# A Web-Based Virtual Machine for Developing Computational Societies*

Sergio Saugar and Juan M. Serrano

Department of Computing
University Rey Juan Carlos
{Sergio.Saugar,JuanManuel.Serrano}@urjc.es

**Abstract.** Different theoretical and practical insights into the field of computational organisations and electronic institutions has led to a clear separation of concerns between societal and agent-based features in the implementation of multiagent systems. From a theoretical perspective, this separation of concerns is also at the core of recent proposals towards a *societal* programming language. Building on the operational model of one of these proposals, this paper addresses the practical issue of implementing a web-based virtual machine for that language. The resulting framework is intended to be used in a wide range of applications, all of them related to the implementation of social processes (business processes, social networks, etc.).

## 1 Introduction

Different theoretical and practical insights into the field of computational organisations and electronic institutions [1, 2, 3, 4] has led to a clear separation of concerns between societal and agent-based features in the implementation of multiagent systems. For instance, the institutional platform AMELI [5] makes a precise distinction between programming the e-institution (using the language of the ISLANDER tool) and programming the agents which participate in the e-institution (e.g. using the AgentBuilder tool). From a theoretical perspective, this separation of concerns is also at the core of recent proposals towards a *societal* programming language [3], which complements the myriads of *agent* programming languages that can be found in the literature (e.g. Jason [6], 3APL [7], etc.). The former kind of languages are aimed at programming socially-enable middlewares, whereas the later are aimed at programming agentified software components.

The design of a language for programming computational societies involves two major tasks: specifying the abstract social middleware – i.e. the abstract machine to be programmed, and specifying its type system. In [3], some preliminary steps towards the first goal are taken. Particularly, the proposed operational model of social interactions precisely states the structure and dynamics

* Research sponsored by the Spanish Ministry of Science and Education (MEC), project TIN2006-15455-C03-03, and the Regional Government of Madrid and University Rey Juan Carlos, project URJC-CM-2006-CET-0300.

M. Klusch, M. Pechoucek, and A. Polleres (Eds.): CIA 2008, LNAI 5180, pp. 162–176, 2008.

of computational societies. Put in another way, it provides the structure and programmable behaviour of an abstract social middleware. Building on this theoretical results, this paper addresses the practical issue of implementing the language. The chosen middleware technology for implementing its virtual machine, i.e. the virtual middleware infrastructure, is the World Wide Web. This paper purports to present the architecture of a web-based social middleware infrastructure, namely the major design choices on its structure and dynamics. In order to attain this goal, the REST architectural style [8] and guidelines will be exploited.

The rest of the paper is structured as follows. Section 2 reviews from a middleware perspective the major concepts on the structure and dynamics of computational societies presented in [3]. Then, sections 3 and 4 address the architectural decisions on the structure and dynamics of a web-based social middleware infrastructure. The last section briefly summarises the major results and discusses current and future lines of work.

## 2 Computational Societies as Social Middleware Infrastructures

This section briefly reviews the operational model of computational societies put forward in [3] and introduces the example that will be used throughout the paper. Moreover, this operational model shall be interpreted as the abstract (i.e. technology-neutral) specification of a *social middleware* infrastructure. From this perspective, the major function of a computational society is to mediate the interactions among heterogeneous, distributed software components. In the next sub-sections, the kinds of *roles* played by software components attached to the social middleware as well as the primitive *interaction mechanisms* enabled by the middleware infrastructure will be considered. Moreover, the different types of external actions performed by software components over the middleware will be summarised. The last sub-sections introduce the abstract identifiers of middleware entities and an example within the university domain.

**Roles.** Software components interacting through an object-oriented middleware are published as *objects*; if the web is considered as the middleware infrastructure, software components play the role of *resources*; in a publish/subscribe infrastructure, components are attached to the middleware as *producers* and/or *consumers*; and so on. In regard with this feature, two kinds of roles are supported by a social middleware: *agents* and *resources*. On the one hand, resources represent those non-autonomous software components which store information and/or provide different computational services to the society. On the other hand, agents represent those autonomous software components which *purport* to achieve some goal within the society. In order to attain that goal, agents are able to perform different kinds of *social actions*, namely to say things to other agents (i.e. to perform communicative actions) and manipulate the environmental resources. The whole activity of some agent may be structured in a role-playing hierarchy of further agents (e.g. if its purpose is too complex).

**Social Interactions.** The interaction space of an object-oriented middleware is made up of *remote method calls*; the interactions through the web are handled in terms of *HTTP requests*; in the case of a publish/subscribe infrastructure, *event channels* are the primitive interaction mechanism. Concerning a social middleware infrastructure, its interaction space is hierarchically structured in terms of a tree of nested *social interactions*. In this way, the computational society itself is represented by the root, or top-level interaction. Social interactions provide the context within which agents and resources are deployed. Thus, a social interaction features a set of member agents, a set of environmental resources and a set of sub-interactions.

**External Actions.** External actions represent the interface between the abstract middleware and the external software components. Thus, a software component attached as an agent to the social middleware directs the behaviour of its agent through different kinds of external actions. For instance, the component may *play*/*suspend* its agent, thereby making public that the component is logged in/off the computational society. When the software component is logged in, it may *attempt* its agent to perform different kinds of *social actions*, e.g. *setting up* a new sub-interaction within a given context.

The social middleware deals with attempts in a three-stage process: firstly, it is checked whether the agent is empowered to do the specified social action (if it is not, the attempt is simply ignored); secondly, it is checked whether the agent is permitted to do the specified action under the current circumstances (if it is not, an event signalling the forbidden attempt is generated); last, if the agent is both empowered and permitted, the action is executed and the corresponding *events* signalling the updates in the social state are generated. These events may be notified to different agents according to their monitoring rules. In turn, these events may be pulled out by software components through the external action *observe*, which allows components to inspect the state of any social middleware entity[1].

**Social Actions.** The *set up* social action, mentioned above, is a communicative action (particularly, a declarative speech act) which is part of a predefined catalogue of standard social actions. This catalogue includes other actions, e.g., to prematurely finish a given sub-interaction (*close*); playing a new agent role within a given interaction context (*join*); and abandoning some played role (*leave*). Any kind of social action is *targeted* at some interaction whose state is intended to be modified (e.g. the target of a join action is the interaction context to which the performer intends to join). The protocol of the target interaction determines which agents are empowered and permitted to do that action.

**Abstract Identifiers.** Social middleware entities (agents, interactions, resources, etc.) have a unique abstract identifier. The abstract identifier of the

[1] The set of external actions mentioned above (*play*, *suspend*, *attempt* and *observe*) is complemented with other actions such as *enter* and *exit*, which deal with the registration of components to the middleware as software agents.

top-level interaction simply consists of a given *name*. Any other entity which is deployed within some interaction context is identified by a local *name* which identifies the entity within its interaction context, plus the identifier of its interaction context. Interaction identifiers are conventionally represented as a dot-separated sequence of local names, $n.n_c.\ldots.n_t$, which starts with its local name $n$ and is followed by the context identifier $n_c.\ldots.n_t$; the sequence ends with the name $n_t$ of the top-level interaction. Agent, resource, event and action identifiers are similarly represented. The only difference is that the name of the entity is separated from the context's identifier using the *at* sign ("@").

**Example.** Let's consider a social middleware to support the different social processes around the management of university *courses*. A given course, e.g. on data structures, is represented by a particular social interaction. On the one hand, this interaction is actually a complex one, made up of lower-level interactions. For instance, within the scope of a course agents will participate in *programming assignment groups*, *examinations*, *lectures*, and so on. On the other hand, courses are run within the scope of a particular *degree* (e.g. computer science), a higher-level interaction. Traversing upwards from a degree to its ancestors, we find its *school*, and finally the *university* (the top-level interaction). Besides schools, *departments* are also sub-interactions of the university. Taking into account the above structure of the interaction space, the identifier *ds.cs.si.urjc* stands for a course on data structures (*ds*) taught as part of the computer science degree (*cs*), managed by the school of informatics (*si*) at the University Rey Juan Carlos (*urjc*).

The agents within this computational society directly correspond to the different roles played by human users[2]. Thus, a student is represented by a role-playing hierarchy whose root is the *student* agent deployed within the degree; this student agent plays different student agent roles within the courses in which it has enrolled; in turn, students of courses may play corresponding roles within the programming assignment groups set up within their courses. Other agent roles, deployed within departments, include *associate professors* and *PhD candidates*, which play the roles of *teachers* and *teaching assistants* within courses, respectively. Concerning resources, we may consider different kinds of informational resources such as *programs* and *test cases*, generated by students within the context of working groups.

In the scenario that will be considered in the next sections, the agent *john@ds.cs.si.urjc* is a student of the course on data structures within the University Rey Juan Carlos. In order to pass the course (the purpose of course students), students have to pass several assignments in collaboration with another student. When the first assignment is published, *john*'s colleague sets up the working group *wg1.ds.cs.si.urjc*, which specifies *john* as an allowed partner. Then, an event representing this change is published and notified to *john*. When *john*'s human user observes these events, it attempts its agent *john* to join the assignment group. Then, since course's students are empowered to join working

[2] In this particular application, software components running the software agents are simply user interfaces, e.g. web browsers.

groups and *john* has been explicitly given permission, the action is executed by the middleware and a new agent *john@wg1.ds.cs.si.urjc*, played by *john*, is created within the assignment group.

## 3 Structure of a Web-Based Social Middleware Infrastructure

This section addresses the major design decisions concerning the *structure* of a web-based social middleware infrastructure, in accordance with the abstract specification introduced in the last section. The use of the web as the underlying distributed technology involves two major structural design problems:

- Firstly, computational societies must be published as web resources. These resources will represent the *entry* points to the social middleware for external software components.
- Secondly, different policies may be considered for the distribution of the interaction space through the network of web servers. Communication among socially-enabled web servers will rely on the previous *entry* points as well.

### 3.1 Publishing Social Entities as Web-Resources

There are several alternatives in order to expose a computational society through a web server. On the one hand, we may simply publish a single resource representing the whole computational society maintained by the web server. On the other, we may follow a fine-grained strategy and publish every major kind of social entity as a web resource. In order to leverage the HTTP protocol [9] to its full potential we follow the second approach. Thus, social interactions, agents, resources, actions and events are published as web resources. This allows, for instance, to implement the attempts of software components as HTTP POST requests over the agent resource, as described in the next section.

The URLs assigned to social entities follow general patterns which are designed after the structure of their corresponding abstract identifiers. Being exclusively based upon the data hold by abstract identifiers, the URLs resulting from these patterns are not expected to change very likely. Moreover, the URLs borrow the hierarchical and meaningfulness features of abstract identifiers as well. Table 1 shows the URL patterns assigned to the different kinds of social entities. Columns *Host:Port* and *Path* represent the corresponding parts of the URL. For every social entity, the *host* and *port* section of its URL represent the web server which manages that social entity. As will be described in the next subsection, with the possible exception of interactions, every entity is managed by the web server to which its interaction context belongs. The two rows of table 1 represent the two possible URL patterns:

- The first one is used for those interactions that are managed by a server different from its context's server. The URL for this kind of interactions is constructed by adding the name of the interaction to the root URL ( '/' )

**Table 1.** Generic Patterns of URLs

| Entity | Name | Context | Host:Port | Path |
|---|---|---|---|---|
| Interaction | *name* | null | server.com:port | /*name* |
| Interaction, Event, Resource, Agent, Action | *name* | *interaction* | server.com:port | /path/to/*interaction*/*name* |

of the server (note that the top-level interaction, whose context is empty, is a special case of this pattern).
- The second pattern applies to the entities that are published in the web server of its context interaction (i.e. events, actions, resources, agents and sub-interactions). The URLs of these entities are formed by appending the name of the entity to the URL of its interaction context (separated by '/').

### 3.2 Distributing the Interaction Space through Web Servers

We may consider two alternative stances on the distribution of the interaction space. The first one consists of ignoring this possibility so that the whole computational society is published through a single server. This means that this server processes all the HTTP requests over every social entity. In our scenario, this alternative forces a single host to process all the HTTP requests over the whole population of agents (students, teachers, etc.) and resources (assignments, solutions, plans of studies, etc.) of the university, as well as over its whole catalogue of social processes (courses, departments, schools, etc.). This alternative is only valid for applications with low demands for scalability, where the population of agents and resources as well as their interactions are kept under strict limits.

The second alternative, advocated by this paper, consists of allowing the distribution of the interaction space through multiple servers. The use of URL-addressable resources allows the distribution of the computational society over the Web, thereby exploiting its potential for scalability. The only constraint imposed on the distribution is that every social entity, but interactions, must be deployed within the web server which manages its interaction context. Sub-interactions may be deployed within the web server which manages its interaction context, but this is not mandatory. On the contrary, the URLs of agents, resources, actions and events always share the *host:port* part with the URL of their interaction context. Without any further restriction, and with the intention of guaranteeing the maximum deployment flexibility, every socially-enabled web server is allowed to manage a forest of interaction trees. Each of them may belong or not to the same computational society.

For instance, figure 1 shows a possible deployment of the interaction space corresponding to the scenario described in section 2. Particularly, it depicts the distribution of the major interactions throughout four hosts, each of them running a single web server. The first host, *www.univhost.com*, manages the top-level interaction *urjc*, representing the university itself. According to the patterns described in section 3.1, its URL is *http://www.univhost.com/urjc*. The second

host, *www.schoolhost.com*, manages that part of the interaction space which is under the primary responsibility of the school of informatics, namely the social interaction representing the school itself *si.urjc* and its different degrees. Besides the computer science degree *cs.si.urjc*, shown in the figure, other degrees such as software and computer engineering may be published through this host as well. The URLs of the school of informatics and the computer science degree are *http://www.schoolhost.com/si* and *http://www.schoolhost.com/si/cs*, respectively. The third host, *www.depthost.com*, is associated to the computer science department of the university, *csd.urjc*, published under the root URL of the host *http://www.depthost.com/csd*. In accordance with the statutes of the university, departments are in charge of the management of courses on the different subjects which are assigned to them. Thus, the course on data structures *ds.cs.si.urjc* is published through the computer science department under the URL *http://www.depthost.com/ds*. In this case, the web server of the department host manages two sub-interaction trees. The last host, *www.studenthost.com*, manages the working group set up by one of the students enrolled in the data structure course. The web server of the student's host may manage different sub-interaction trees from other computational societies as well (e.g. a discussion forum set up by the student within a social network).

## 4 Dynamics of a Web-Based Social Middleware Infrastructure

The dynamics of a computational society is primarily influenced by the external actions which software components execute over the social middleware which manages that society. Since the social middleware is implemented as a network of socially-enabled web servers, external actions are implemented as different kinds of HTTP requests. Moreover, the activity carried out by the middleware in order to process the different external actions heavily relies in the HTTP protocol as well. This is a direct consequence of the distribution of the computational society across the network of web servers. Therefore, HTTP requests may represent either an external action or some internal action executed by the middleware as part of the external action processing. Both kinds of HTTP request are handled through a pool of *conceptual* execution threads. These threads have a one-to-one correspondence to the different kinds of social entities. Thus, the agent execution thread processes every HTTP request whose target URL denotes an agent resource. Similarly, the interaction, resource, action and event execution threads manage the HTTP requests addressed to the corresponding kinds of social entities.

The remainder of this section proceeds to describe the implementation of the external actions mentioned in section 2. Particularly, it will be described both the way in which a given external action is represented as an HTTP request and the roles played by the different conceptual threads involved in its processing. The external actions *play* and *suspend*, related with *login* features, are taken into account first. Next, the mapping and internal processing of *observation* actions is

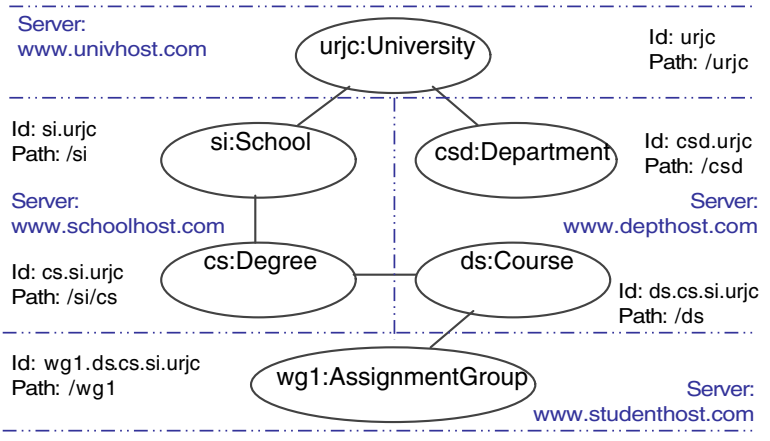


**Fig. 1.** Distribution of the example's interaction space

considered. Finally, we present the *attempt* processing cycle. Before delving into the different external actions, however, the major HTTP methods and response status are briefly summarised.

**Review of HTTP.** HTTP is a client-server protocol: a client sends a request message to a server, which does some processing and returns afterwards a response message containing a status code and the result of the request (or information about the status code). The format of a request message consist of a request line, zero or more header lines, and an optional message body. Both the standard semantics of status codes and HTTP headers are explained in [9]. A request line has three parts, separated by spaces: a method name, the local path of the requested resource (*Request-URI*), and the version of HTTP being used. A typical request is:

```
METHOD /path/to/resource HTTP/1.1
Header: value
...
Message-Body
```

*HTTP Methods.* The semantics of the request rely on the chosen HTTP method. We restrict our review to the four basic HTTP methods: GET, POST, PUT and DELETE.

- GET: This method is intended to obtain a representation of the resource identified by the Request-URI. It can be parameterized in order to constrain or restrict the desired representation.
- POST: This method is used both to create new resources and to append data to an existing resource. If the method is used to create new resources the body of the request will contain an entity. This entity must be created

by the resource identified by the Request-URI and the decision about the URL of the new entity is left to the server. On the contrary, if it is used to append data, the body of the request will represent data that must be added or processed by the Request-URI.
- PUT: The PUT method is used for creating a new resource (or updating the state of an existing one) under the supplied Request-URI. The message body of the request encodes the entity that will be published. If an entity already exists on the Request-URI, then the message body encapsulates an update of the entity (either full, affecting to the totality of their attributes, or partial).
- DELETE: This method unbinds a resource from the specified Request-URI. Note that this method does not imply the deletion of the actual data held by the resource or the software component behind it.

*Headers.* HTTP defines 47 headers which add optional meta-information about the Message-Body or, if no body is present, about the resource identified by the request. The most relevant header from the point of view of this paper are the following: *Authorization*, *Host*, *Location*, *Referer* and *WWW-Authenticate*.

*Status Codes.* Response messages to HTTP requests consists of a Status-Code element, some headers and a message body. The Status-Code is a 3-digit integer code which represents the result of the attempt made by the server to understand and satisfy the request. The first digit of the Status-Code defines the class of response (1xx informational, 2xx success, 3xx redirection, 4xx client error, 5xx server error). The body of response messages may give a short textual description of the Status-Code. Some of the codes we use in this paper are: *200 ("OK"), 201 ("Created"), 202 ("Accepted"), 204 ("No Content"), 400 ("Bad Request"),401 ("Unauthorized"), 403 ("Forbidden").*

***Play Processing.*** The external action *play* is used by a software component to initiate a logging session with a given agent, thereby obtaining the corresponding credentials to manipulate it. To log in is a mandatory requirement for performing some external actions such as *attempts*. Other actions, however, can be executed by non-logged components (e.g. *enter*, *observe*, the *play* action itself, etc.).

A digest access authentication scheme is proposed for dealing with credentials [10]. This scheme assumes that component credentials consist of the top-level agent name, a password defined when the agent was registered, a unique value shared between server and client and the MD5 algorithm. Once the component has got credentials, it may include them in every subsequent request using the *Authorization* header. The *play* external action is implemented as a GET request about the credentials of the agent, targeted over the URL of the agent.

```
GET /path/to/agent/credentials HTTP/1.1
Host: www.server.com
```

This action is processed by the agent execution thread. If some component has already initiated a session with the agent, the request is ignored and a response

with status code 400 is sent back. Otherwise, a response with status code 401 is returned. In this response, a WWW-Authenticate header includes the protected realm (the top-level agent), a unique value named *nonce* and a digest algorithm.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="top-agent's name",
                  nonce="84e0a095cfd25153b2e4014ea87a0980",
                  algorithm=MD5
```

In subsequent requests, the component composes a valid credential applying the MD5 algorithm to a combination of the top-agent's name, the nonce value and the password, among other parameters. The server does the same computation and if it yields the same credentials, it can be sure that the component is in possession of the correct password.

***Suspend Processing.*** Components can finalise its session with some agent with the *suspend* external action. This action deletes the nonce value associated with the component. Afterwards, the credentials of the server and the client won't match and the component's authorization credentials will be invalid. This external action is translated as a PUT action over the agent's credentials URL with a null message body.

```
PUT /path/to/agent/credentials HTTP/1.1
Host: www.server.com
```

This request is processed by the agent execution thread. This thread deletes the current nonce value, invalidating the following requests of the component. The response's status code is 204, because the server executes the action but declines to send back any representation. At a later time, a component may *play* again its agent, thus renewing its agent credentials.

***Observe Processing.*** Components can get representations of the different published entities (interactions, agents, resources, etc.) through the *observe* external action. This action returns a representation of the requested entity, restricted according to the visibility rules of the society. This external action is translated as a GET query over the URL of the entity. This query can be parameterized to select some parts of the resource instead of its full representation.

```
GET /path/to/entity HTTP/1.1
Accept:
Host: www.server.com
```

This request is processed by the execution thread corresponding to the kind of entity to be inspected. This thread checks the protocol for visibility restrictions and returns a response with status code 200. The message body of the response

contains the representation of the resource (maybe restricted according to its visibility permissions). This method can be executed by any component. If the component has initiated a session with an agent, the obtained representation will be tailored to the visibility permissions corresponding to agents of that type; otherwise, if the component hasn't got any agent credentials the obtained representation is the one associated by the protocol to agents of any type.

For instance, figure 2 shows a sequence diagram depicting the activity of the middleware in response to the scenario introduced in section 2. The roles displayed in the sequence diagram represent the different published resources; their lifelines describe the activity of the threads that manage those kinds of entities. The sequence diagram assumes that the student John has previously initiated a session as a student of the course on data structures (*ds*) with the web server `http://www.depthost.com` of the department of computer science, using a web `Browser`. The first message shows John *observing* the event queue of its agent *http://www.depthost.com/ds/john*, i.e. John's client producing the HTTP request *GET /ds/john?show=events* (message 1)[3]. Then, the response includes a representation of the event queue including the URLs of the events received by the agent (message 2). One of them refers to the new assignment group `http://www.studenthost.com/wg1` set up by its colleague. The remaining messages pertain to the processing of the attempt made by John to make its student join this assignment group.

***Attempt Processing.*** The external action *attempt* aims at *adding* a new pending action to the specified performer agent. According to this specification, this kind of external action is translated as a POST request over the agent URL. The attempt data (the action as well as other attributes) is included in the message body:

```
POST /path/to/agent HTTP/1.1
Host: www.server.com

<attempt>
 <action>...</action>
 ...
</attempt>
```

Message 3 of figure 2 represents an instance of the previous HTTP pattern. Particularly, it refers to the HTTP request corresponding to the attempt of John to make its student agent join the assignment group set up by its colleague.

An *attempt* HTTP request may refer to an action which is not targeted at some interaction managed by the same web server of the performer agent. Since the protocol of the target interaction must be consulted to check the empowerments of the agent, the agent execution thread processes an *attempt* HTTP request by requesting the target interaction to create it. Particularly, the request is actually

[3] Commonly, the request will actually be generated by the user interface components of the web browser, e.g. Java script code.

issued through a POST method over the target interaction URL[4]. The request includes the *referrer* header that indicates the performer agent of the action. The message body of this request is the action; the name of the action is set by the agent execution thread based on the reserved word "act", the performer's name and an incremental counter. Message 4 of figure 2 shows an instance of the following request pattern:

```
POST /path/to/target/interaction HTTP/1.1
Host: www.server.com
Referer: http://www.maybeotherserver.com/path/to/performer

<action name="act_performer_1">...</action>
```

If the performer is not empowered to do the action then a request with status code 400 is returned. Otherwise, the interaction execution thread creates the action for the specified performer agent – in the *referrer*'s server. This is encoded using a PUT action over the interaction context's URL. The content of the message body is the action as shown in the following scheme:

```
PUT /path/to/referrer/interaction/act_performer_1 HTTP/1.1
Host: www.server.com

<action name="act_performer_1">...</action>
```

For instance, message 5 of figure 2 shows the creation of a new action resource in the server of the computer science department, in accordance with the protocol's empowerment rules of assignment group interactions. Then, the action execution thread of the computing department server processes this request, creates the action, and sent back a response with a status code 201 with the action URL in the *Location* header (message 6). This response is forwarded by the interaction execution thread of the student server to the agent execution thread of the computing department server (message 7), which finishes the processing of message 4. Then, the student's execution thread sends a response back to the browser with a status code 202 and the corresponding *Location* header (message 8), which finishes the processing of the attempt HTTP request (message 3). This status code indicates that the action has been just accepted for execution. The *Location* header refers to the action resource as the monitor to check the processing state.

The action execution thread is responsible for the execution of the social action as soon as it is created. The way in which the action is executed depends on its semantics. Nevertheless, an HTTP request will be involved which must contain a *Referer* header with the URL of the action performer. For instance, the execution of a *join* action involves the creation of a new agent resource in

[4] Thus, the target interaction is acting here as an action *factory*.

the target interaction. Therefore, the corresponding HTTP request will be a PUT request which attempts to create a new agent resource. The message body contains the description of the new agent instance:

```
PUT /path/to/new/agent HTTP/1.1
Host: www.server.com
Referer: http://www.maybeotherserver.com/path/to/performer

<agent>...</agent>
```

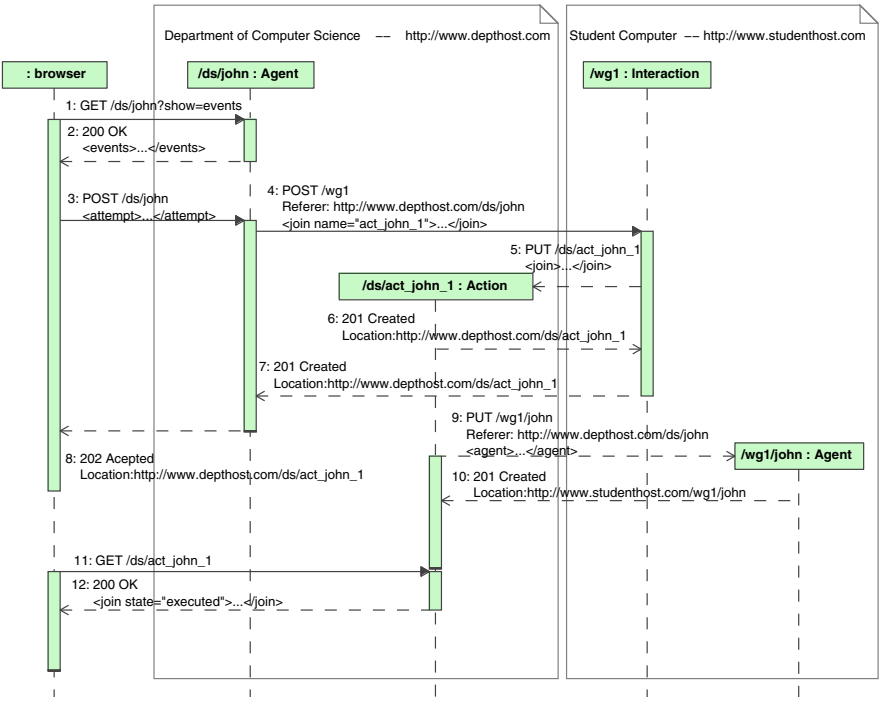


**Fig. 2.** Join to an Assignment Group

For instance, message 9 of figure 2 shows the PUT request issued by the action execution thread of the computing department server. This request aims at creating the agent `/wg1/john` within the assignment group of the student's server. The request is processed by the agent execution thread[5], which checks if the referrer agent (i.e. the performer of the action) has permissions to execute the action.

[5] Requests corresponding to other actions, such as *leave*, *set up*, *close*, etc. would be processed by other threads. For instance, *set up* and *close* involves PUT and DELETE requests over web interaction resources. Therefore, these methods would be processed by the interaction execution thread.

If it is not permitted then a response with a status code 403 is returned. Otherwise, the action is executed and a suitable response is generated. Message 10 of figure 2 shows the successful creation of the agent, which means that John's agent was permitted to join the assignment group. Then, the action execution thread changes the state of the action accordingly. The scenario is finished when John (actually, the web browser) *observes* the state of the action execution (using the URL monitor sent back in message 8), through messages 11 and 12.

## 5 Conclusion

This paper has put forward some of the major architectural decisions in the development of a web-based middleware infrastructure for the implementation of multiagent societies. Firstly, a computational society is published in the web through the refinement of web resources into three major sub-kinds: *web agents*, *web (institutional) resources* and *web interactions* (i.e. processes). Social actions and events are also published as web resources. Secondly, to account for *scalability*, *privacy*, and *flexibility of deployment* requirements, computational societies are allowed to be distributed across different socially-enabled web servers, each of them managing a forest of subinteraction trees. Last, in order to exploit the HTTP protocol in its full potential, the major HTTP methods (GET, PUT, POST and DELETE) are used to implement the external actions performed by software components towards the computational society, and the internal processing of the socially-enabled web servers.

One of the distinctive features of the proposed agent-based middleware infrastructure is the use of the web as the underlying middleware technology. On the contrary, other approaches face the web as a complementary – not fundamental – distributed infrastructure. In these contexts, the web appears in the issue of interoperability between agents and web service components (e.g. [11]). In our view, using the web as the underlying distributed infrastructure presents two major advantages: firstly, software components acting as agents within the society can be entirely *de-coupled* from the middleware server, which fits well with the autonomy requirement of agents; secondly, the web is one of the largest deployed distributed infrastructures, so that the network of social servers have not to be built from scratch.

Current work focuses on the implementation of the proposed architecture using the Restlet framework [12]. The resulting framework is intended to be used in a wide range of applications, all of them related to the implementation of social processes: business processes, e-government, e-democracy, etc. Particularly, we intend to demonstrate the feasibility and potential of the social stance of multiagent technologies on distributed computing, as well as the web-based approach to their implementation proposed in this paper, in the programming of social networks. Social networks like Facebook, Myspace, LastFM, etc., provides its users with different interaction mechanisms (chats, discussion groups, etc.). The modification and extension of these social networks essentially involves programming new social interactions. A web-based, social-oriented approach to the implementation of social networks would make this task much easier.

## References

1. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: An organizational view of multi-agent systems. In: Giorgini, P., Müller, J.P., Odell, J.J. (eds.) AOSE 2003. LNCS, vol. 2935, pp. 214–230. Springer, Heidelberg (2004)
2. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering and Methodology 12(3), 317–370 (2003)
3. Serrano, J.M., Saugar, S.: Operational semantics of multiagent interactions. In: Proceedings of the Sixth Intl. Joint Conf. on Autonomous Agents and Multiagent Systems, Honolulu, Hawai'i, 14-18 May 2007, pp. 884–891. ACM Press, New York (2007)
4. Esteva, M., Rodriguez, J.A., Sierra, C., Garcia, P., Arcos, J.L.: On the formal specifications of electronic institutions. In: Sierra, C., Dignum, F.P.M. (eds.) Agent-mediated Electronic Commerce (The European AgentLink Perspective). LNCS (LNAI), vol. 1991, pp. 126–147. Springer, Heidelberg (2001)
5. Esteva, M., Rosell, B., Rodríguez-Aguilar, J.A., Arcos, J.L.: AMELI: An agent-based middleware for electronic institutions. In: Proc. 3rd. Int. Joint Conf. on Autonomous Agents and Multiagent Systems, vol. 1, pp. 236–243 (2004)
6. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the golden fleece of agent-oriented programming. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms and Applications, Springer, Heidelberg (2005)
7. Hindriks, K.V., Boer, F.S.D., der Hoek, W.V., Meyer, J.J.C.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems 2(4), 357–401 (1999)
8. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Inter. Tech. 2(2), 115–150 (2002)
9. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol – HTTP 1.1 (1999)
10. Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L.: Http authentication: Basic and digest access authentication (1999)
11. JADE: Jade web services integration gateway (2007), `http://jade.cselt.it`
12. Consulting, N.: Restlet - lightweight rest framework for java (2007), `http://www.restlet.org`