major problems, summarized in the following questions:

- Does a host provide all indispensable primitives for the agent's execution?
- Do the agent requirements conform with the host policies?

In this paper we assert that a host should be aware of agent's requirements before accepting it. We propose a scheme for coupling agents with their *execution properties*, where the latter term refers to the resources, and the functional and nonfunctional services engaged in the host-agent interaction. Our goal is to transfer existing knowledge from the field of formal specifications in distributed systems into the field of Web agents. We start by reviewing the benefits of formal specifications in todays distributed systems (§2), and then we analyze how existing technology can be used to promote agent-based solutions in the Web framework (§3). Our approach is illustrated through an example of a Web agent requiring reliable execution for accessing a remote database (§4). The paper concludes with an overview of our contribution, and a discussion on the applicability of the proposed approach (§5).

2. Background

Agents are similar to distributed objects in that they may perform some internal computations and they interact with the surrounding environment according to some well defined set of operations. This set consists of the *exported* operations that are the functionalities offered by the object to its environment, and the *imported* operations which are the functionalities the object expects from its environment. A distributed application can be constructed from a set of objects interconnected together by binding imported operations of each object to the corresponding exported operations of other objects. This results in a client/server architecture, where for each binding the object importing the operation is the client and the one exporting it is the server. A prominent advantage of this architecture is that an interface can be associated to each object, where exported (and sometimes imported) operations are declared. The interface hides the implementation details of the object from the programmer, who may interconnect objects and reason on the correctness of their interactions, based on their interface declarations.

Besides importing operations exported by some other application object, in some client/server architectures the client can also import a number of services offered by the execution platform, which acts as a server. For example, in the case of the CORBA [9], which extends the basic client/server architecture in many ways, the execution platform offers access control facilities, support for persistent objects, and transactional facilities, in the form of *Common Facilities* and of *Object Services*. Appropriate interfaces contain the declarations of the services offered by the execution platform.

By dissociating object implementations from object interactions, the use of interfaces facilitates the construction of distributed applications. However, the aggregate volume of code and its complexity makes reasoning about the properties of object interactions a tiresome job for the programmer. In the 80s, a number of solutions based on Hoare's logic [4] employed predicate logic to formally specify the properties of the objects. The formal specifications were used at compile time to detect inconsistencies on behavior requirements of the interacting objects (e.g. see [8]). Some recent approaches to distributed programming (e.g. see [6]) have adopted the use of formal specifications not only to verify the semantic correctness of the bindings, but also to aid the selection of the services provided by the execution platform.

A number of utilities have been proposed and implemented, to aid the programmer on using the formal specifications for reasoning on the correctness of object interactions. Such utilities employ *specification matching* techniques to conclude whether the guarantees of some object form an *exact* or *plug-in* match to the requirements of each of the object bound to it (e.g. see [12]). Depending on the code granularity that has been formally specified, these utilities can reason at various levels of interaction in a distributed system (e.g. binding, interface or object level).

3. Reasoning on Host-Agent Interaction

In the client/server architecture of the Web, an agent is the client that executes in the environment provided by a Web host, and the host plays the role of the server. Currently, the fact that a Web host considers agents mostly as hostile entities and decides on accepting them without taking into account their behaviors, results in a poor remote execution schema. As a solution, we suggest that interfaces should be associated to both agents and hosts, where they should declare the behaviors they exhibit during their execution. In the remainder, we assume the existence of an environment supporting agents and we explain in detail how the declared behaviors can be used in the decision for accepting an agent and in the construction of an execution environment customized to agent's requirements.

3.1. Functional and Nonfunctional Execution Properties

The formal specifications of the host and agent behaviors are called their *execution properties*, and encompass three different aspects of the host-agent interaction:

i. those regarding the *resources* utilization (e.g. memory, and disk space),

ii. those regarding the *operations* that an agent imports from or exports to its execution environment (e.g. system calls), and

iii. those regarding QoS requirements (e.g. degree of replication, termination deadlines, etc).

The first aspect refers to the *execution primitives* that define the size of the environment where an agent can be executed. The imported/exported operations describe the interaction points of the agent with its execution environment, which correspond to all potential bindings between the agent and the host. We call such execution properties *functional*, since they represent the exchange of functionality between the interacting sides. On the contrary, *nonfunctional* execution properties are those referring to QoS requirements. Nonfunctional properties do not represent any binding, but rather the QoS level required by a binding.

Functional properties can be declared in a CORBA-like interface which has been extended with two clauses, one for the imported operations and another for the declaration of the execution primitives that determine the "dimensions" of the execution environment. Figure 1 illustrates a possible form of interface declarations for an agent and a host, where the first three clauses (i.e. **export**, **import**, and **resource**) are used to distinguish among exported operations, imported ones, and execution primitives. We bring to the reader's attention that we do not distinguish the structure of host and agent interfaces. In general, we allow a host to declare an imported operation as much as an agent to declare an exported one, and we also expect primitives to be declared in both host and agent interfaces. The way the run-time system interprets the information in interface declarations, is discusses in §3.3.

The QoS requirements expressed by nonfunctional properties refer to the guarantees that bindings should eventually provide to the interacting sides. For example, a host may guarantee a certain degree of availability for a *read()*

```
interface agent-X {                 interface host-Y {
  export                              export
    Key, State;                         open, read, write, close;
  import                              import
    open, read, write, close;           Key;
  resource                            resource
    MEMORY = 15KB;                      MEMORY = 1MB;
    DISK = 40KB;                        DISK = 1GB;
  provide                             provide
    Key: Authentication;                read: Transactional;
    State: Checkpoint;                  write: Transactional;
  require                             require
    read: Failure-Atomicity;            open: Authentication;
    write: Failure-Atomicity;
}                                   }
```

Figure 1. Declaration of execution properties.

operation it exports and at the same time it may require a certain degree of security from the agent that imports the *read()* operation. Similarly, the agent importing the operation may have its own QoS guarantees and requirements for the corresponding binding. Hence, it becomes clear that the nonfunctional properties of an agent or a host are associated with some functional properties declared in their interfaces. One way to support this coupling of functional properties with their associated QoS requirements and guarantees, is by explicitly stating for each declared operation the nonfunctional properties that it provides and it requires. Figure 1 illustrates such a possible interface declaration, where previously declared operations state their nonfunctional properties in the clauses **provide** and **require**.

3.2. Using Formal Specifications

Until this point, we have considered the declaration of execution properties in terms of operation signatures, constant assignments to execution primitives (e.g. disk = 1GB), and of QoS "names" associated to declared operations (e.g. read: *Transactional*). The type-checking process typically employed in distributed programming environments to verify the correctness of bindings among declared operations is based on pattern matching. However, pattern matching techniques cannot be used to verify the satisfaction of the QoS requirements placed on bindings by the interacting sides, since each "name" declared in interfaces constitutes an informal description of some nonfunctional execution property. In such cases, specification matching techniques can be used to verify the correctness of bindings with respect to the required execution properties.

It should be obvious to the reader that we cannot rely on pattern matching performed on "names" describing the nonfunctional execution properties (e.g. *Authentication, Checkpoint, Transactional,* etc), to verify the correctness of bindings, because a slightly different interpretation of the same "name" by each of the interacting sides may cause communication problems difficult to track down and resolve. However, by associating to a "name" a set of formal specifications that serve as its definition, we allow both clean interface declarations and QoS requirements resolution based on specification matching (e.g. see [6]).

Adopting such an approach implies that the same set of primitive terms is used for the definitions of "names" in the vocabularies of both interacting sides, and also that the host possesses the knowledge to analyze agents' execution properties and to reason on the combinations of different degrees of QoS as sensible as security (e.g. see [2]). In [6], the definitions in the vocabulary are given in terms of a minimal set of primitive predicates expressing the transfer of data among interacting objects, and the same set of predicates is used to express the aforementioned knowledge as a set of rules describing the compatibility of nonfunctional properties, and their allowed combinations.

By integrating a similar system that performs typechecking on operation signatures and specification matching on QoS requirements (e.g. see [5]) with a standard Web server, we obtain a host which is able to consider the compatibility of agent execution properties with its own execution properties in order to decide on accepting the agent.

3.3. Interpreting Execution Properties

Specifying the execution properties of agents and hosts is the preparatory step for reasoning on their interactions. When an agent requests to be executed at a given host, the execution properties declared in the interfaces of both sides should be first interpreted and then the fact that all requirements are satisfied, should be verified. Following Hoare's logic, the required and provided behaviors can be seen respectively as the *pre-* and *post-* conditions of agent execution. In order to accept an agent, a host should verify that it can satisfy the pre-conditions, and that post-conditions conform with its policies. Obviously, an agent will not be accepted if it requires more resources than those available at the host, or if it imports an operation not provided by the host, or finally if some QoS requirements of either the agent or the host are not satisfied. More formally, the following formula should be verified in order for a host h to accept an agent a:

$$(RSC_a \leq RSC_h) \;\wedge\; (IMP_a \subseteq EXP_h) \wedge \\ (IMP_h \subseteq EXP_a) \;\wedge\; (REQ_a \cup REQ_h \prec PRV_a \cup PRV_h)$$

where RSC denotes the set of resources, IMP and EXP the sets of imported and exported operations, REQ and PRV the sets of required and provided nonfunctional properties, subscripts denote the entity (i.e. host h or agent a) to which these execution properties belong, and the symbol $\prec$ denotes a match with respect to the criterion used in the specification matching process.

The interpretation of the execution properties and the specification matching process should be done at the host side, each time that an agent's request for execution arrives. In case of a mismatch in execution properties, the agent cannot be accepted and different scenarios may be followed:

- The agent is rejected and a message is sent to notify its originator of the reasons of rejection (e.g. which QoS requirements produced conflicts with host policies).
- In the above case, the host may also send the execution properties that it is willing to dispose for the agent execution. In this case, the agent's originator can send another agent whose execution properties match those offered by the host.
- If the conflict is only due to the requested resources, the agent can be placed in a queue waiting for sufficient resources to become available, and then it can be executed.

The action to be taken in the case of a rejection of an agent, is a policy of the host and it may depend on different parameters like the load of the host and the identity of the agent. A discussion on the negotiation of agent properties is beyond the scope of this paper. Instead, in the remainder of this section we raise the problem of agent execution that conforms with the execution properties declared in its interface, and we present a solution borrowed from the domain of configurable distributed systems.

3.4. Customizing the Host Environment

Considering the agent's profile instead of treating it as a hostile entity facilitates its acceptance by the host. However, until now we have not considered issues related to the safety of the host, which in the Web community are considered much more important than the acceptance of an agent. By accepting an agent relying on the declared properties, we risk to accept an agent that actually exhibits a different behavior than the one it declares. In that case, the agent may cause damages to the execution environment. Obviously, for our approach to be viable, situations like the above should never raise.

The solution is to borrow techniques from the field of configurable systems, where an execution environment can be customized to the needs of an application [5]. In the context of Web agents, the application is the agent and the execution environment is the host. The execution properties declared by the agent allow the host to have a complete image of the requested execution environment, i.e. the exact points of interaction with the agent and their properties. Based on this knowledge the host can first verify that there is no conflict between its own policies and agent's requirements, and then construct a customized execution environment that provides the approved execution properties.

Besides the allocated resources (i.e. private memory and disk space) which the agent can access without any restrictions, the only way to access the host is to pass through the declared bindings conforming to the associated QoS constraints. Hence, the execution environment is safe for the host since it does not allow the agent to perform any actions other than those declared in its interface. On the other hand, the host can guarantee the correct execution of an agent whose behavior conforms with the one declared in its interface, while no guarantees exist for the execution of agents whose behaviors deviate from the declared ones.

We bring to the reader's attention that the customized environment does not necessarily provide *exactly* the execution properties required by the agent. It is also possible to provide some "stronger" properties that imply the required ones. Such cases include the allocation of a bigger portion of resources than the one required, the support for all ACID properties when only failure atomicity was requested, etc. In cases like these, the specification matching process verifies that the combinations of the agent execution properties with the additional execution properties offered by the customized environment do not produce some conflict with the host policies.

4. An Example

In this section we illustrate through an example how our approach applies to a usual case of a Web agent. We assume that the host owns a system which performs specification matching and configuration of the execution environment, and we study the scenario where the host builds a customized execution environment that matches exactly the functional properties requested by the agent, while it provide "stronger" nonfunctional properties than the agent's QoS requirements.

4.1. The Scenario

We assume that a number of enterprises have placed on the Web their product catalogs with various types of information including images, technical characteristics, prices, stock availability, possibility of home delivery, issues of magazines that have included the specific articles in comparative reviews, etc. Using today's Web technology, to obtain some specific information about a product, one has to visit all related sites and explicitly search into their catalogs for the desired information. Alternatively, the entire product catalogs can be downloaded from each site, and then searched locally for the desired information. Both cases are too time wasting since most of the information in the catalogs is irrelevant to the interests of the specific person.

The employment of Web agents in this case would significantly decrease the required network bandwidth, and the time spent by the user on searching and processing the desired information. An agent responsible for reading a catalog, selecting only specific information, classifying them according to some user-defined criteria, and sending it back to its originator, can be sent to the Web sites of the enterprises. Various deployment strategies can be followed according to the capabilities of the infrastructure supporting agents and the time constraints of the agent. For example, different copies of the same agent may be sent concurrently to different Web sites, or the same agent may visit the different Web sites, one at a time. In the remainder, we do not refer any further to agent deployment, but we focus on the acceptance of the agent by some host on the Web.

4.2. Accepting the agent

To make our example more specific, we study the case of an agent which reads the product catalog, processes the information, and sends the results by e-mail to its originator. The agent is also capable of performing checkpoint actions upon request from its execution environment, i.e. it can return a coherent internal state from which it can resume its execution if requested (e.g. in case of a failure). There are no time constraints on agent's execution, but the agent requires that any failures on the interactions with the host should be masked.

The interface of this agent is given in figure 2. The imported and exported operations are declared, followed by the declaration of the required resources. Then, the required and provided nonfunctional execution properties are respectively declared. We assume that *MaskFailure, ReadAccess* and *Checkpoint* are "names" defined in the vocabularies of QoS terms that are used by the host and the agent. The agent requires a mechanism to mask failures (*MaskFailure*) for all three bindings created on the imported operations. Moreover, since all accesses to files outside agent's private disk space are *a priori* prohibited, the agent requires a mechanism for the *read* operation which guarantees access for reading (*ReadAccess*) the specific file where the product catalogue resides. Finally, the agent provides the execution property named *Checkpoint* on the binding of the *SaveState* operation.

When the agent requests acceptance by some host, the run-time system that is responsible for accepting the agent analyzes its interface to verify that the host $i)$ has enough resources, $ii)$ exports all three requested operations, and $iii)$ supports all agent's QoS requirements. Hence, the agent described by the interface in figure 2 is accepted *if and only if* the host has at least 10KB of memory and 29KB of disk to offer to the agent, it allows the use of the *read, sed,* and *mail* operations by the agent, and its policies support failure masking and allow the agent to read the product catalog.

```
interface agent {
  import
    read, sed, mail;
  export
    SaveState;
  resource
    MEMORY = 10KB;
    DISK = 29KB;
  require
    read: MaskFailure;
    sed: MaskFailure;
    mail: MaskFailure;
    read: ReadAccess (http://www.siteX.com/cat.html);
  provide
    SaveState: Checkpoint;
}
```

Figure 2. The agent's interface.

4.3. Customizing the Execution Environment

Once the agent is accepted, the host should configure an execution environment according to the declared execution properties. First the requested resources should be allocated. Then, bindings are constructed for all operations imported by the agent by considering the QoS requirements associated to them.

For example, analyzing the agent's requirements for read access on the product catalog results in the insertion of a software component in the customized execution environment, which enforces the host access control policy to each use of the binding corresponding to the *read* operation. The satisfaction of access control constraints is further detailed in [2]. Similarly, analyzing the failure masking requirements results in the insertion of a software component which guarantees the successful termination of all the operations performed over the three bindings.

The exact behavior of the failure masking mechanism depends on the analysis performed by the host on the *MaskFailure* property. A possible analysis is the one informally described by expression (1), where failure masking is accomplished by checkpointing the agent's state and restarting the failed agent from its last saved state. If *MaskFailure* is analyzed according to it, then it is possible to build the customized execution environment by combining the operation *SaveState* exported by the agent which provides

Checkpoint with some host service providing *Restart*. The resulting customized execution environment is graphically illustrated in figure 3.

$$MaskFailure \equiv Checkpoint \wedge (failure \Rightarrow Restart) \quad (1)$$

$$MaskFailure \equiv Replicate \wedge Vote \quad (2)$$

If a rich vocabulary of execution properties' definitions is available, then more than one alternative definitions can exist for the same nonfunctional execution property. For example, the *MaskFailure* property can be defined by either of the rules informally described by the expressions (1) and (2). Each definition corresponds to a different mechanism for tolerating failures, and the one that will be finally chosen depends on various parameters.

If the host policies do not allow the use of the nonfunctional execution properties provided by the agent, and the host itself does not provide the *Checkpoint* property, then expression (1) cannot be used to guide the configuration of the customized execution environment. Then, an alternative decomposition like the one expressed in (2) can be used. In this case, the *Replica* property can be further analyzed into *replication in time* and *replication in space*.

In general, a decomposition of an execution property required for a given binding can be in conflict with some constraints placed on that binding (e.g. time constraints might conflict with *replication in time*). If all possible decompositions of some required execution property are eliminated due to such conflicts then the agent is rejected. In the case of more than one acceptable decompositions, the host may choose the cheaper to implement according to some specific criteria defined in its policies (e.g. the cheaper in terms of resource allocation, or in terms of time to configure the execution environment).

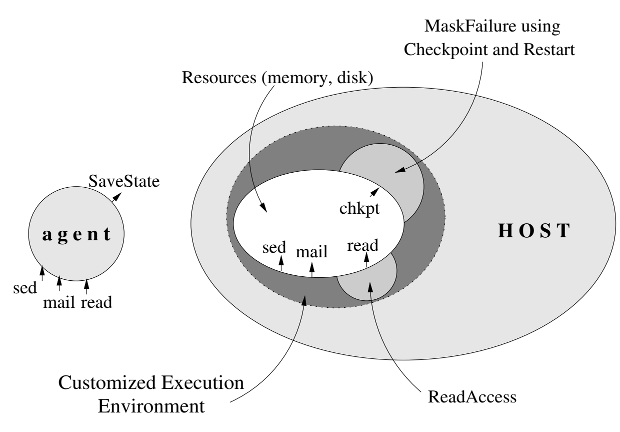

Figure 3. Agent's execution environment.

5. Conclusions

In this paper, we have proposed a schema for reasoning on the acceptance of agents and for constructing execution environments customized to agents' requirements. Our approach is based on associating interfaces to both agents and hosts, where they declare their execution properties. We aimed at transferring existing technology from distributed systems and formal specifications for software engineering into the Web, in order to promote the use of agent-based approaches to Web computing.

The proposed schema suggests that hosting an agent consists of verifying the compatibility of the agent's requirements with the host's policies, and then customizing the host environment to meet the approved requirements. Our proposal is easy to use in the simple case (i.e. the interface declaration for agents that do not have any QoS requirements is equally easy to the declaration of a CORBA interface), while it supports the declaration of complex execution properties without sacrificing the functionality, performance, or safety of the host (i.e. the customized execution environment guarantees both agent's requirements and host's constraints).

The proposed schema conveniently adapts existing technology from the fields of distributed systems and formal specifications to the needs of Web agents. Issues related to code mobility and its remote execution in foreign hosts can be uniformly expressed as execution properties. Execution properties integrate agent characteristics regarding both mobility support primitives (e.g. *attach, move*, and *clone* [7]), and QoS requirements (e.g. *authentication, availability, secrecy*, and *integrity* [11]). Our proposal can be beneficial for a number of different approaches to Web agents, like *HTTP-based Mobile Agents* [7], and *Mobile Assistant Programming* [10].

Besides the benefits for the agent, the presented approach provides support for modular constructions of host environments, which results in flexible and scalable Web servers. The customization of the agent's execution environment permits the use of resource management algorithms which allow the host to concurrently serve more than one agents without significant impacts on its performance (e.g. see [3]). Moreover, modular constructions allow the Web servers to apply different hosting policies, according to various criteria based on agent characteristics. As a consequence, a number of other research issues can be addressed in the Web agent framework, including how a host can use its current state for deciding to accept an agent, how to advertise, dispose of, and cost its resources, what combinations of QoS should the host accept, how to assure fair treatment of agents with equivalent requirements, etc.

References

[1] ACM Press. Intelligent Agents. *Communications of the ACM*, 37(7), 1994.

[2] C. Bidan and V. Issarny. Security Benefits from Software Architecture. In *Proceedings of the International Conference on Coordination*, pages 64–80, September 1997. Also available at http://www.irisa.fr/solidor/work/aster.html.

[3] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. In *Proceedings of the 5th International World Wide Web Conference*, pages 941–952, May 1996.

[4] C. A. R. Hoare. An Axiomatic Approach to Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[5] V. Issarny and C. Bidan. Aster: a Corba-based Interconnection System Supporting Distributed System Customization. In *Proceedings of the International Conference on Configurable Distributed Systems*, pages 194–201, May 1996. Also available at http://www.irisa.fr/solidor/work/aster.html.

[6] V. Issarny and C. Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 586–593, May 1996. Also available at http://www.irisa.fr/solidor/work/aster.html.

[7] A. Lingnau, O. Drobnik, and P. Dömel. An HTTP-based Infrastructure for Mobile Agents. In *Proceedings of the 4th International World Wide Web Conference*, pages 461–472, December 1995.

[8] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988.

[9] OMG. The Common Object Request Broker: Architecture and Specification – Revision 2.0. Technical report, OMG Document, 1995.

[10] S. Perret and A. Duda. Mobile Assistant Programming for Efficient Information Access on the WWW. In *Proceedings of the 5th International World Wide Web Conference*, pages 1373–1384, May 1996.

[11] J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transaction on Programming Languages and Systems*, 12(4):537–565, October 1990.

[12] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 6–17, October 1995.