ISSN 1392–124X (print), ISSN 2335–884X (online) INFORMATION TECHNOLOGY AND CONTROL, 2016, T. 45, Nr. 2

Siebog: an Enterprise-Scale Multiagent Middleware

Dejan Mitrović[1], Mirjana Ivanović[1], Milan Vidaković[2], Zoran Budimac[1], Jovana Vidaković[1]

[1]*Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia*
e-mail: dejan@dmi.uns.ac.rs, mira@dmi.uns.ac.rs, zjb@dmi.uns.ac.rs, jovana@dmi.uns.ac.rs

[2]*Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia*
e-mail: minja@uns.ac.rs

crossref http://dx.doi.org/10.5755/j01.itc.45.2.12621

Abstract. This paper presents a new multiagent middleware named Siebog, made to provide high performance and platform independence for software agents. This framework was built by combining the features of the Radigost and XJAF agent frameworks. It provides an infrastructural support for both client-side and server-side agents. The client-side agents are written in JavaScript and can be executed on a wide variety of software and hardware platforms, including desktops, smartphones and tablets, and Smart TVs. The server-side agents, on the other hand, can harness the benefits of clustered environments, and rely on automated load-balancing and fault-tolerance for the uninterrupted delivery of services. The two sides of Siebog have been integrated in a way that enables cross-platform messaging, agent code sharing, and even heterogeneous agent mobility.

Keywords: clustered computing; HTML5; Java EE; multiagent middleware; software agents.

1. Introduction

For modern agent middlewares it is not enough just to support software agent lifecycle. Platform independence and high performance of agents utilized through distributed computing has become mandatory feature for agent middlewares. In this paper, we present a novel multiagent middleware that offers those new features by using the state-of-the-art technologies. The name of the presented multiagent middleware is Siebog and it has been developed from two previously developed frameworks: Radigost and XJAF.

Radigost framework offers JavaScript-based agents which can execute on large number of platforms, including smartphones, smart TVs, desktops, tablets, etc. On the other hand, XJAF can be used as a clustered environment for the server-based agents. In papers [1], [2], we have shown that Siebog achieves better performances and uses more advanced technologies than other existing agent middlewares.

1.1. Origins of the Siebog middleware

During the last decade, *the web* has become an important software platform. It has gradually evolved into an environment capable of providing functionalities previously available in desktop applications only.

One of the driving forces behind the proliferation of web applications has been the HTML5 standard, and its support on a variety of hardware and software platforms, including mobile devices [3]. Among the many HTML5-related specifications, two are of the special importance for the work presented in this paper: *Web Workers* and the *WebSocket* protocol. Web Workers provide true multi-threading in JavaScript, along with a ready-made messaging infrastructure for asynchronous communication. The WebSocket protocol, on the other hand, provides full-duplex communication over a single channel. It enables web application servers to *push* information to remote clients, offering an alternative to usual *pull* approach.

In line with these advances, we have developed *Radigost*, a purely web-based multiagent middleware designed to harness the numerous benefits of HTML5 [4], [5]. The platform is divided into two major parts: a client and a server. The client side and the agents themselves are implemented in JavaScript and executed inside of a web browser. The server side provides support for more complex features, such as the agent state persistence, discussed later.

Improvements brought by HTML5 on the client have been supported by corresponding server-side technologies. The *Enterprise Edition* of Java (Java EE) represents one of the most widely used technologies for server-side software development. It offers a wide range of technical solutions suitable for developing scalable and reliable software system. As such, it represents an excellent platform for developing multiagent systems.

Extensible Java EE-based Agent Framework (XJAF) is our additional multiagent middleware built using technical solutions of Java EE [6], [7]. One of the main goals of XJAF is to demonstrate how *standard* Java EE technologies can be used to implement many functional requirements imposed on multiagent middleware. For example, the *Java Message Service* makes it very easy to implement asynchronous inter-agent communication, without "reinventing the wheel." Recent developments have been focused on executing XJAF on top of computer clusters, providing high-availability of deployed multiagent applications [1].

Obviously, both Radigost and XJAF serve similar purposes. Both are designed to support modern, web-based enterprise applications, with Radigost being dedicated to the client-side, and XJAF dedicated to the server-side agent execution. Therefore, an integration of the two systems represents the next natural step.

This paper presents *Siebog*, an enterprise-scale multiagent middleware built by combining Radigost and XJAF into a unified multiagent framework. This approach offers a number of advantages over existing multiagent solutions.

First of all, on the client side, our system is platform-independent. As shown in [4], [5], its agents can be executed, without any modifications, on desktop computers, smartphone and tablet devices, and Smart TVs. In addition, no prior installation or configuration steps are required. This is beneficial to both agent developers, because of the *write once, run anywhere* approach, and to end-users, since they can utilize the benefits of the agent technology in the most convenient manner. Finally, its client-side runtime performance is comparable to that of a classical, desktop multiagent platform [5].

On the server side, the main features of Siebog include scalability and fault-tolerance. Siebog automatically distributes agents among available cluster nodes, supporting large agent societies [1]. In addition, the internal state of each agent is replicated across the cluster. In case its host becomes unavailable, the agent can continue its operations on one of the remaining nodes.

As shown in Section 5, this range of functionalities is not available in any existing multiagent solution. During the development of both Radigost and XJAF, and now Siebog, a strong emphasis has been put on standards compliance. This increases the interoperability of Siebog and non-agent-based enterprise solutions. The learning curve is also flatter for developers familiar with web and enterprise software technologies.

Having in mind above-mentioned advantages of our Siebog system, it offers great opportunities and possibilities to be used in wide range of contemporary applications. Some of them are briefly mentioned below. For example, it can support *virtual assistants* [8] which, thanks to the use of modern technologies, can easily be integrated with popular cloud-based services (i.e., online calendars and schedulers). Similarly, due to the cross-platform nature of its client side, Siebog can support the development of *smart environments* (e.g., buildings, industrial installations, etc.) [8]. Our system can also be used to develop adaptive, personalized web-based e-learning environments [10]. Finally, due to its highly distributed nature and obtaining high-performances, Siebog is especially useful in applications that need to launch large numbers of agents (e.g., swarm intelligence [11]).

Siebog is released as free software, under the generous Apache License version 2.0, and placed on the GitHub [12]. It is actively used for scientific purposes [1], [2], [4], [5], [6], [7] as well as for educational purposes [13].

The rest of the paper is organized as follows. Section 2 presents a brief overview of XJAF and Radigost and their functionalities. The details about Siebog, its architecture and internal organization, are given in Section 3. Section 4 presents a case study that demonstrates the main benefits of Siebog. Related work is discussed in Section 5, while the final conclusions and future research directions are given in Section 6.

2. Overviews of XJAF and Radigost

XJAF and Radigost represent the building blocks of the new Siebog multiagent middleware. Therefore, to fully understand the characteristics and functionalities of Siebog, some basic insight into the two sub-systems is required. This section highlights the most important features of XJAF and Radigost. For more information, see [4], [5], [7].

2.1. XJAF middleware

As noted earlier, Java EE includes a wide range of technologies that can be used to realize many functional requirements of a multiagent middleware. Example technologies include [14]: *Enterprise JavaBeans* (EJBs), server-side components that implement the applications' business logic, *Java Message Service* (JMS), an API for asynchronous messaging between loosely coupled components, and *Java Naming and Directory Interface*, a directory service.

XJAF is a multiagent middleware that builds on top of Java EE [6], [7]. The focus is on using existing *standardized* and well-tested technologies for enterprise application development. In this way, XJAF and its agents can be integrated into existing enterprise applications with the minimum amount of effort.

In addition to aforementioned technologies, Java EE offers an extensive support for clustered computing and the deployment of scalable and fault-tolerant applications. Obviously, XJAF successfully exploits this fact. It uses a computer cluster in order to achieve two important functionalities: agent load-balancing and state replication and failover. Load-balancing is concerned with distributing running agents across cluster nodes, and sharing the computational load. The state replication process copies the agent's internal state to a pre-defined number of backup nodes. In case the agent's host node becomes unavailable, the failover process will transparently restore the agent on one of the remaining nodes.

XJAF consists of a number of components called *managers*. In the latest incarnation of the system [1], three managers are available: *Connection Manager, Agent Manager*, and *Message Manager.* The Connection Manager is in charge of connecting XJAF clusters into a single computational network.

The Agent Manager controls the agents' life-cycles and maintains the directory of deployed and running agents. Internally, it maps agents to EJB components, and then passes them on to the enterprise application server. The server, in turn, provides a number of features, including concurrent access control, transaction management, and state replication and failover described earlier.

Finally, the Message Manager provides the inter-agent messaging infrastructure. Again, internally it relies on the JMS for asynchronous and reliable message delivery and processing.

Performance evaluation results presented in [1] outline one possible use case for XJAF. That is, our system is well-suited for applications that need to deploy and run large populations of agents in computer clusters. However, being built on Java EE standards, XJAF might help to bridge the gap between existing enterprise solutions and the agent technology.

2.2. Radigost platform

As a multiagent platform, Radigost provides an architectural support for the execution and interaction of its agents. Its core functionalities include agent life-cycle management, a communication infrastructure, and a yellow-pages service. For example, agents can find other (active or inactive) agents, spawn new agent instances, and interact with agents running not only within the system boundaries, but also in third-party multiagent solutions.

Being a JavaScript application, the life-cycle of an agent is inherently tied to its host web page. However, many targeted web applications, such as online e-learning and tutoring systems, require long-running, ongoing interaction with end-users. One of the key features of Radigost is, therefore, the support of agent state persistence. The state is restored during the agent initialization (i.e., once the user loads the host web page), and also stored on a remote persistence server during the finalization (i.e., when the user closes the web page). From the agent's point of view, the entire process is performed transparently. The runtime state is an arbitrary data structure, and depends on the actual needs of the agent.

One of the key features of agents is a social interaction. In Radigost, this interaction is achieved through asynchronous message exchange. For maximum performance and interoperability, the platform utilizes the existing, standardized messaging infrastructure of Web Workers.

The general architecture of Radigost is shown in Figure 1. The platform includes a client and a server side. The client side is executed inside of a web browser, and is comprised of agents and a client library. The client library exposes most of the platform's functionalities to agents and agent developers. For example, it provides the *Agent prototype* which defines the default functionalities for all agents. It also includes the necessary functions that support the inter-agent communication through message exchange. Finally, in addition to inter-agent messaging, the library provides a support for communication between an agent and the client application (e.g., the web page), realized through the well-known *Observer* software design pattern.

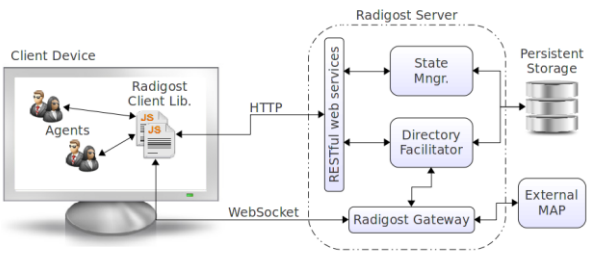

Figure 1. General architecture of Radigost [5]

The server-side of Radigost includes three core components: *State Manager*, *Directory Facilitator*, and *Gateway*. State Manager enables previously described agent state persistence. Directory Facilitator implements the standard yellow-pages service. It enables agents to register and publish descriptions of their functionalities, and to search for other registered agents.

The final server-side component of Radigost is Gateway. It adds the interoperability to our system, enabling its agents to seamlessly interact with agents deployed in third-party multiagent solutions, such as JADE [15]. The Gateway itself consists of two specialized sub-components: *Bridge* and *Socket*. Bridge performs on-the-fly agent name mappings and transformations of FIPA ACL messages. It needs to be re-implemented for each supported third-party multiagent solution.

The Socket component represents a channel for the flow of messages between Radigost and third-party agents. Socket relies on the WebSocket protocol, enabling full-duplex communication: for example, not only can Radigost agents send messages to JADE agents, but JADE agents can also initiate the interaction with any Radigost agent running in any connected client. This feature is very important, as it increases the practical applicability of our system in a significant way.

3. The Siebog multiagent middleware

Siebog is a multiagent middleware built by integrating XJAF and Radigost into a single framework. In this way, it can harness the benefits of both systems. For example, it can provide a clustered computing on the server, and assure the platform-independence on the client side. All of its internal components are standards-compliant and can easily interact with or be integrated into existing web and enterprise software systems. For example, Siebog agents can publish their functionalities in a form of web services, and can easily invoke other EJB components or perform object-relational mapping.

Besides functionalities that are directly extracted from its individual components, Siebog has several other important features, including the following:

- Cross-platform messaging: Radigost agents can communicate with XJAF agents in the same way as with Radigost agents, and vice-versa.
- Code sharing: an agent written once can be executed both on Radigost (client-side) and XJAF (server-side).
- Heterogeneous agent mobility: an agent can freely move between the Radigost client and the XJAF server. This migration is achieved indirectly, through the internal state transfers.

This section provides more details about the features of Siebog, and describes its architecture and internal organization.

3.1. Software integration patterns

Heterogeneous system integration is a common and well-understood problem. Several integration patterns have emerged over time, including the *Shared Database*, *Message-Oriented Middleware*, and *Remote Procedure Invocation* patterns [16]. Shared Database is applicable when different sub-systems need to share the data, but otherwise operate independently of each other. The database is directly accessible by all components, and usually provides a single schema. The Message-Oriented Middleware pattern offers the greatest degree of component independence [16]. Different parts of the system exchange messages, carrying (usually) small packets of information, in an asynchronous manner.

Remote Procedure Invocation enables heterogeneous sub-systems of the overall application to share their functionalities, rather than data [16]. The public functionality of each sub-system is exposed using an agreed-upon format. During the invocation, all internal communication (i.e., within a sub-system) is automatically transformed into a standardized external protocol. Over time, Remote Procedure Invocation has been realized in a number of concrete forms, including CORBA, Java RMI, and web services, where web services currently represent the most widely-used approach.

In the case of Radigost and XJAF, the integration approach is mainly dictated by their underlying implementation technologies. The most natural and straightforward way of integrating the JavaScript-based Radigost and the Java EE-based XJAF is to use web services. Although in Radigost especially a strong emphasis has been put on interoperability, a tighter integration through this layer is more beneficial in the long run. By replacing the previously described Gateway component with a web services-based layer, the two systems can:

- Cooperate more efficiently, e.g., by eliminating the need for agent name mappings and message transformations;
- Achieve a greater level of interoperability, through e.g., agent mobility; and
- Offer simplified agent development process through code sharing: an agent written once can be executed, without modifications, on any of the two systems.

Interoperability with third-party multiagent solutions is still planned, but it will be redesigned as part of XJAF.

3.2. XJAF as a web-service oriented architecture

The main goal of web services is to provide machine-to-machine communication. In general, a web service consists of an interface understandable by machines (and humans), and a communication protocol. The first step in developing Siebog is, therefore, to provide web service-based interfaces for XJAF managers. This, however, can be achieved in different ways.

XML web services represent one of the two competing approaches for developing and using web services. It encompasses a *wide* range of standards and specifications, covering interface definition, description and discovery, communication, security, etc. Communication is, in most cases, performed using XML-encoded messages transmitted over HTTP, although other approaches are possible as well. Unfortunately, the sheer amount of (sometimes conflicting) standards and specifications related to XML web services has turned out to be their weakest point, preventing them from gaining much traction in the developer community.

Representational state transfer (REST) is a more recent, alternative design approach for applications based on web services [17]. It uses the four HTTP operations – *GET*, *POST*, *PUT*, and *DELETE* – to query and manipulate *resources*. Resources themselves are represented using *Uniform Resource Identifiers* (URIs). REST is a “standard-less” set of architectural design principles and constraints. The *Stateless* constraint, for example, states that all communication between the client and the server is stateless, in the sense that the server should not store any contextual information about the client [17]. Web services that adhere to all of REST principles and constraints are often referred-to as *RESTful*.

It is worth noting that an older version of XJAF has been also provided in a form of a web service-oriented architecture, with its managers designed as XML web services [6]. However, RESTful web services represent a “better fit” for the intended purpose of integrating JavaScript and Java EE systems. They are much easier to use from the JavaScript client, especially when *JavaScript Object Notation* (JSON) format is used to represent objects. In addition, RESTful services provide better performance, due to less runtime overhead [18].

Since EJBs are used to implement major parts of XJAF, including the managers, the process of transforming them to RESTful web services is straightforward. This is one example of how the standards-compliance can bring benefits to software development. The majority of work has consisted on annotating the appropriate parts of code. Custom (de-)serializations for JSON messages had to be provided in some cases (e.g., for objects representing FIPA ACL messages), but the entire process was completed without any technical difficulties.

Table 1 outlines the proposed REST interface of the agent manager. Its base URI is “`/agents`“, and in all cases the input arguments and return values are represented as JSON-formatted strings.

Table 1. Part of the agent manager's REST API. All methods consume and produce objects of type application/json. Parts of URIs enclosed in curly braces represent variables

Method	URI	Description
GET	/classes	Returns the list of available agent classes.
GET	/running	Returns the list of running agents (their AIDs).
PUT	/running/{agClass}/{name}	Runs a new agent of the given class, and with the given runtime name.
DELETE	/running/{aid}	Stops the agent with the given AID.

Although managers have been re-designed as RESTful web services, internal Java components, such as agents, still invoke them as regular EJBs. This is because EJB invocations incur far less overhead than REST interfaces. For example, when both the agent and the manager are located on the same machine (which is the usual case), no serialization of method parameters is required. Luckily, REST interface definitions can be mixed in with regular EJB method implementations.

3.3. Integrating server-side components

Within the Siebog middleware, the focus of Radigost is solely on the client-side agent execution. At the same time, there is some overlap in functionalities between existing server-side Radigost components shown earlier in Figure 1 and XJAF managers. For example, the Directory Facilitator component in Radigost shares functionalities with the XJAF’s Agent Manager.

In the process of integrating the two systems, a new XJAF manager, named *WebClient Manager*, is introduced. The new manager acts as a layer between Radigost on the client, and XJAF on the server side. It provides two main functionalities.

First, the WebClient Manager acts as a WebSocket server endpoint. It delivers all messages from server-side XJAF components (incl. agents) to client-side agents. Secondly, the new manager takes over the role of the State Manager in the original Radigost configuration. That is, its REST API can be used to persist the runtime state of a Radigost agent, and later restore it (e.g. when the user reloads the web page).

Out of the remaining server-side components shown in Figure 1, Directory Facilitator has been fully replaced by the Agent Manager, while the concrete realization of the Gateway component within the Siebog architecture has been left for further development.

The final proposed architecture of Siebog is shown in Figure 2. Its client-side devices use the Agent Manager as the directory service. Also, whenever an agent is created on the client, its *stub* counterpart will be placed on the server. To external entities, a stub appears as a regular XJAF agent, but any messages sent to it will be forwarded to the client agent.

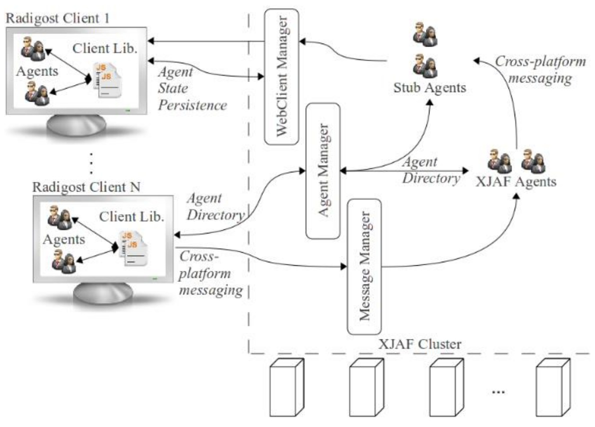

Figure 2. Architecture of the Siebog multiagent middleware

The use of stubs does not introduce more computational overhead than necessary. In the client-server agent communication, messages have to be transferred as JSON strings. Instead of having a centralized repository of client-side agents, which acts as a message (de-)serializer, a more efficient approach is used (e.g., no bottlenecks, no single point of failure, etc.). Since the agent identifier on the client (among other things) incorporates the browser session identifier, it is impossible for two different client-side agents to reference the same server-side stub.

Server-to-client messaging is achieved through the newly introduced WebClient Manager, and over the WebSocket protocol. Client-to-server messaging is performed through the Message Manager's newly developed REST API, as described next.

3.4. Interaction between Radigost and XJAF

Both Radigost and XJAF can act as clients in an interaction. As shown in Figure 3, Radigost includes *stub* implementations of XJAF's managers. Each stub implementation simply performs an asynchronous AJAX call to the appropriate RESTful interface. On the other hand, when XJAF (or one of its agents) needs to interact with Radigost (or one of its agents), the standard WebSocket protocol is used. The message is serialized on the server side into a JSON-formatted string, transferred to the client's web browser, de-serialized into a corresponding Radigost message, and delivered to the target. Unfortunately, the (de-) serialization process cannot be fully avoided at the moment, as web browsers in general do not support binary data transfer through WebSockets.

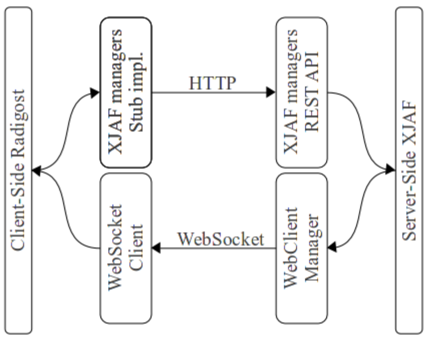

Figure 3. Communication flow from Radigost to XJAF through the REST API, and from XJAF to Radigost over the WebSocket protocol

As noted earlier, the interaction between Radigost and XJAF is manifested in three different ways: code sharing, message exchange, and agent mobility. Obviously, code sharing is possible as long as the agent implementation satisfies all constraints imposed by web environments, and relies only on libraries available in both JavaScript and Java.

Message exchange is the easiest to achieve. It is enough to extend the AID representation in both Radigost and XJAF with a *platform identifier*. The message sending routines in both sub-systems can then simply compare this value with their host platforms' identifiers, and forward the message appropriately.

The idea of code sharing is that the agent developer can write an agent once, using his/hers preferred programming language. The Siebog platform then takes a care of executing the agent on Radigost and XJAF, as needed. This feature has two aspects: executing JavaScript agents in the Java Virtual Machine (VM), and executing Java agents in web browsers.

The execution of JavaScript agents on Java VM is a much simpler task. *Java Specification Request (JSR) 223* defines the standard *Scripting API* for Java VM [19]. Besides executing JavaScript code, the API offers some advanced features. For example, JavaScript programs can import and use Java classes, and indirectly implement Java interfaces, which are then directly accessible in Java programs, etc.

Unfortunately, there is no standard way of executing Java code in web browsers. The approach of embedding XJAF agents and functionalities in Java applets would be *subpar*, beating the purpose and advantages of Radigost. However, an efficient third-party solution does exist. *Google Web Toolkit* (GWT) is a popular set of open-source libraries and tools that transform complex Java-based web applications into pure JavaScript applications [20]. One of its defining features is *cross-browser compatibility*: GWT will produce a number of *compilations* from the same Java source, each optimized for a distinct web browser. In this way, developers are relieved from worrying about browser-specific implementations, and the best possible runtime performance can be achieved. Given its many advantages, Siebog relies on GWT for executing Java agents in Radigost.

Although the code sharing feature of Siebog does work in practice, developers need to be aware of its limitations. For example, writing performance-centric agents in JavaScript and then executing them in Java VM might not be the best option. Instead, it would be better to move the core implementation to a Java-based agent, and then communicate with it from Radigost. Similarly, although powerful, GWT poses a number of limitations on Java implementations; more details are available in the official GWT documentation.

The final aspect of the Radigost-XJAF interaction is agent mobility. For example, an agent running in the web browser should be able to move to the server, replicate and distribute itself across the cluster, and finally return to the web browser carrying the computational result. With the existence of state persistence and code sharing, this functionality can be achieved in a straightforward manner. An example of its use is given in the next section.

4. A case study

The previous performance evaluation of Radigost presented in [5] has shown that the system offers the runtime execution speed comparable to that of a desktop-based multiagent solution. Similarly, it has been shown in [1] that XJAF performs better than a third-party multiagent solution for scenarios with large populations of agents. Here, instead of a performance evaluation, we will demonstrate one practical application of Siebog. The case study utilizes new features that emerge from the proposed Radigost-XJAF integration. More concretely, it shows the heterogeneous agent mobility in practice.

The case study includes a couple of different hardware devices; for example, a smartphone and a Smart TV. The user visits the application's web page on the smartphone, and takes a photo using the device's camera. He/she then activates the mobile agent, which:

- Moves to the server, carrying the photo with it;
- Persists the photo in the user's database;
- Moves to the Smart TV, and shows the photo.

The application's execution flow is shown in Figure **4**. A simplified version of this case study was previously used in [4], that, in turn, was inspired by the case study in [21].

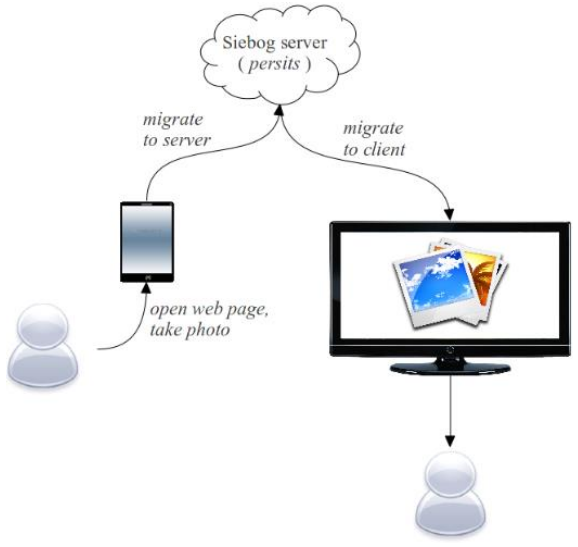

Figure 4. Execution flow of the presented case study

As noted, whenever a web page with Siebog agents is loaded on a client device, the agents and the client device itself are registered with the server. This enables any interested party to inspect and interact with client-side Siebog agents regardless of their physical location. Also, it provides the starting point for agent mobility required here.

The case study consists of the host web page and the mobile agent. The web page shows a list of connected client devices and a number of control buttons. It does not require any external plugins to take photos, since the media capture and streaming are part of the HTML5 standard. Security is provided at the web browser level: the user is asked whether the application can access the camera.

The *full* source code of the mobile agent, named *PhotoAgent*, is shown in Listing 1. Once the user takes a photo, the agent is started. Its initialization function `onInit`, receives the photo along with the destination device, and the user's session identifier. At the end of the initialization phase, the agent moves itself to the server.

Before the migration process starts, Siebog retrieves and remembers the agent's internal state. The actual migration is performed by making the appropriate REST API call to the WebClient Manager.

On the server side, each JavaScript agent is embedded in an instance of the *RadigostAgent* component. That is, RadigostAgent is a server-side (i.e., XJAF) agent that acts as a wrapper around a JavaScript agent. It uses the previously mentioned Java Scripting API to execute and interact with the embedded JavaScript code. For example, Radigost-Agent will first restore the agent's internal state, and then invoke its `onArrived` function.

In order to better support the execution of client-side agents on the server, a special helper class has been developed. In the given source code, the agent uses this helper class to persist the photo in the user's database. The helper's function `persist` relies on the *Java Persistence API*, a Java EE specification for object-relational mapping. As the final execution step on the server, the agent migrates to the destination client to show the photo.

Listing 1. The full source code of the PhotoAgent used in the case study. This mobile agent moves between client devices and the server, carrying the user's photo with it.

```
importScripts("/siebog/radigost.js");
function PhotoAgent() { };
PhotoAgent.prototype = new Agent();

PhotoAgent.prototype.onInit = function(args) {
  this.photo = args.photo;
  this.destClient = args.destClient;
  this.sessionId = args.sessionId;
  this.moveToServer();
};

PhotoAgent.prototype.onArrived =
    function(hap, isServer) {
  if (isServer) {
    var helper = this.getRadigostHelper();
    helper.persist(this.sessionId, this.photo);
    helper.moveToClient(this.aid,
      this.destClient);
  } else { // on the dest client, show the
photo
    this.onStep(this.photo);
  }
};
```

In conclusion, the case study demonstrates the benefits of combining agents with HTML5 and Java EE technologies. By following the well-established and widely-used software development standards, the Siebog multiagent middleware can seamlessly integrate software agents into modern web and enterprise applications.

5. Related work

This section provides a comparison of Siebog and other existing multiagent middlewares. In the first part, the focus is on the client-side (i.e., Radigost-provided) features of our framework, while the second part compares its server-side (i.e., XJAF-provided) functionalities.

Generally speaking it is evident lack of contemporary research efforts and published papers concerning fault tolerance in agent clusters which is, in fact, one of the main advantages of our system. Few papers [22], [23] deal only with theoretical aspects of the fault tolerance in agent clusters, but they are rather out of date. On the other hand, only paper [1] deals with industrial-based solutions for agent clustering, and that paper is based on and uses XJAF.

To the best of our knowledge, papers regarding client-side agents are also not so common. According to that, the following subsections will describe related work in both areas (client-side and server-side) using widely available, contemporary and frequently used agent frameworks, middlewares and architectures.

5.1. Comparing the client-side features of Siebog

Currently, there exist several web-based multiagent middlewares. *JACK Intelligent Agents WebBot* utilizes several enterprise Java technologies. The framework is executed on the server, and consists of three layers: Servlet Container, which acts as an interface to the application front-end *JACK application*, which contains the actual agent code, and *WebBot*, which acts as an intermediary between the other two layers.

The popular multiagent framework JADE is known for its extensibility [15]. The web support is added through *JadeGateway* that acts as a link between server-side agents and remote clients. An additional Java-based web agent framework is *JaCa-Web*, that enables the web-based execution of *Jason* agents, along with the *CArtAgO* framework for artefacts' modeling [24].

Smart Python multi-Agent Development Environment (SPADE) is a client-server architecture [25]. It relies on the existing communication protocol for agent interaction, and offers a number of FIPA-standardized agent services.

The main difference between Siebog and all of these middlewares is that its client side (i.e., Radigost) is developed in the manner of modern web applications: using the HTML5 set of technologies. An important advantage of this approach is greater platform-independence: Siebog is the only system among this group that requires no virtual machine or browser plug-in to run. Also, it is readily available to end-users, without any installation or configuration steps. Finally, unlike *JACK WebBot* and *JadeGateway*, Siebog agents are actually executed on the client side, reducing the server load.

The only other purely HTML5-based agent platform that we are aware of is described in [21]. There are several important differences between the two systems. Siebog is more advanced on the client side, as it fully utilizes the advantages of Web Workers and the WebSocket protocol. On the server side, while Siebog relies on Java EE, their platform conveniently uses *Node.js*, a JavaScript-based server framework. Although this approach simplifies the implementation of certain functionalities, such as agent mobility, it lacks the cluster-based features of Java EE available in XJAF.

5.2. Comparing the server-side features of Siebog

Currently, there are several multiagent middlewares that offer agent load-balancing and/or fault-tolerance. *Cognitive Agent Architecture* (Cougaar) is a Java-based distributed agent architecture specifically designed for unstable environments [26]. Cougaar provides state persistence and error recovery for its agents. Since its internal components are designed as agents, they are protected by the fault-tolerance subsystem too. Agent distribution and fault-tolerant features in Cougaar are more powerful than those found in XJAF. However, our system demonstrates how many of these features can be realized with much less effort and much fewer resources, by using standard and ready-made solutions in Java EE.

Magentix is a Linux-based multiagent middleware. It is built with the runtime performance as the primary focus [27]. For this purpose, the system is heavily based on low-level features offered by the operating system. For example, each agent is represented by a Linux process with three internal threads. The platform itself can be distributed across a number of computers. Although it achieves remarkable runtime performance, Magentix lacks previously described features of XJAF that stem from the use of computer clusters.

JADE is a popular, Java-based multiagent middleware [15]. It supports the development of both reactive and cognitive agents, and features an extensive ecosystem of plug-ins. JADE's agent containers can be distributed across a computer network, and have a support for fault-tolerance at the container level. XJAF provides more advanced load-balancing and fault-tolerant features than JADE. And as shown in [1], it performs better in scenarios with large populations of agents.

Whitestein LS/TS includes a set of tools, a UML-based modeling language, and a high-level library for developing agents [28]. It's Java-based and offered in three separate editions. However, its higher-level abstractions enable an agent written once to execute on any of the editions without changes. Although the *Enterprise* edition of *Whitestein LS/TS* is developed in Java EE, it is a commercial product, and so a deeper analysis could not be performed.

Finally, as it can be concluded from the presented analysis, none of the described multiagent middlewares provides the combination of features available in Siebog, namely, the HTML5-based agent support on the client side, and the Java EE-based agent support for clustered environments on the server side. This combination of functionalities is in line with modern approaches to enterprise web application development, enabling an easier integration of Siebog and its agents into mainstream enterprise solutions.

6. Conclusions and future work

Siebog is a multiagent middleware that builds on the successes of HTML5 and Java EE. The main goal of this middleware is to offer high performance and platform independence for software agents. As shown in this paper, those features are realized using modern web and enterprise application development standards. In this way, any multiagent system built on top of Siebog has several important features.

On the client side, Siebog offers true platform-independence. By running in web browsers, Siebog agents can be executed on a wide variety of hardware and software platforms, including desktops, smartphones and tablets, and Smart TVs. This is beneficial to both agent developers, which can write agents in the *write once, run anywhere* manner, and to end-users, which can access their Siebog-based applications in the most convenient way.

On the server side, Siebog runs on a top of computer clusters offering a high-availability of deployed applications. That is, the system achieves scalability through automated agent load-balancing and fault-tolerance through agent state replication and failover. These advanced functionalities were not implemented from scratch. Instead, Siebog uses standard, readily made technical solutions of Java EE.

As discussed earlier, the Siebog middleware is not simply a set of individual components. Our earlier multiagent middlewares, Radigost and XJAF, have been integrated in a way that enables cross-platform agent interaction, code sharing, and even heterogeneous agent mobility.

The future work on Siebog will be focused on a number of areas. Here, security plays an important role. As shown in the case study, a mobile agent can easily move from one client to another, while carrying custom data with it. This could present serious security issues, and therefore a full assessment of possible security flaws is required.

The social ability of agents is one of the defining characteristics of the agent technology. Therefore, we are currently incorporating a number of standard interaction and action coordination approaches into our system. The main focus is on *reliability*: Siebog will employ state replication in order to assure that the interaction and coordination sub-systems continue to operate regardless of hardware and software failures.

For any system to become production-ready, applications that are more practical are needed. Given the highly-distributed nature of Siebog, the initial focus will be on applications in the field of ant colony optimizations [11]. Future work will also include practical applications in web environments, including web service management [29], personalization and recommendation of online material [30], [31], grid computing [31], online games [20], web crawling [33], and various kinds of online pedagogical agents [34], [35]. Having in mind that Siebog provides the necessary infrastructural support for fault tolerant clustered environment as well as having agents running in wide range of devices, many of the above-mentioned applications can benefit from using Siebog.

Currently, Siebog is suitable for developing not only reactive, but also reasoning *BDI (belief-desire-intention)* agents. For the latter, it includes a Java EE version of the AgentSpeak [36] that is an agent-oriented programming language for the popular Jason platform. We are also in the process of developing a unique, distributed reasoning engine.

Finally, having in mind all advantages, mentioned within the paper, that our system posses and offers, it represents also a good starting point for development of more sophisticated and „clever“ agents [37]. For example Agreement technologies [38], [39] play important role in wide range of computer systems and environments in which autonomous agents negotiate usually on behalf of humans to achieve mutually acceptable agreements. Interactions between agents and environment must be also supported by sophisticated activities like reasoning, learning, or planning. Concepts like norms, trust, reputation, and argumentation are essential in interactions between agents within such systems. Obviously it is not easy task to handle, maintain and incorporate all such sophisticated concepts within an integral framework. But, possibilities for practical development and real applications of high-quality, high-performance, and highly reliable agents based on agreement technologies with Siebog are promising. Additional superiority of Siebog agents in the area of agreement technologies lies in the fact that they can easily run on a wide variety of software and hardware platforms and can harness the benefits of clustered environments.

Acknowledgements

This work was partially supported by Ministry of Education, Science and Technological Development of the Republic of Serbia, through project no. OI174023: “Intelligent techniques and their integration into wide-spectrum decision support.”

References

[1] **D. Mitrović, M. Ivanović, M. Vidaković, Z. Budimac**. Extensible Java EE-based agent framework in clustered environments. *Lecture Notes in Computer Science*, 2014, Vol. 8732, 202-215.

[2] **D. Mitrović, M. Ivanović, M. Vidaković, Z. Budimac**. A scalable Distributed architecture for web-based software agents. *Lecture Notes in Computer Science*, 2015, Vol. 9329, 67-76.

[3] **S. Xinogalos, K. E. Psannis, A. Sifaleras**. Recent advances delivered by HTML 5 in mobile cloud computing applications: a survey. In: *Proceedings of the Fifth Balkan Conference in Informatics, (BCI 2012)*, ACM, New York, NY, USA, 2012, pp. 199-204.

[4] **D. Mitrović, M. Ivanović, C. Bădică**. Delivering the multiagent technology to end-users through the web. In: *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics*, Article no. 54, ACM, New York, NY, USA, 2014.

[5] **D. Mitrović, M. Ivanović, Z. Budimac, M. Vidaković**. Radigost: Interoperable web-based multi-agent platform. *Journal of Systems and Software*, 2014, Vol. 90, No. 4, 167–178.

[6] **D. Mitrović, M. Ivanović, Z. Budimac, M. Vidaković**. Supporting heterogeneous agent mobility with ALAS. *Computer Science and Information Systems*, 2012, Vol. 9, No. 3, 1203-1229.

[7] **M. Vidaković, M. Ivanović, D. Mitrović, Z. Budimac**. Extensible Java EE-based agent framework – past, present, future. In: M. Ganzha and L. C. Jain, (eds.), *Multiagent Systems and Applications*, Vol. 45 of *Intelligent Systems Reference Library*, pp. 55-88, Springer Berlin Heidelberg, 2013.

[8] **V. Dignum**. An overview of agents in knowledge management. In: *Proceedings of INAP-05, Lecture Notes in Artificial Intelligence*, 2006, Vol. 4369, pp. 175-189.

[9] **H. Nakashima, H. Aghajan, J. C. Augusto**. Handbook of ambient intelligence and smart environments. Springer, 2010.

[10] **B. Vesin, M. Ivanović, A. Klašnja-Milićević, Z. Budimac**. Protus 2.0: Ontology-based semantic recommendation in programming tutoring system. *Expert Systems with Applications*, 2012, Vol. 39, No. 12, 12229-12246.

[11] **S. Ilie, C. Bădică**. Multi-agent approach to distributed ant colony optimization. *Science of Computer Programming*, 2013, Vol. 78, No. 6, 762-774.

[12] Siebog: An Enterprise-Scale Multiagent Middleware. *Github repository*. Available: https://github.com/gcvt/siebog.

[13] Distributed Artificial Intelligence Course. *Faculty of Technical Sciences, University of Novi Sad*. Available: http://informatika.ftn.uns.ac.rs/DVIIA/

[14] **A. Gonclaves**. Beginning Java EE^{TM} 6 platform with GlassFish 3, Second Edition. ISBN: 978-1-4302-2889-9, Apress Media LLC, 2010.

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

[16] **G. Hohpe, B. Woolf**. Enterprise integration patterns: designing, building, and deploying messaging *solutions. ISBN: 978-0321200686,* Addison Wesley, 2003.

[17] **R. T. Fielding**. Architectural styles and the design of network-based software architectures. PhD Dissertation, University of California, Irvine, 2000.

[18] **G. Mulligan, D. Gracanin**. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. In: *Proceedings of the 2009 Winter Simulation Conference*, 2009, pp. 1423-1432.

[19] **M. Grogan**. JSR-223: Scripting for the Java^{TM} platform. Sun Microsystems, Inc. 2006.

[20] **A. Tacy, R. Hanson, J. Essington, A. Tökke**. GWT in action, second edition. ISBN: 978-1935182849, *Manning Publications*, 2014.

[21] **L. Jarvenpaa, M. Lintinen, A.-L. Mattila, T. Mikkonen, K. Systa, J.-P. Voutilainen**. Mobile agents for the internet of things. In: *17th International Conference on System Theory, Control and Computing*, 2013, pp. 763-767.

[22] **K. Park**. A Fault Tolerant Mobile Agent Model in Replicated Secure Services. *Lecture Notes in Computer Science*, 2004, Vol. 3043, 500-509.

[23] **H. Pals, S. Petri, C. Grewe**. FANTOMAS Fault Tolerance for Mobile Agents in Clusters. *Lecture Notes in Computer Science*, 2000, Vol. 1800, 1236-1247.

[24] **M. Minotti, A. Santi, A. Ricci**. Developing web client applications with Jaca-Web. 11th Workshop nazionale “Dagli Oggetti agli Agenti”, September 2010.

[25] **E. Argente, J. Palanca, G. Aranda, V. Julian, V. Botti, A. Garcia-Fornes, A. Espinosa**. Supporting agent organizations. In: Burkhard, H.D., Lindemann, G., Verbrugge, R., Varga, L. (eds.) Multi-Agent Systems and Applications V, Lecture Notes in Computer Science, 2007, Vol. 4696, pp. 236-245.

[26] **S. Siracuse, R. Tomlinson, T. Wright, J. Zinky**. Experience with task/allocation coordination primitive for building survivable multi-agent systems. In: *International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, April 2007 pp. 40-45.

[27] **J. M. Alberola, J. M. Such, V. Botti, A. Espinosa, A. Garcia-Fornes**. A scalable multiagent platform for large systems. Computer Science and Information Systems, 2013, Vol. 10, No. 1, 51-77.

[28] **G. Rimassa, M. Calisti, M. E. Kernland**. Living systems ® technology suite. In: R. Unland, M. Klusch, M. Calisti, (eds.), *Software Agent-Based Applications, Platforms and Development Kits*, pp. 73-93, Birkhauser Verlag, 2005.

[29] **J. Bentahar, Z. Maamar, D. Benslimane, P. Thiran**. Using argumentative agents to manage communities of web services. In: *21st International Conference on Advanced Information Networking and Applications Workshops*, May 2007, Vol. 2, pp. 588-593.

[30] **P. Lops, M. Gemmis, G. Semeraro**. Content-based recommender systems: State of the art and trends. In: F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, (eds.), *Recommender Systems Handbook*, pp. 73-105. Springer US, 2011.

[31] **R. M. Swezey, S. Shiramatsu, T. Ozono, T. Shintani**. Intelligent page recommender agents: real-time content delivery for articles and pages related to similar topics. In: K. G. Mehrotra, C. K. Mohan, J. C. Oh, P. K. Varshney, and M. Ali, (eds.), *Modern Approaches in Applied Intelligence*, 2011, Vol. 6704 of *Lecture Notes in Computer Science*, pp. 173-182.

[32] **J. Cao, D. P. Spooner, S. A. Jarvis, G. R. Nudd**. Grid load balancing using intelligent agents. *Future Generation Computer Systems*, 2005, Vol. 21, No. 1, 135–149.

[33] **L. Ding, R. Pan, T. Finin, A. Joshi, Y. Peng, P. Kolari**. Finding and ranking knowledge on the semantic web. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, (eds.), Vol. 3729 of *Lecture Notes in Computer Science*, pp. 156–170. Springer Berlin Heidelberg, 2005.

[34] **Y.-M. Cheng, L.-S. Chen, H.-C. Huang, S.-F. Weng, Y.-G. Chen, C.-H. Lin**. Building a general purpose

pedagogical agent in a web-based multimedia clinical simulation system for medical education. *IEEE Transactions on Learning Technologies*, 2009, Vol. 2, No. 3, 216–225.

[35] **M. Ivanović, D. Mitrović, Z. Budimac, M. Vidaković**. Metadata harvesting learning resources - an agent-oriented approach. In: *Proceedings of the 15th International Conference on System Theory, Control and Computing*, October 2011, pp. 306-311.

[36] **R.H. Bordini, J.F. Hubner**. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, (eds.), *Computational Logic in Multi-Agent Systems*, Vol. 3900 of *Lecture Notes in Computer Science*, pp. 143-164, Springer Berlin Heidelberg, 2006.

[37] **S. Heras**. Case-based argumentation framework for agent societies. *Ph.D. thesis, Departamento de Sistemas Informáticos y Computación. Universitat Politècnica de València*, 2011.

[38] **S. Ossowski**. Agreement Technologies, *Law, Governance and Technology Series*, 2013, Vol. 8, No. XXXV.

[39] **M. Ivanović, Z. Budimac**. Agreements Technologies - Towards Sophisticated Software Agents, *Computational Collective Intelligence Technologies and Applications*, 2015, Vol. 8733, 105-126.

Received July 2015.