Increasing agent mobility in Siebog Agent Middleware

Milan Vidaković*, Mirjana Ivanović**, Jovana Vidaković**
* Faculty of Technical Sciences, Novi Sad, Serbia
** Faculty of Sciences, Novi Sad, Serbia
minja@uns.ac.rs, {mira, jovana}@dmi.pmf.uns.ac.rs

***Abstract*— Agent-based systems provide users with multitude of software entities – agents, which can perform various tasks, while demonstrating autonomy, reactivity social ability and pro-activity. One of the key features of the agent technology is the agent mobility. This feature provides agents with the capability of moving from one computer to another, from one platform to another. Agent mobility is a basic part of the Siebog Agent Middleware since its beginnings. At first, agents in Siebog have been written in Java, and were able to migrate only to Java-based Siebog middleware. With the advent of JavaScript support in browsers, it was possible to have multitude of software agents running in browsers. Siebog added this feature with the introduction of the Radigost – a JavaScript support for both client-side (browsers) and server-side. However, the Radigost did not provide a full agent mobility. For example, it was not possible to migrate the server-written agent to the browser. This paper describes an implementation of a full agent mobility. This enhanced mobility now enables agents to freely move from browser to server, from server to another server, from server to browser, and from browser to browser.**

I. INTRODUCTION

Agent mobility is one of the key characteristics of agent frameworks. It allows agents to migrate from one computer to another. First generations of agents were capable of migrating from one server to another only. In recent years, agents are capable of migrating from servers to browsers (also know as client side mobility), and vice-versa.

This paper deals with the implementation of all possible types of mobility of agents written for the Siebog agent middleware. It will cover server-to-server, server-to-browser, browser-to-server, and even browser-to-browser types of mobility.

A. Siebog agent Middleware

Siebog Agent Middleware [1] is written in Java, and it supports primarily Java-written software agents. However, it can also execute JavaScript-written agents on the server-side and provide support for the JavaScript-written agents to run in browsers. So far, JavaScript-written agents in the Siebog middleware have been able to run at server-side and in browsers [2], but haven't been able to migrate easily between server-side and browsers in both directions, and haven't been able to move from server-side to another server-side, nor to move from one browser to another. The main motivation in this paper is to improve mobility, by allowing agents to migrate freely from server-side to browsers (client-side) and vice versa, but also to support server-side to server-side and browser to browser mobility.

Since Siebog Agent Middleware has been developed for the past 15 years using JavaEE technology, its agents were initially implemented using Java only. Since the introduction of the Radigost [2] – a Siebog subsystem which supports JavaScript-written agents, it was possible to develop agents in JavaScript. Those agents were able to run in browsers, or on the server-side, but haven't been able to migrate easily from server to client, in both directions.

Browser-based agents were able to migrate to the server-side, but it was not possible to migrate in the other direction. Also, server-side JavaScript agents were not able to migrate to another Siebog middleware. The main goal in this research was to make Java-Script based agent mobility fully implemented, in all cases and directions, which includes:

- server-side to server-side,
- browser to browser,
- server-side to browser, and
- browser to server-side.

The browser to browser mobility was implemented using the server-side. In that case, the agent would first migrate from the browser to the server-side, and then from the server-side to another browser..

II. RELATED WORK

Server-side mobility exists from the very beginning of the agent technology. In Java-based agent frameworks [1, 3], it is based on the Java serialization and is restricted to the Java programming language only.

With the introduction of Agent Languages [4, 5, 6], it was possible to write agents in agent languages, and then to transform that domain-specific language code to the destination platform. Siebog agent middleware has introduced the ALAS agent language [6, 7], which is based on the transformation of the ALAS domain-specific agent language to Java. This concept provides a mobility that heavily depends on the code transformation routines, and with the development of the different code transformers, other destination languages will be included.

While the server-side mobility exists for a long time, browser-based mobility has appeared just recently. The advent of the JavaScript programming language has provided a possibility of having agents operate in browsers, while Node.js server-side JavaScript engine has made JavaScript-based agents to operate on servers.

Node.js also made browser-to-server and server-to-browser mobility possible. Papers [8, 9] describe one solution for the implementation of mobile agents written in JavaScript which are based on the Node.js server runtime. This solution provides agent mobility, but does not employ advanced JavaScript features like WebWorkers. Siebog mobility employs a wide variety of advanced JavaScript features, like WebWorkers and WebSockets.

III. SIEBOG MOBILITY

Mobility in agent technology is achieved using serialization. In Java programming language, this is a built-in feature, which can be easily achieved programmatically. In JavaScript it is supported due to the possibility to programmatically examine all properties of the agent (JavaScript object) in the runtime and export them into the JSON format. In both languages, mobility depends on the existence of the agent source code on both ends of the mobility path.

For JavaScript-written agents, the mobility is achieved by enumerating all the properties, sending them to the destination, and then reconstructing the original object. However, it is not sufficient to just enable mobility. It is also important to maintain message exchange even if the agent moved to the other location (other server, or browser). This is possible because Siebog maintains message exchange between all registered nodes in the agent framework, regardless of the type – server-side or client-side (browser).

Since the previous version of Siebog middleware did not support all four types of mobility, it had to be extended to fully support JavaScript-written agent mobility. To do so, it was necessary to extend both XJAF and Radigost subsystems, which are main parts of the Siebog middleware. XJAF subsystem is used for the server-side Java agents, while Radigost is a subsystem which is used to support both client-side (browser-based) and server-side JavaScript agents. The key element was the implementation of the **`moveToX()`** function in both subsystems.

A. Moving client-based agent to the server-side

For client-based agents, mobility to the server-side is achieved by executing the following code:

```
this.moveToServer(destinationServer);
```

This code initiates the following process:

- serialize the agent state into the JSON string,
- send the message to the destination server, containing agent name and agent state via REST call,
- reconstruct the JavaScript agent on the destination server using the JSON data, and
- invoke the **`onArrived()`** callback on the arrived agent.

Serialization of an agent state is done by invoking the **`getState()`** method:

```
Agent.prototype.getState = function() {
    var state = {};
    for (var prop in this)
      if (typeof this[prop] !==
        "function")
        state[prop] = this[prop];
    return state;
  };
```

Listing 1. Serialization of the agent state

The method above iterates through all properties of an agent, gathers all non-function ones and stores them in the **`state`** variable. After that, the agent state is ready to be sent to the server-side.

Sending the client-based agent state via REST is done in the **`acceptAgent()`** method:

```
xjaf.acceptAgent = function(agent, state)
{
  state["url"] = agent.url;
  $.ajax({
    url: "/siebog-
war/rest/managers/acceptRadigostAgent",
    type:"PUT",
    data: JSON.stringify(state),
    contentType:"application/json",
    dataType:"json",
    complete: function(data) {
      // Agent sent
      agent.myAid.radigost = false;
    }
  });
}
```

Listing 2. Sending the client-based agent state via REST

The **`acceptAgent()`** method sends an AJAX request to the Siebog server with the agent state as a payload. The Siebog is then ready to reconstruct the agent using the received agent state.

Agent reconstruction on the server-side is done in the **`acceptRadigostAgent()`** method:

```
public String acceptRadigostAgent(String
agentState, ServletContext ctx) {
  Map<String, Object> map =
JSON.g.fromJson(agentState, Map.class);
  ...
  AID aid = new AID(
    myAid.get("name").toString(),
    NodeManager.getNodeName(),
    RadigostStub.AGENT_CLASS);
  aid.radigost = true;
  agm.startServerAgent(aid,args,true);
  return aid.getStr();
}
```

Listing 3. Agent reconstruction on the server-side

The agent state is reconstructed, and then the server-side JavaScript agent is started and its internal state is set to the received state.

When the agent is reconstructed and registered on the destination server, the **`onArrived()`** callback method is invoked:

```
invocable.invokeMethod(jsAgent,
"onArrived", NodeManager.getNodeName(),
true);
```

Listing 4. Invoking the callback upon arrival

The **onArrived()** callback is used to notify the agent that it has arrived on the destination. In this method, the programmer puts the code which should be activated when the agent reaches the destination.

B. Moving server-based agent to the client-side

To move the server-based agent to the client-side (browser), it is necessary to execute the following code:

```
this.moveToClient();
```

This code initiates the following procedure:

- send a message to the Siebog middleware to serialize agent state,
- serialize the agent state into the JSON string,
- send the agent state to the browser page, where the **radigost.js** library has been loaded (this library installs the **Radigost** part of the Siebog middleware on the web page), via WebSockets technology,
- restore the agent state using the JSON data, and
- invoke the **onArrived()** callback on the arrived agent.

Listing of the **moveToClient()** function follows:

```
Agent.prototype.moveToClient = function()
{
  var agState = this.getState();
  var msg = {
    opcode : opCode.MOVE_TO_CLIENT,
    myAid : JSON.stringify(this.myAid),
    state : JSON.stringify(agState)
  };
  this.post(msg);
};
```

Listing 5. Sending the system-level message to migrate

Agent migration starts with the message to the Siebog middleware to start the serialization. Siebog starts the serialization using the same **getState()** function, which is used by the client-based agent, and which was shown in the Listing 1.

After the agent state was serialized, the agent is sent to the client-side using the WebSockets connection, which is established automatically, when a client (browser) loads the Siebog web page:

```
socket: new WebSocket("ws://" +
window.location.host + "/siebog-
war/webclient")
```

Listing 6. Establishing WebSockets connection

The WebSockets connection is used for sending messages between server-side and client-side. Agent state is also sent to the client-side using this connection:

```
String aidStr = (String)
_arg.get("myAid");
String stateStr = (String)
_arg.get("state");
ACLMessage mess = new ACLMessage();
mess.opcode = opcode;
AID receiver = JSON.g.fromJson(aidStr,
AID.class);
mess.userArgs.put("state", stateStr);
mess.receivers.add(receiver);
mess.performative =
Performative.PROPAGATE;
webSocketEvent.sendMessageToClient(mess);
agm().stopAgent(myAid);
```

Listing 7. Sending the agent state via WebSockets connection

The code above illustrates sending agent state from the server-side to the client-side using WebSockets. When an agent is received on the client-side, it is ready to be reconstructed.

Agent reconstruction on the client-side is done in the **onmessage()** method of the WebSockets connection:

```
radigost.socket.onmessage =
function(message) {
  var msg = JSON.parse(message.data);
  if (msg.opcode ===
opCode.MOVE_TO_CLIENT) {
    var aid = msg.receivers[0];
    var state = msg.userArgs["state"];
    state = typeof state === "string" ?
JSON.parse(state) : state;
    var ag = radigost.getAgent(aid);
    if (!ag) {
      // agent is not running.
      // Let's try to start one.
      radigost.start(new
AgentClass(state.url, null, null),
aid.name, false);
      ag = radigost.getAgent(aid);
    }
    ag.myAid.radigost = true;
    ag.myAid.agClass = undefined;
    // we will send the Worker
    // message to the agent's worker,
    // so it will invoke the onArrived
    ag.worker.postMessage(msg);
    removeServerAgent(ag.myAid);
    addClientAgent(ag.myAid);
    return;
  }
  ...
```

Listing 8. Reconstructing the agent on the client-side

The code above receives agent state, and then checks if the agent already exists. If not, a new agent is created on the client-side, and its state is set to the received one.

When the agent is reconstructed, the Radigost will send the message it using agent's WebWorker, so it would finally invoke the **onArrived()** method:

```
self.onmessage = function(ev) {
  var msg = ev.data;
  if (msg.opcode ===
    opCode.MOVE_TO_CLIENT) {
```

```
    // Agent just arrived from the
    // server
    // (radigost.socket.onmessage).
    // Set the state and call
    // the onArrived callback.
  self.agentInstance.setState(
    msg.userArgs.state);
  self.agentInstance.onArrived(
    self.agentInstance.myAid.host,
    false);
}
...
```

Listing 9. Sending the message to the newly-arrived agent

When the agent's WebWorker receives the message, it will call the **`onArrived()`** callback, which will signalize the agent that it has arrived on the destination.

C. Moving server-based agent to another server-side

To move the server-based to another server-side, it is necessary to execute the following code:

```
this.moveToServer(destinationServer);
```

This code initiates the following procedure:

- serialize the agent state into the JSON string,
- send the agent state to another server-side, via REST call,
- restore the agent state using the JSON data, and
- invoke the **`onArrived()`** callback on the arrived agent.

Agent serialization is done the same way it is done with the client-based agent – by calling the **`getState()`** method (as shown in the Listing 1).

Sending the server-based agent state via REST is done in the **`moveToServer()`** method:

```
Function<String, Void> moveToServer =
(arg) -> {
  String newHost = arg;
  try {
    String jsState =
JSON.g.toJson(invocable.invokeMethod(jsAg
ent, "getState"));
    ResteasyClient client = new
ResteasyClientBuilder().build();
    ResteasyWebTarget rtarget =
client.target("http://" + newHost +
"/siebog-war/rest/managers");
    SiebogRest rest =
rtarget.proxy(SiebogRest.class);
    rest.acceptRadigostAgent(jsState,
null);
    agm().stopAgent(myAid);
  } catch (Exception e) {
    e.printStackTrace();
  }
  return null;
  };
```

Listing 10. Sending the server-based agent state via REST

The code above sends the HTTP request containing agent state. The receiver of that request is the **`acceptRadigostAgent()`** method, shown in the Listing 3.

On the recipient side, the agent state is restored, and the **`onArrived()`** method is invoked on the newly-reconstructed agent, to indicate that the agent has arrived to the destination (shown in the Listing 3).

D. Moving client-based agents to another client

Mobility between two browsers is done in two steps:

- agent moves from browser to server,
- agent moves from one server to another, and
- agent moves from server to the other browser.

All the steps above have been described in previous sections.

Whenever an agent arrives at the destination, it is necessary for it to be able to determine where it has come. For that purpose, the property **`this.isServer`** has been introduced. This property has a value of **true** if the agent has arrived on the server, and **false** if it has arrived in the browser. Wherever the agent has arrived, it is also capable of sending and receiving messages due to the built-in support for the message exchange, which supports both types of agents – server-side and client-side.

Client-side agents can even create a stub agent at the server-side from which they arrived, so the messages addressed to them can arrive from the server-side. For example, one client-based agent can send a message to another client-based agent (in the same browser), or one server-based agent can send the message to a client-based agent via server-side stub agent. This is possible due to the implementation of the **MessageManager** subsystem in the Siebog middleware. This implementation provides message exchange using JMS (Java Messaging System) for server-based agents, and WebSockets with Web Workers for the client-based agents. Both solutions [10] provide high volume of message exchange, since the JMS can operate in clustered environment, having multiple JavaEE application servers distribute the load and provide safe fail-over. On the other hand, Web Sockets and Web workers provide the possibility of having multiple client-based agents installed on multiple browsers, all able to send and receive messages.

IV. CONCLUSION

This paper describes an improvement in agent mobility for the Siebog agent middleware. The improvement enables all four types of mobility:

- Server-to-server,
- Server-to-browser,
- Browser-to-server, and
- Browser-to-browser.

All four types of mobility are implemented for the JavaScript-based agents. Those agents can operate in both browsers and Java Virtual Machine, thus enabling full mobility between servers and clients.

Future work will include incorporation of the ALAS-written agents in the Siebog, which will provide the ultimate mobility, since the ALAS is an agent-oriented programming language, which can be transformed into the destination language of choice. Currently, ALAS supports

Java language only, but is intended to be transformed into the JavaScript as well.

ACKNOWLEDGMENT

Results presented in this paper are part of the research conducted within the Grant No. OI-174023, Ministry of Education, Science and Technological Development of the Republic of Serbia.

REFERENCES

[1] D. Mitrović, M. Ivanović, M. Vidaković, Z. Budimac, "The Siebog Multiagent Middleware", In Knowledge-Based Systems, vol 103, no C, july 2016, pp. 56 – 59, DOI: 10.1016/j.knosys.2016.03.017

[2] D. Mitrovic, M. Ivanovic, Z. Budimac, M. Vidakovic, "Radigost: interoperable web-based multi-agent platform", In Journal of systems and software, pp. 167-178, vol. 90, No. 4, DOI 10.1016/j.jss.2013.12.029, 2014, ISSN: 0164-1212

[3] Bellifemine, F., Caire, G., Greenwood, D., *Developing Multi-Agent Systems with JADE*, John Wiley & Sons, 2007

[4] Demirkol, S., M. Challenger, S. Getir, T. Kosar, G. Kardas, and M. Mernik. 2012. "SEA_L: A Domain-Specific Language for Semantic Web Enabled Multi-Agent Systems", In Federated Conference on Computer Science and Information Systems (FedCSIS), 2012, pp. 1373-1380, Wroclaw, Poland

[5] Demirkol, S., M. Challenger, S. Getir, T. Kosar, G. Kardas, and M. Mernik. 2013. "A DSL for the Development of Software Agents Working within A Semantic Web Environment", In Computer Science and Information Systems 10 (4): 1525-1556, doi:10.2298/CSIS121105044D.

[6] Mitrović, D., Ivanović, M., Vidaković, M., "Introducing ALAS: A Novel Agent-Oriented Programming Language", In Symposium on Computer Languages, Implementations, and Tools (SCLIT 2011), September 19-25, Halkidiki, Greece.

[7] Sredojević, D., Vidaković, M., Ivanović, M., "ALAS: agent-oriented domain-specific language for the development of intelligent distributed nonaxiomatic reasoning agents", Enterprise Information Systems, 2018, pp. 1-25, ISSN: 1751-7575, DOI: 10.1080/17517575.2018.1482567

[8] Jarvenpaa, L., Lintinen, M., Mattila, A.L., Mikkonen, T., Systa, K., Voutilainen, J.P., "Mobile agents for the internet of things", 17th International Conference on System Theory, Control and Computing (ICSTCC), pp. 763–767, October 2013.

[9] Systa, K., Mikkonen, T., Jarvenpaa, L., "Html5 agents: Mobile agents for the web", Krempels, K.H., Stocker, A. (eds.) Web Information Systems and Technologies, Lecture Notes in Business Information Processing, vol. 189, pp. 53–67, 2014.

[10] Dejan Mitrović, Mirjana Ivanović, Milan Vidaković, Zoran Budimac, "A scalable distributed architecture for web-based software agents", Proceedings of the 7'th International Conference on Computational Collective Intelligence (ICCCI), September 21-23, 2015, Madrid, Spain, In M. Nunes, N. T. Nguyen, D. Camacho, B. Trawinski, editors, volume 9329 of Lecture Notes in Artificial Intelligence, pp. 67-76, Springer International Publishing, 2015, DOI: 10.1007/978-3-319-24069-5_7, Print ISBN: 978-3-319-24068-8, Online ISBN: 978-3-319-24069-5