XML Document Agents

Craig Sayers
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-288
November 12th , 2001*

XML document, agents, persistence

In this report, we describe a mechanism for creating lightweight mobile agents by embedding agent code within web documents. Each of our agents is a document, and each of our documents is an agent. As agents, they make use of well-defined messaging protocols to communicate, both with other document agents, and with external software agents. As documents, they have a natural persistence and web presence.

By making agents that are very lightweight, we can dedicate each agent to a particular narrow domain. For example, we could have one agent for each combination of user and task. Our goal is to support thousands of agents on a single host, while easing the job of developing and debugging them. In addition, each of our lightweight agents is encoded as an XML document, so all of the usual tools for interacting with such documents apply to our agents. For example: we create human-readable web pages by running an XML style sheet over the document agent; we query the state of an agent using standard Xpath/Xpointer expressions; and we persist agents by storing the XML in a file.

* Internal Accession Date Only
Approved for External Publication

XML Document Agents

Craig Sayers

Hewlett Packard Labs, Palo Alto, CA.

Abstract

In this report, we describe a mechanism for creating lightweight mobile agents by embedding agent code within web documents. Each of our agents is a document, and each of our documents is an agent. As agents, they make use of well-defined messaging protocols to communicate, both with other document agents, and with external software agents. As documents, they have a natural persistence and web presence.

By making agents that are very lightweight, we can dedicate each agent to a particular narrow domain. For example, we could have one agent for each combination of user and task. Our goal is to support thousands of agents on a single host, while easing the job of developing and debugging them. In addition, each of our lightweight agents is encoded as an XML document, so all of the usual tools for interacting with such documents apply to our agents. For example: we create human-readable web pages by running an XML style sheet over the document agent; we query the state of an agent using standard Xpath/Xpointer expressions; and we persist agents by storing the XML in a file.

HP Labs Technical Report HPL-2001-288,
Hewlett Packard Laboratories, Palo Alto, California, November 2001

1 Introduction

The World Wide Web is primarily described by HTML-encoded documents. Those were intended for human consumption. The markup tags define how the content is to be formatted, but not what the content represents.

Recently, with the move to XML encoding [1] and, in particular, the introduction of higher-level representations such as RDF [2], it is practical to provide web pages that contain machine-readable descriptions. In this evolution of the World Wide Web to the Semantic Web, web pages become a store of data to be mined by autonomous software agents [3].

We believe the next step will be to encode, not just the data about which agents may reason, but also the agents themselves, in web-accessible documents. In particular, we are exploring the notion of document agents, where every agent *is an* XML document.

1.1 Background

The idea of having mobile, persistent, agents is not new (see [4] for a great discussion). Constructing mobile code may now be achieved relatively easily in Java-based systems with the assistance of Java serialization [5], or in languages such as Scheme, through the transmission of a closure [6].

The idea of embedding software agents inside documents is also not new. For example, Bharat and Cardelli developed a system for "migratory applications" [7]. Their idea was to have user interface applications that combined the code and user interface within a migrating software agent. Migrating agents have even been applied to multi-media applications [8].

In our approach, the agent state is stored in a transparent XML format, the state is always present in the same document as the code and data (rather than only being present while the agent it being migrated) and the code is written in an extended version of ECMAScript [9]. In addition, our agents have a natural persistence (since all state is maintained in a persistent XML document) and have a natural web presence (since they exist as files which may be placed on a web server, and viewed through a web browser).

Similar prior work is that of the "follow-me" project [10]. They encoded an agent's "mission" using XML to describe input parameters and code. Relevant work using XML to describe agents has also been implemented by Lange et al [11]. They used XML to write the code of an agent, and have an elegant solution using a custom programming language with an XML syntax.

2 XML Document Agents

Each of our document agents is both an agent and a web-accessible XML document (see Figure 1). When viewed from the agent world, they look just like any other agent. If you send one an agent communication language (ACL) message [12], you'll receive a response just as you would from any other agent. When visible on a web server, they appear just like any other XML document. You can format them with style sheets [13], and you can inspect them with the usual Xpath/Xpointer expressions [14,15].

Each document agent includes its own data, code, and state. As the agent processes messages, it stores any updated data, code or state by modifying its own XML document. So, if we start with a document, X, the act of performing a single unit of execution is to generate a new document, X', which replaces the original. Thus there is no state outside the document. To clone or migrate an agent, we need only copy or move the document. To examine the running state of the agent, we need only examine the document. The document *is* the agent, and the agent *is* the document.

Since all of the code and state is stored locally, it is independent of other changes in the system. To see why this may be helpful, imagine that we require a user to fill out a sequence of forms. We would implement that by cloning a new document agent for each new combination of user and sequence. Each user's interaction is then with a particular custom agent. That agent remembers where their particular user is in the sequence of forms. The user may stop and resume at any time. More interestingly, if we needed to upgrade the system, we simply update the way in which we generate future document agents. Any previously-created agents are unaffected. Their users continue to see a consistent interface throughout the remainder of their particular sequence of forms.

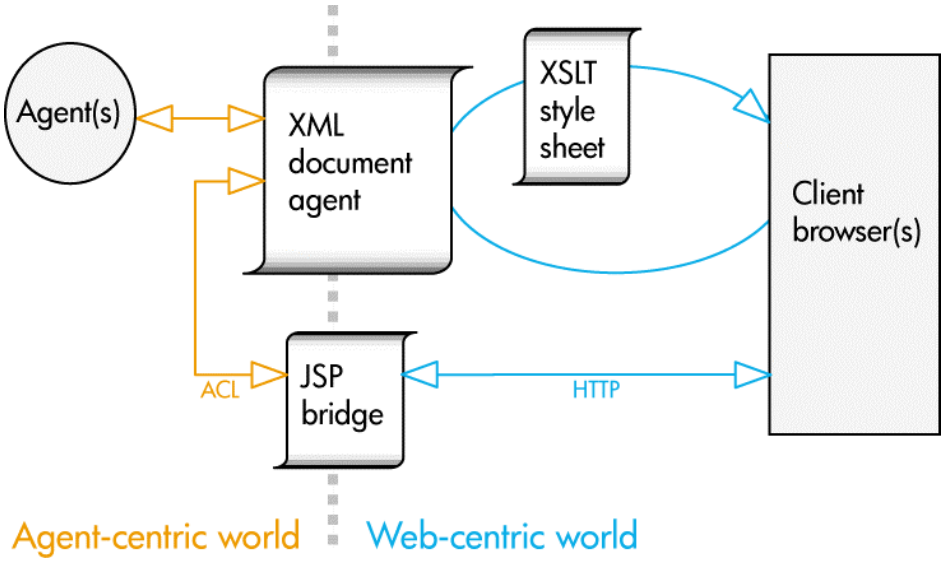

Figure 1. Each document agent coexists as both a document (that may be accessed via the web), and an agent (which may send and receive messages). In addition, we provide the option of having a document agent perform processing in response to web-based queries by adding a JSP [16] bridge that converts HTTP get/post operations into agent communication language (ACL) messages.

3 Agents as documents

Each of our agents is an XML document. Thus, all the usual operations on XML documents apply. For example, we can examine the state of the agent by using regular Xpath/Xpointer expressions to look inside the XML document.

To generate different "views" of an agent, we apply style sheets (XSLT) to the document. There are four basic implementations for this: client side, server side, agent-responsive, and agent-generated.

Client side

Link to a single style sheet from within the document agent; then a client can directly load the document agent and apply the style sheet locally. In this case, all of the processing is occurring on the client (just as it would for any other XML document).

Server side

Rely on a JSP to apply a style sheet to the document agent – this is more robust (we don't have to worry about incompatibilities in style-sheet processing among clients) and gives the option of dynamically selecting from among several style sheets. Here the processing is occurring on the server, rather than the client. Again, this is exactly the same procedure we'd use to do server-side styling of any other XML document (even if it were not an agent).

As in the previous case, the agent is not directly aware of these transactions. They are suitable for cases where an agent wishes to publish information, but need not be directly involved in each request for a copy. They are also suitable for debugging agents.

Agent-responsive

In this case, we give the agent an opportunity to perform processing in response to an incoming HTTP post/get. Each new request is converted to an agent communication language (ACL) message, and sent to the document agent. Then document agent then processes the message, updating itself in the process. The JSP waits for a response from the agent (indicating it has finished processing), before instructing the client to reload the page. That reload may then initiate either a server-side, or client-side, application of the style sheet (see above).

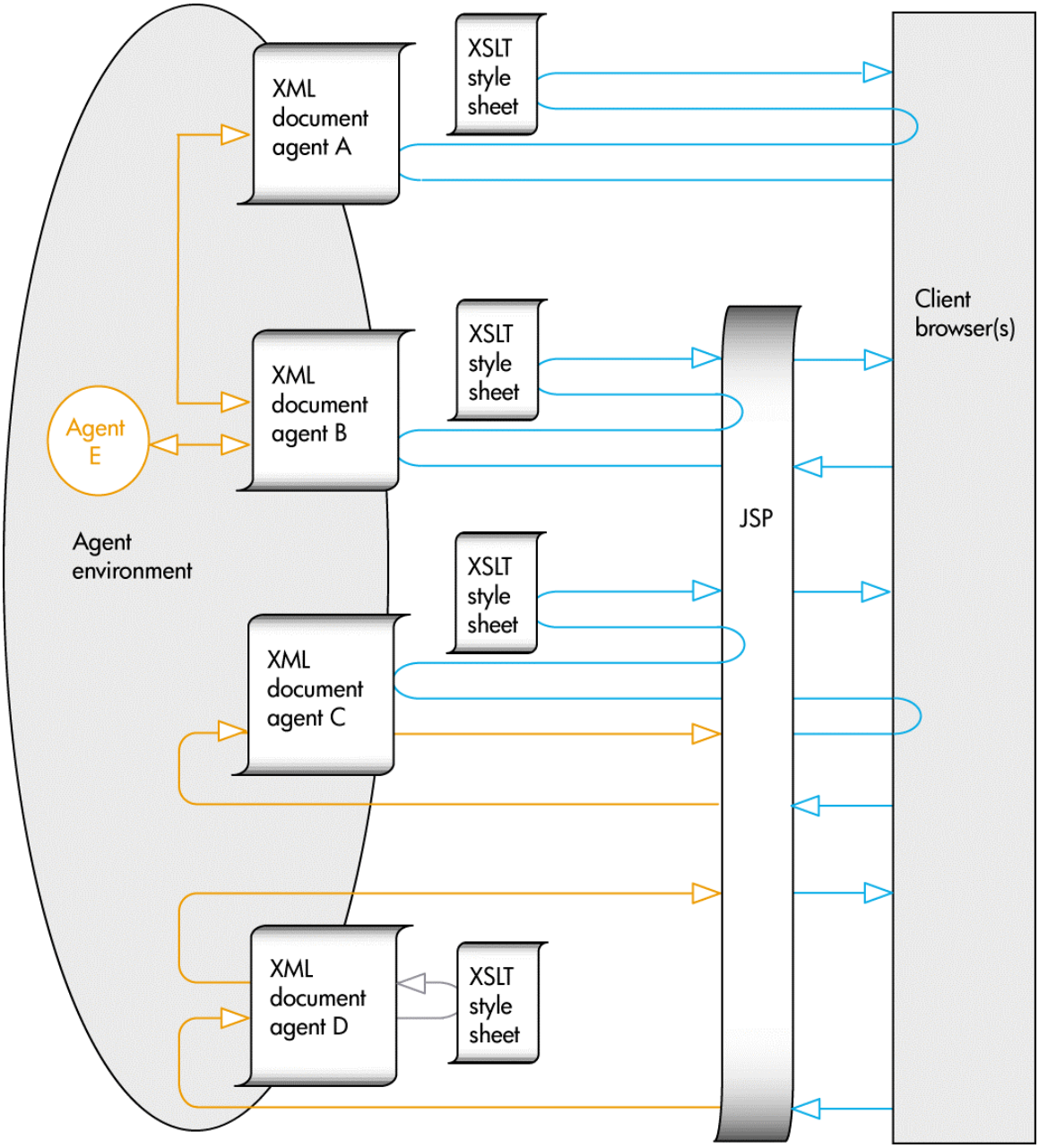

Figure 2. Overview of a document agent system. In this case there are four document agents. The client views agent A using client-side style sheets and agent B using server-side style sheets. Agent C is given the opportunity to perform processing in response to an incoming HTTP post/get operation, and agent D using agent-generated content. In addition, agent B is seen using the agent communication language (ACL) to communicate with both another document agent and an external agent (E).

Agent-generated

In this case, we again convert the incoming HTTP post/get to an ACL message and send it to the document agent. However, in this instance, we expect that the agent itself will respond with an ACL message containing the formatted document suitable for presentation. The agent has the option of simply sending itself, applying a style sheet to itself, generating content programmatically, or perhaps returning a small document which forces the client to reload an alternate page. This particular method is analogous to the way JSPs work [16], and is similar to the scheme used by Lange et al [11].

3.1 Persistent storage

We may store the XML document agents using regular files as the persistence mechanism. In that case, all of the usual operations on files naturally apply to our active document agents.

The act of migrating an agent is equivalent to moving the file (assuming, of course, that the machine you move it to is running a suitable document agent system). Similarly, the act of cloning an agent is equivalent to copying a file.

The last-modified date on the agent file serves as a convenient indication of recent activity by that agent. In our current implementation, we show a list of agents on the system, and color-code them based on how recently they were used. This is no more difficult than obtaining a directory listing.

4 Documents as agents

Each of our documents is a lightweight agent. It combines data, state, and code within a single XML document. The code takes the form of an interpreted language (in our current implementation, we use a version of ECMAScript [9] enhanced with a few custom extensions to support sending/receiving/interpreting ACL messages, and to simplify interaction with XML-encoded content).

In common with other agent systems, our agents communicate by passing messages. They may communicate both with other document agents, and with external agents (in our current implementation, built on top of the JADE agent platform [17], they may communicate with any FIPA-compliant agent).

Upon receiving a new message, the agent "wakes up", processes the message (optionally updating itself as a side-effect of that processing), and "goes back to sleep".

Waking up

Assuming it's not already "awake", the agent is awoken by reading the XML document into an in-memory DOM.

Processing incoming message(s)

To process a message, the system searches the DOM for an appropriate behavior node (one which contains a matching template for that message) and then executes the code within that node. That code may include instructions to remove a message from the input queue, to send one or more outgoing messages, and to modify its own DOM.

Recall that, if we start with a DOM, X, the act of performing a single unit of execution is to generate an updated DOM, X' which replaces the original. In this implementation, one unit of execution is equivalent to the processing of a single behavior (initiated by a single incoming message). A side effect of updating the DOM is to update the persistent store for the agent (in the current implementation, this involves writing the DOM out to a file).

In future implementations, and when using interpreted languages other than ECMAscript, it may be possible (and desirable) to reduce the size of an execution unit below that of a whole behavior.

Going to sleep

The agent is put back to sleep by simply removing the DOM from memory (it continues to reside in the persistent store). In some implementations, the agent may awake for each incoming message. In others, it may remain awake until some period of inactivity has occurred. Clearly there's a tradeoff here: agents consume more resources when awake, but also react more quickly to incoming messages.

5 Example – meeting assistant agents

In this toy example, we explore how agents could benefit the participants in a meeting. We provide each attendee with an agent to assist him, or her, in taking note of action items, and we provide the meeting itself with an agent to collect all those notes for public display.

In this system, there are several document agents (one for the meeting itself, and one for each attendee). Each agent supports a single view using a single style sheet.

The document agent representing the meeting maintains a list of action items. When it receives an incoming ACL message containing a new action item, it adds that to its list. (This is implemented by creating a node in the XML document to hold action items. When an incoming message arrives, the agent parses the XML describing that item, and appends it to that document node).

When that document agent is viewed in a browser, the client-side application of a style sheet formats the document (including the list of action items) for display.

An overview of this agent is shown in Figure 3, and the XML for the agent itself is shown in Figure 4.

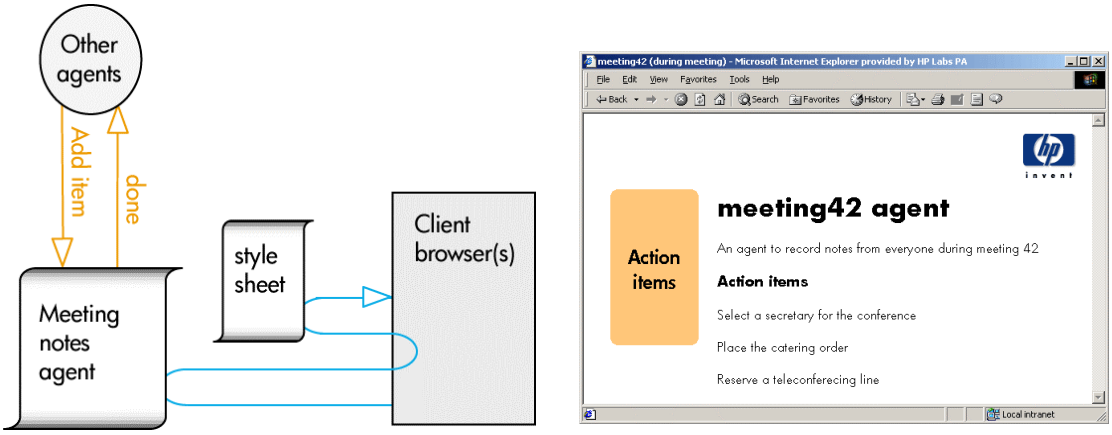

Figure 3. The meeting notes agent adds items in response to incoming ACL messages from other agents. Clients may view the agent directly by loading its XML file into a browser, and performing client-side application of a style sheet. They may view newly-added items by hitting reload.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="duringMeeting.xsl"?>

<program name="meeting42">
<annotation>An agent to record notes from everyone during
meeting 42</annotation>

<data ID="history">
</data>

<behaviour name="main">
   <var ID="count">0</var>
   <code>
     var message = FIPAReceive();
     var messageContent = FIPAParseContent(message);
     var history = XPathAPI.selectSingleNode("program/data");
     history.appendChild(messageContent);
     var reply = message.createReply();
     reply.setPerformative(7);
     reply.setContent( "done" );
     FIPASend(reply);
     count++;
   </code>
</behaviour>

</program>
```

Figure 4. Example of an XML document agent – in this case, one for collecting action items during a meeting. In this document, the `data` node is used to store an action-item history. When a new message arrives, the agent parses the content (assuming it contains XML), and then appends the message content to its data node, before constructing and sending a reply. The final step in the behavior is to increment a count variable (which is visible outside the document as the tag `var` with ID `count`).
The language used is ECMAscript (implemented by calling the FESI implementation [18]), with custom extensions to send/receive ACL messages (implemented by calling Jade's agent message-handling functions) and to parse content and find nodes in the XML DOM (implemented by calling Apache's Xerces and Xalan functions).
In addition to manually storing persistent data directly in XML, we have added support for persistent Javascript variables – these are instantiated from the persistent XML document whenever the script is executed, and the persistent store is updated automatically after every execution step. (In this implementation, one execution step is the processing of a single behavior). Note that we have simplified this example by hiding code for message validation and error handling.

The document agents representing each attendee operate in a similar manner to the one described above. Each maintains a list of action items, but this time, they are for a particular attendee at this particular meeting.

When the attendee adds a new action item (using the web view of the agent), the resulting HTTP post is translated into an ACL message and sent to the agent. Upon receipt, it adds the new action item to its internal list, sends a message to the meeting agent (asking it to add the item to the global list) and then replies to the original message (indicating that processing is complete). That reply causes the client to reload the page, which again causes client-side application of the style sheet, and the updated list becomes visible.

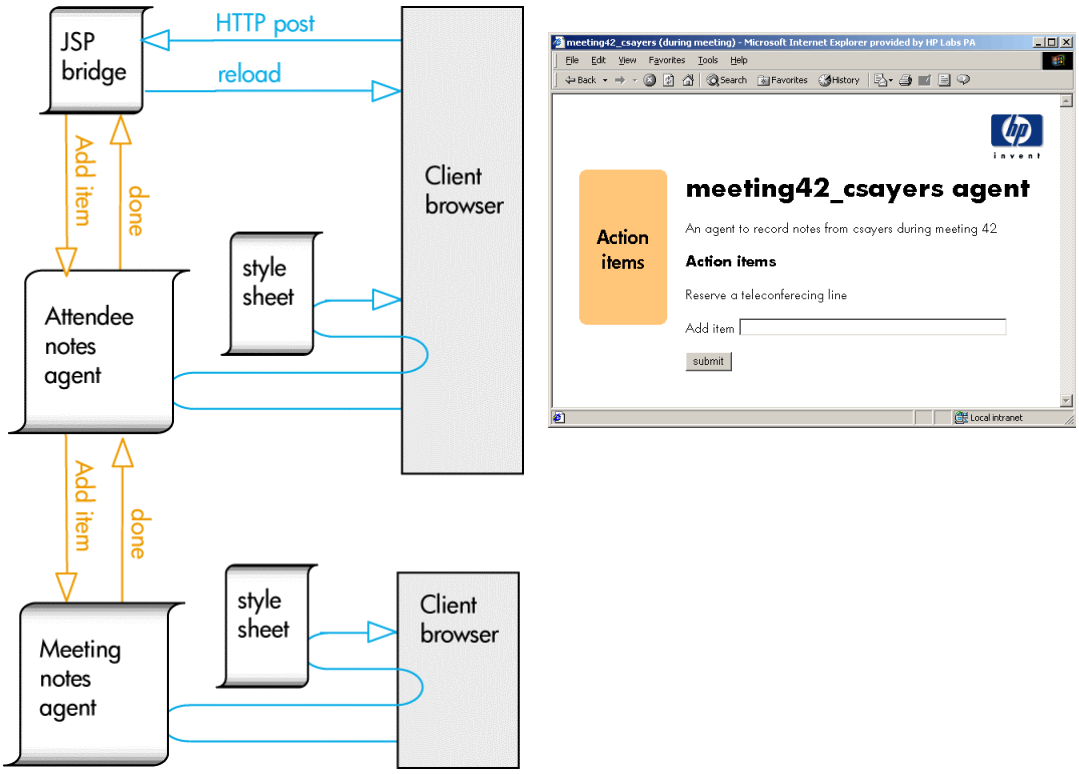

Figure 5. The attendee notes agent adds items in response to incoming ACL messages (initiated, in this case, by an HTTP post from the client browser). It forwards a copy of the new item to the meeting notes agent, before updating itself, and indicating completion. At that point the client browser reloads the page to see the updated agent with the newly-added item.

6 Discussion and Conclusions

The document agents described here have a natural persistence and web presence. They are also lightweight, consuming only disk space while asleep. It is thus possible to support millions of agents on a single host system.

These document agents encourage a different programming model than is usual for web-based applications. Rather than thinking of web pages as just the output of the software system, we instead treat the web pages as the system itself. Each of our documents is an agent, and each agent is a web-accessible document.

For agents that do a significant amount of computation, or which are accessed by many users, it is appropriate to use a compiler – expending the cost of interpretation only once, and gaining additional guarantees of correctness and reliability up-front. Existing java-based agent platforms, such as Jade [17], serve this well.

Document agents are best used when there are a large number of agents, where it is desirable for the agents to have a web presence, or where the number of accesses to any one agent is relatively low.

In many cases, a hybrid approach will work best, using conventional software agents for the heavy processing, and using document agents to provide personalized web-based interfaces.

While it would be possible to provide similar web interfaces using only JSPs, the wins to using document agents are the incorporation of the messaging infrastructure (making it trivial to communicate with other agent-based systems and with other document agents), the desirability of having the code, data and state encapsulated within a single XML document (and visible using standard XML tools) and the convenience and agility of using an interpreted language.

7 Acknowledgements

Many thanks to David Bell at HP Labs for his careful review of this document.

The described testbed system was simplified due to the availability of excellent libraries. The authors would particularly like to thank all those developers who contributed to the Jade agent platform, the FESI ECMAScript interpreter, and Apache's Xerces and Xalan libraries.

8 References

1 Extensible Markup Language (XML), http://www.w3.org/XML/

2 Resource Description Framework (RDF), http://www.w3.org/RDF/

3 Tim Berners-Lee, James Hendler and Ora Lassila, The Semantic Web, Scientific American, May 2001.

4 Dejan Milojicic et al, Mobile agent applications, IEEE Concurency, pages 80—90 July-September, 1999.

5 Danny B. Lange and Mitsuro Oshima, Mobile Agents with Java: The Aglet API, in Programming and Deploying Mobile Agents with Java, Addison Wesley, 1998.

6 David Halls, John Bates and Jean Bacon, Flexible Distributed Programming using Mobile Code. In Proceedings of the Seventh ACM SIGOPS European Workshop, pages 225—231, September, 1996.

7 Krishna Bharat and Luca Cardelli, Migratory Applications, Proceedings of the ACM Symposium on User Interface Software and Technology 1995 (Pittsburgh, PA, Nov 1995). 1995.

8 John Bates, David Halls and Jean Bacon. Middleware support for mobile multimedia applications, ICL Systems Journal, 12(2):289—314, November 1997.

9 ECMAScript Language Specification, Standard ECMA-262, 3rd Edition, December 1999.

10 M. Breu et al. ESPRIT Project 25-338, Citrix Systems, Cambridge

11 Lange et al. A New Internet Agent Scripting Language Using XML, AAAI-99 Workshop on AI in Electronic Commerce, Orlando, Florida, July 1999.

12 Foundation for Intelligent Physical Agents (FIPA) Agent Communication Language (ACL) Message Structure Specification, http://www.fipa.org/specs/fipa00061/

13 The Extensible Stylesheet Language (XSLT), http://www.w3.org/Style/XSL/

14 XML Path Language (Xpath) Version 1.0, W3C Recommendation 16 Nov 1999, http://www.w3.org/TR/xpath

15 XML Pointer Language (Xpointer) Version 1.0, W3C Candidate Recommendation, 11 Sept 2001, http://www.w3.org/TR/xptr/

16 JavaServer Pages: Dynamically Generated Web Content, http://java.sun.com/products/jsp/

17 The Java Agent Development framework (JADE), http://www.sharon.cselt.it/projects/jade/

18 FESI a Free EcmaScript Interpreter, http://home.worldcom.ch/jmlugrin/fesi/