

A scalable distributed architecture for client and server-side software agents

Mirjana Ivanović¹ · Milan Vidaković² · Zoran Budimac¹ · Dejan Mitrović¹

Received: 4 May 2016 / Accepted: 7 September 2016 / Published online: 16 September 2016
© The Author(s) 2016. This article is published with open access at Springerlink.com

Abstract This paper describes recent developments of the Siebog agent middleware regarding performance. This middleware supports both server-side and client-side agents. Server side agents exist as EJB session beans on the JavaEE application server, while client-side agents exist as JavaScript Worker objects in the browser. Siebog employs enterprise technologies on the server side to provide automatic agent load-balancing and fault-tolerance. On the client side this distributed architecture relies on HTML5 and related standards to support smooth running on a wide variety of hardware and software platforms. Such architecture supports rather easy, reliable and efficient communication, interaction, and coexistence between numerous agents. With the automatic clustering and state persistence, Siebog can support thousands of server-side agents, as well as thousands of external devices hosting tens of client-side agents. Performed and presented experiments showed promising results for real life applications of our architecture.

Keywords Agent middleware · Clustering · Load-balancing · Fault-tolerance · HTML5

✉ Mirjana Ivanović
mira@dmi.uns.ac.rs

Milan Vidaković
minja@uns.ac.rs

Zoran Budimac
zjb@dmi.uns.ac.rs

Dejan Mitrović
mitrovic.dejan@gmail.com

¹ Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Novi Sad, Serbia

² Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia

1 Introduction

During the last decade, there has been an obvious paradigm shift in software development. The web has evolved into an environment capable of providing functionalities not so long ago available only in desktop applications. One of the main reasons for the increasing popularity of web-only applications is their cross-platform nature, which allows the end-users to access their favorite applications in a wide variety of ways and devices. As the overall result, the server-based generation of web content is becoming less and less relevant, having browser-based code (namely JavaScript) to interact with the user and perform just like the desktop applications used to work.

Computer clusters are unavoidable nowadays and they play an important role in modern web and enterprise applications and in development of complex software applications. They provide high-availability of deployed applications [1]. This feature provides continuous, uninterrupted delivery of services, regardless of hardware and software failures, or numbers of incoming requests. The high-availability is achieved through the so-called horizontal scaling, which is the process of adding more nodes to the cluster as the demands for processing power increase.

Siebog is our multiagent middleware designed and implemented to provide support for intelligent software agents in clustered environments [2]. It efficiently combines the HTML5 and related web standards on the client side [3,4], and the Enterprise edition of Java (Java EE) on the server side [2,5,6], in an effort to bridge the gap between the agent technology and useful industry applications.

By utilizing the standards and technologies readily-available in Java EE, Siebog offers “native” support for computer clusters on the server. The purpose of this paper is to provide results of clustering of agents on the server

side, and to discuss how this support is extended to the client side of Siebog as well. The goal is to provide the support for clusters that consist of arbitrary client devices, such as personal computers, smartphones, and tablets.

The motivation for this approach is straightforward. There can be an order of magnitude more external client devices than there can be server-side computers. Siebog could be used to distribute agents among connected clients and to support applications that require launching large populations of agents. Since the state-of-the-art smartphones have more processing power than many laptop or desktop machines, they represent a significant computational resource.

This approach, however, does pose some technical challenges. Client-side clusters are highly dynamic, in the sense that clients are able to join and leave at any time. To deal with this situation, we added a highly-scalable infrastructure for agent state persistence which allows the agents to “rise above” the interruptions and to operate regardless of their physical locations.

The rest of the paper is organized as follows. Section 2 discusses the overall motivation behind this paper, and presents relevant related work. Section 3 presents our approach for server-side clustering, while Sect. 4 shows how the support for dynamic and heterogeneous client-side clusters was incorporated into the Siebog. The experimental evaluation of proposed architecture and overall performance of the evaluation are presented in Sects. 5 and 6 (server-side and client-side, respectively). The final conclusions and future research directions are given in Sect. 7.

2 Background

With the rise of popularity of multi-agent systems, a large number of different software tools, systems, platforms and environments that allow development and deployment of multi-agent systems and their applications in different domains have been designed and implemented. Most of them rely on proprietary solutions, while only a handful of them use some industrial-ready solution integrated in the system. As a consequence, most of the multi-agent systems do not offer clustering, or has it implemented on a very poor level. On the other hand, the Siebog agent middleware uses as much JavaEE technology as possible for most of its subsystems. For example, it uses Java Messaging System (JMS) to exchange messages, Java Naming and Directory Interface (JNDI) for agent lookup, Enterprise Java Beans (EJB) for agent implementation, etc. This enables Siebog to use tested and proven solutions implemented into the JavaEE application servers, having clustering, load-balancing, and safe-failover working out of box. Significant functionality and challenge of our architecture is that agents deployed in the Siebog can operate on clusters, being able to sur-

vive node failures and being able to serve numerous clients [2,4].

On the client side, we have web applications playing an increasingly important role in the contemporary computing. They offer a number of advantages over traditional desktop application, such as the lack of need for installation, configuration, or upgrade. The importance of web applications is emphasized by the continuously increasing sales of mobile devices and the ability of web applications to run as native applications on these devices [7].

To maintain its relevance in this new era, the agent technology not only needs to move to the web, but it needs to do so in accordance to the modern standards and the end-users’ expectations. Agent-based applications need to seamlessly be integrated into web and enterprise applications to reach the end-users more easily, and to stay relevant in this new state of affairs.

Current research in the agent technology area is very dynamic and promising, there exists a large number of both open-source and commercial agent middlewares [8,9]. However, almost none of these systems have fully exploited the advantages of web environments. Some efforts aimed at extending existing systems with web support have been made, but usually in an inefficient manner. For example, in many Java-based middlewares, such as JADE [10] or JaCa-Web [11], the extensions are based on Java applets. But, Java applets require a browser plug-in to run, which is unavailable on some platforms (e.g. iOS and Smart TVs). With some desktop-based browsers also starting to disable Java support, the applicability of Java-based web solutions becomes limited to a narrower set of hardware and software platforms.

One of the prominent and promising ways of migrating and intensively using agent technology in the web is to use the expanding HTML5 and related set of standards [12]. HTML5 covers various aspects of web and enterprise applications, from audio and video playbacks, to offline application support, to more advanced features, such as multi-threaded execution and push-based communication. In addition, since web browser vendors keep investing significant resources into improving the overall performance of their respective JavaScript virtual machines, the HTML5 is expected to become “a mainstream enterprise application development environment” in near future [13]. One of the important elements in our approach, comparing to other existing approaches, is the usage of essential HTML5 concepts for client-side agents that enables them to “spread” over wide range of client browsers.

2.1 Related work

As outlined in [9], a large number of multi-agent middlewares has been developed over the years. However it appears that not all of those systems are still being actively devel-

oped and/or used today. Most of them have been developed for academic and scientific purposes within specific projects and their development and use finishes after completion of the projects. As the first step in making agent development more popular and practically applicable is to allow agents' frameworks and architectures to be opened and freely available to wide range of agent communities. It offers other research groups possibility to actively use them and also further improve and update them. For this reason the full source code of our system is freely available [2].

Agent Developing Framework [14] enables the user to build an interoperable, flexible, and scalable application. Agent Developing Framework uses Java EE technologies such as JNDI, JMS and JMX. Communication is done synchronously or asynchronously through JMS (Java Message Service). Although this framework uses the same technology as the Siebog, ADF is no longer maintained. Currently downloadable version is from the year 2005 and practically is not any more adjusted to the modern trends and technologies of development multi-agent systems.

Voyager [15] is middleware software designed for distributed application development, and it does provide the option of developing applications using multi-agent programming, although that is not its main purpose. Latest version available is Voyager 8.0. It's a simple yet powerful technology for creating mobile agents in Java. It represented an improvement over other existing platforms (Concordia, Aglets, Odyssey, etc.) which only allowed developers to create agents and launch them into a network to fulfill its mission. But none of the mentioned platforms and middleware allowed sending messages to a moving agent, which made it difficult to communicate with an agent once it has been launched and also for agents to communicate with other agents. Voyager seamlessly integrated fundamental distributed computing with agent technology. Voyager provides flexible life spans for agents, by supporting a variety of life span methods. In spite the fact that this framework supports scalability and fault-tolerance as our middleware, its main drawback comparing to our system is that it is a commercial product.

JADE [10] is a MAS written in Java and strongly adherent to the FIPA [16] standard and provides a wide range of functionalities to agent developers, either as built-in features, or through its extensive ecosystem of plug-ins. The system itself can be executed as a set of containers on top of a computer network. Fault-tolerance is achieved through both container and agent state replication processes.

In [17] authors have established through experiments a correlation between the number of computers and the latency, as well as between the number of computers and throughput. They have used JADE in an online auction system for their experiments.

In [18] authors have established a set of benchmarks which are used to measure performance of the JADE. We were

inspired by this paper as well as by [19] when we measured performance of the Siebog middleware.

Siebog, the system we have been developing for several years uses JMS for message exchange, while JADE has its own system. The main difference between these two systems, however, lies in the fact that, when creating clusters, JADE agents have to be manually divided between cluster nodes, while in Siebog it is done automatically. So another important characteristic of Siebog system is that in it an agent is defined at the level of the computer cluster, and not at an individual node.

Two systems are applicable to different scenarios. We argue that, due to its clustering features, Siebog represents a better solution for applications that need to launch large populations of agents (e.g. [20]), and/or need to provide high-level of fault-tolerance. For example, JADE consumes a single thread per agent and has a predefined number of message processing threads. In Siebog, these numbers are increased or decreased automatically, depending on the current load. For other use-cases, using JADE might represent a better approach, since it consumes less resources and its usage is a bit simpler.

Along with the more recent trends, there have been several proposals of using mobile agents within the so-called Internet of Things (IoT) concept [13], in smart objects [21, 22] and in smart cities [23]. The role of Siebog in these practical applications is possible and feasible and would be in providing a standards-compliant, platform-independent, and efficient [4] multiagent middleware. In addition, with the work presented in this paper, we intend to bring the more traditional agent applications to the web. As it is discussed later, Siebog is suitable and reliable for distributed systems with large populations of agents. A concrete example of its possible practical application would be in the area of swarm intelligence, as it is presented [23].

As we discussed previously in [3, 4], many traditional (i.e. desktop- or server-based) multiagent middlewares have exposed their functionalities to the web through Java applets. This approach does provide many important benefits, such as the immediate availability of complex reasoning agents in web browsers [11]. However, with the lack of Java support in many popular modern platforms, this approach is no longer sufficient.

To the best of our knowledge, currently there exists only one additional HTML5-based multiagent middleware [24, 25]. The middleware is focused on using (primarily) mobile agents to support the IoT requirements while our primary attention is to develop system that could be widely used in different areas and environments. On the technical viewpoint, the client side of their system, in comparison to our approach, does not utilize the full range of HTML5 and related standards (such as Web Workers [12]). In approach presented in [24, 25] it is also not clear how multiple agents could be

started within the same host, and how would they interact with each other without the server. The authors show how the most advanced variant of moving code, mobile agents, can be used for operating and managing Internet-connected systems composed of gadgets, sensors and actuators. They pointed out that the use of mobile agents brings several benefits but practical use is not presented clearly. One of them is that mobile agents help to reduce the network load, overcome network latency, and encapsulate protocols. The need for moving agents is even more significant if the applications and other factors of the overall experience should follow the user to new contexts. When multiple agents are used to provide the user with services, some mechanisms to manage the agents are needed. In the context of Internet-of-Things such management should reflect the physical spaces and other relevant contexts. The backend in their approach is conveniently based on Node.js, which simplifies certain development aspects (such as mobility), but lacks several advanced features found in Siebog, namely automated agent load-balancing and fault-tolerance.

To summarize above discussion we can pointed out that Siebog is a web-based, enterprise-scale multiagent middleware. Its uniqueness is that it combines the enterprise technologies on the server with HTML5 and related standards on the client. Such combination together with scaling possibilities represent the main contribution, advantages and differences of our approach comparing to above presented systems and approaches. As well usage of well-proven industrial solution for the JavaEE application server provides server-side agents with the possibility of operating in clusters with features like load-balancing and safe-failover working out of box. Additionally the usage of essential HTML5 concepts supports client-side agents to “spread” over enormous number of client browsers, with the number of instances.

3 Clustering server-side Siebog agents

In this section we will briefly cover Siebog implementation of clustered agent middleware. The goal of Siebog is to provide an infrastructure for reliable executing agents in web environments, but in accordance to the modern standards. It is built on our previous two systems, XJAF [6] and Radigost [4], in a way that it not only combines their individual functionalities, but it also adds new features on both server and client side. On the server side Siebog offers:

- **Scalability:** Agents are automatically distributed across the cluster to reduce to computational load of individual nodes. This makes Siebog suitable for applications that need to launch large populations of agents in a computer cluster.

- **Fault-tolerance:** The state of each server-side component, including agents themselves, is copied to other nodes making the whole system resilient to hardware and software failures.

Siebog heavily depends on the JavaEE application server features such as clustering, load-balancing and safe-failover. Currently, the Siebog is deployed on the JBoss application server called Wildfly [26]. The framework is organized as a set of loosely-coupled components called managers, as shown in Fig. 1. Each manager is dedicated to handling a distinct part of the overall functionality. A manager is represented and used only by its interface, and even multiple implementations of the same interface can be active simultaneously. AgentManager keeps record of available and used agents. MessageManager delivers messages to agents, while ConnectionManager keeps track of other Siebog instances. This design approach offers the highest level of flexibility, and allows third-party re-implementations of individual components.

The organization of the Siebog cluster is shown in Fig. 2. A single node within the cluster is described as master, while the others (zero or more) are described as slaves. Within a node, the JBoss host controller is used to manage the Siebog instance [26]. In addition, the master node can be used to remotely control the entire cluster, through the JBoss domain controller [26]. This is the only difference between the master and the slaves; all nodes in a cluster have the same execution priority, can directly communicate to each other, etc.

The preferred approach of inter-node communication and information sharing is given through the Infinispan cache system [3]. Infinispan cache is one of the core clustering technologies used by JBoss. It is a distributed, concurrent and highly-efficient key/value data structure. Infinispan cache represents the backbone of the state replication and failover process described later, but it can also store arbitrary user data. Whenever it runs a new agent, for example, the agent manager stores all the necessary information in the Infinispancache (e.g. agentIdentifier -> beanInstance). This information can later be retrieved by the message manager to deliver a message to the agent. Since the cache is distributed across the cluster, the managers themselves can be hosted on any node. In fact, for maximum performance, they are implemented as clustered stateless beans by default.

The cluster has two main functionalities: state replication and failover&load-balancing. State replication and failover are applicable to stateful beans only. Whenever a stateful bean’s internal state is changed, the replication process copies it across other nodes in the cluster. In case the bean’s node becomes unavailable, the failover process fully restores the bean object on one of the remaining nodes. From the client’s

Fig. 1 Architecture of the Siebog agent middleware

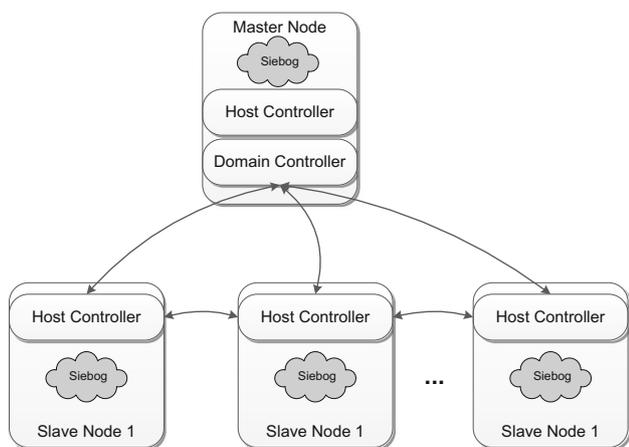
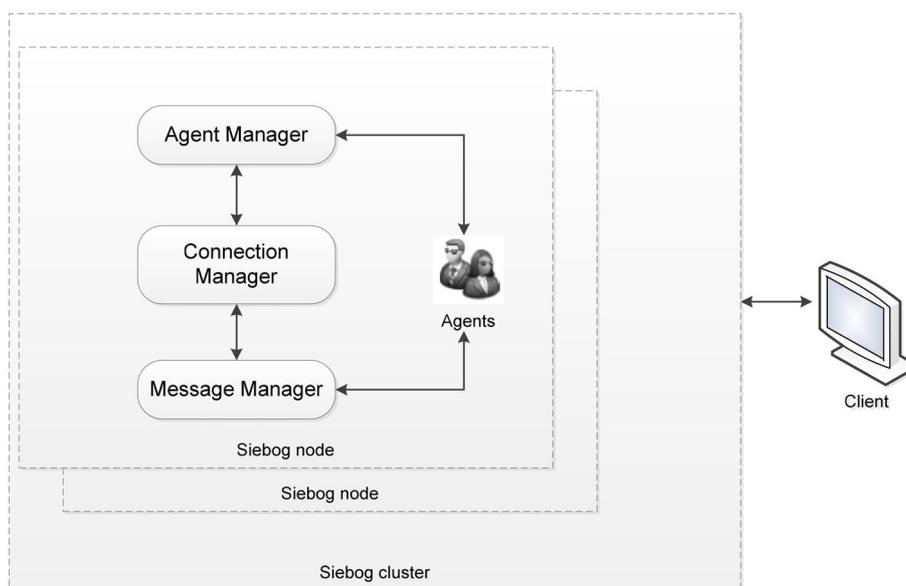


Fig. 2 Siebog operates in a symmetric cluster; each node is connected to other node

point of view, the entire process is executed transparently: all subsequent method invocation will end-up in the newly created object.

Load-balancing is used to automatically distribute agents across different nodes in the cluster, and to speed up the overall runtime performance of Siebog. It works with both stateful and stateless beans, although the behavior is slightly different. When the client creates a new stateful bean instance, the server places it in one of the available nodes, and all subsequent invocations of the bean’s methods end-up there. In case of stateless beans, the load-balancing works on a per-method basis. At any time, there can be many instances of the same stateless bean running in parallel across the cluster. Once the client invokes a method of the bean, one of the instances is selected to serve the request.

4 Clustering client-side agents

As we already mentioned in previous section Siebog provides an infrastructure for executing agents in web environments, adding new features on both server and client side.

The client-side component of Siebog has the following set of unique characteristics:

- It is platform-independent, supporting a range of hardware and software platforms. To agent developers, this provides the write once, run anywhere approach. The end-users, on the other hand, can utilize the benefits of the agent technology in the most convenient manner.
- It requires no prior installation or configuration steps.
- Its client-side runtime performance is comparable to that of a classical, desktop multiagent platform

In this section we will describe how Siebog is extended to support automatic clustering and load-balancing of its client-side agents [27]. More concretely, we discuss how a possibly large set of heterogeneous client-side devices can be observed as a coherent cluster. The cluster can then be used to execute resource-demanding and computationally-expensive tasks, such as launching large populations of agents.

The support for the client-side clustering is shown graphically in Fig. 3. On the server side, Siebog introduces one more manager, for the client-side agent management: WebClient Manager, which acts as an intermediary for server-to-client (i.e. push) messaging, and also handles state persistence for client-side agents.

Client-side agents are executed inside web browsers [2–4] or possibly in dedicated JavaScript runtimes of external devices. Inside a device, agents rely on the Siebog client

Fig. 3 Client-side clustering support in the Siebog middleware

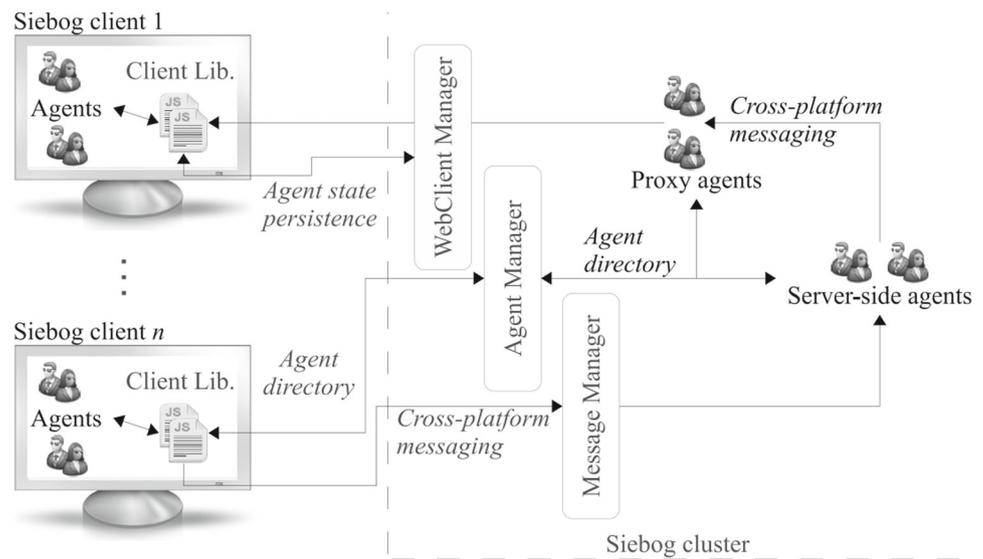
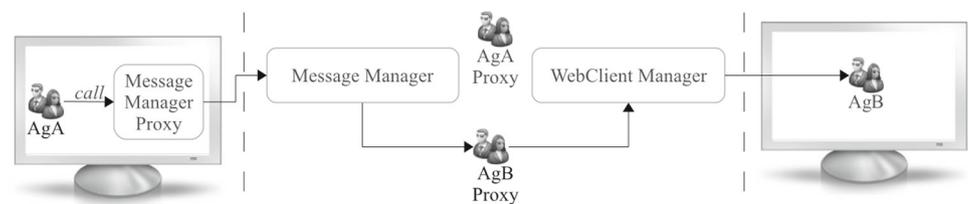


Fig. 4 Transparent communication of client-side agents across different devices



library for execution support and for communicating with the server. For example, the client library offers proxy implementations of server-side components. To send a message to the server-side agent, the client-side agent simply invokes the proxy implementation of the MessageManager. Underneath, the proxy then turns this invocation into a corresponding AJAX call to the server.

Another important feature of Siebog is that its server can (if configured) hold a proxy representation of each client-side agent. This representation simply forwards all incoming messages to the corresponding client-side counterpart (and through the WebClient Manager). This feature opens-up the possibility for transparent agent communication across different devices.

An illustrative example is shown in Fig. 4. Suppose that there exist two agents, AgA and AgB, hosted by two different devices, Device A and Device B, respectively. When the AgA decides to send a message to AgB, the message will be delivered in the following way:

- AgA makes the appropriate call to the MessageManager Proxy.
- This call is transformed into an AJAX call to the server-side MessageManager.
- The MessageManager delivers the message to the AgB proxy.

- Since this is a proxy representation, it forwards the message to the WebClient Manager.
- The WebClient Manager, which is aware of all external clients, finally pushes the message to the target agent.

Due to limitations imposed by certain web browsers [4], each client device can run up to a few dozens of agents. But, there can be large number of physical devices that are active simultaneously. The main idea here is to exploit this possibility to distribute portions of a large population of agents. This is conceptually similar to, for example, the famous SETI@home scientific experiment. As an important advantage, the Siebog does not require any software installation: all the end-user needs to do is to visit the corresponding web page. The main issue here is how to efficiently support these large numbers of external devices, and to deal with their inherently dynamic availability.

4.1 Managing heterogeneous and dynamic clusters

A central component in a computer cluster is a load-balancer with the task of distributing the work across available machines. In the context of Siebog, the load-balancer continuously accepts tasks that need to be solved. For example, it can accept large maps for the Traveling Salesman Problem

and then partition each map [28] and send it (along with the corresponding set of ants) to a subset of available devices.

In the majority of existing non agent-based distributed architectures, the load-balancer selects the target device randomly. In this way, the workload is distributed “for free” and, in the longer run, equally among all available devices. However, the clusters that consist of Siebog clients are heterogeneous, in the sense that they can include devices with very different processing capabilities. Therefore, the load-balancing process is a bit more complex.

When it comes to load-balancing in heterogeneous systems, the agent-oriented research has proposed some rather complex approaches (e.g. [29–31]). In case of Siebog, however, we decided to follow the industry norms of keeping things as simple as possible. Once a device joins the cluster for the first time, a performance benchmark is executed. The results of this benchmark are used to assign a number of compute units (CUs) to the device. Now, during the load-balancing phase, the target device is selected with the probability that corresponds to its number of CUs.

From the end-user’s point of view, joining the Siebog cluster is fairly simple: he/she only needs to visit the appropriate web page that hosts the worker agents. Unfortunately, it is also very easy to leave the cluster; once the end-user closes the web browser or switches of the device, the hosted agents are lost. For meaningful practical applications, however, the agents need to be able to run regardless of these interruptions.

To support possibly large numbers of agents, Siebog needs a scalable datastore, one capable of serving multitudes of requests per second. The datastore should also be fault-tolerant—capable of surviving server crashes. More formally, these requirements can be described as principles of the so-called Dynamo systems [32]. Currently, there exist several concrete Dynamo realizations. After a careful evaluation of these solutions, we determined that the open-source Apache Cassandra datastore fulfills the needs of the Siebog client-side clusters. The client-side Siebog library has been extended to allow the agents the interact with the datastore directly, and over the WebSocket protocol [3,4].

The integrated Siebog architecture enables transparent inter-agent communication and action coordination, regardless of the types and physical locations of agents. A client-side agent can send a message to a server-side agent via the appropriate stub call. However, if the target agent is actually a stub representation of a different client-side agent, the message may end up in a different web page or on a different device. This opens up a range of possible practical applications; for example, in case of smart environments agents hosted in physically distributed smart objects can seamlessly exchange information and coordinate their actions.

The performance evaluation of the new architecture we proposed in this paper is discussed in the following two sections.

5 Performance evaluation of server-side agents

In addition to the advanced programming features described earlier, an important factor for the wider acceptance of Siebog is its runtime performance. Therefore, a case-study has been developed to assess this aspect of our system [33]. The case-study includes a pair of agents, named Sender and Receiver. The first agent issues a request to the second, which then performs a computationally expensive task, and replies with the result. The message round-trip time (RTT) is used as a measure; it expresses the time since the Sender issues the request and until it receives the reply. This relatively simple, but effective performance study is inspired by those described in paper [4, 18, 19, 34]. More complex use-cases, e.g. implementation of an ant colony optimization algorithm for the Traveling salesman problem [20], can be found at our recent paper [2] in which the description of Siebog is given together with link to homepage of our system. There reader can find detailed manual and precise directions how to use Siebog [2].

Experimental setup was as follows:

- Hardware: Intel Dual-Core CPU at 3 GHz, with 2 GB of RAM. The CPU is capable of executing four threads simultaneously;
- 32-bit version of Ubuntu 14.04LTS;
- OpenJDK 7 for JADE, and OpenJDK 8 for Siebog; the maximum heap size for each Siebog node/JADE container was set to 512 MB;
- JBoss Wildfly 9.0;
- The Receiver agent used a brute-force algorithm for finding all prime numbers up to a certain limit;
- Each of the two messages exchanged between a Sender and the Receiver included a string of 65 K random characters.

The utilized Wildfly server has a specific feature. When one EJB (directly or indirectly) invokes a method of another EJB, the target will be executed on the same node as the source EJB. This is an optimization feature, applied to reduce expensive network communication: if two agents exchange a lot of messages, then they should reside in the same node. Although this default behavior can (and, in case of multi-agent systems, often should) be changed, it was left as-is for this case-study. This means that the Sender and its corresponding Receiver are always executed on the same node.

A set of analogous JADE agents was implemented and used as a reference point. By default, during the load-balancing process JBoss selects an available cluster node randomly. To achieve a similar distribution of agents in both Siebog and JADE implementations, and obtain more relevant results, we’ve setup the JBoss server to use a round robin node selector. The final organization of the case-study and the

distribution of agents in both implementations are shown in Fig. 5.

Two evaluation scenarios were executed. First, we measured how many agents each of the frameworks can execute per machine and how the average RTT changes as the number of agents increases. The results of this experiment are shown in Fig. 6. For lower numbers of agents, JADE offers better runtime performance. This is expected, since there is an overhead associated with remote EJB invocations. However, as the number of agents increases, Siebog scales better. Moreover, in our setup, once the number of pairs is set to 2048 (i.e. 4096 agents), JADE starts discarding messages and eventually crashes with the out-of-memory error. On the other hand, our system Siebog is perfectly capable of executing this many agents (and more), due to built-in optimization features described earlier.

The second scenario was designed to measure the scaling factor of Siebog as more and more nodes are added to the cluster. The number of agent pairs was fixed to 2048 (i.e.

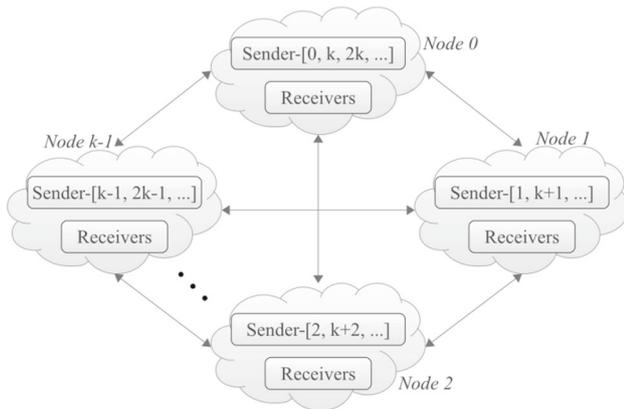
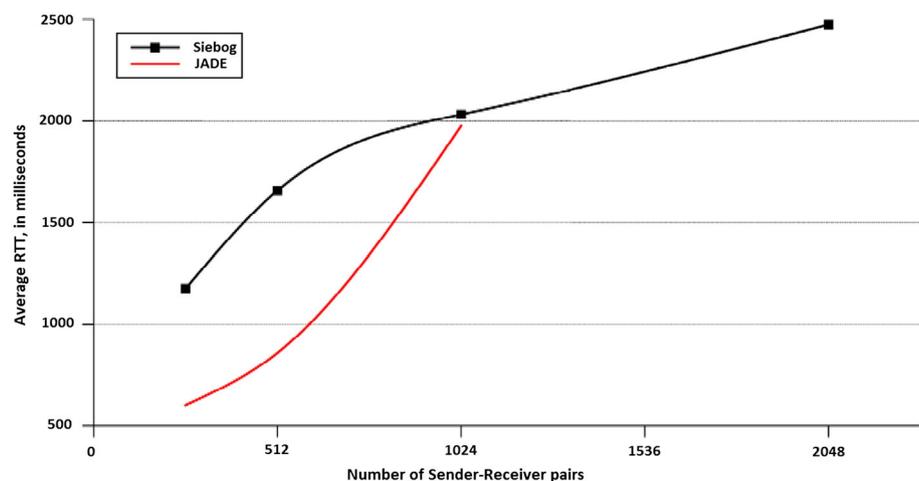


Fig. 5 Organization of the case-study and the round robin-based distribution of Sender-Receiver pairs in both the Siebog and the JADE implementation

Fig. 6 Correlations between the number of agents and the average RTTs in Siebog and JADE implementations of the Sender-Receiver experiment (prime limit = 20,000)



4096 agents), and the prime limit on the Receiver's end was set to 60,000. Four rounds of experiment were conducted: using 1, 2, 4 and 8 nodes, each having the same hardware and software configuration. In this evaluation, it was observed that, as the number of nodes doubles, the execution speed of Siebog increases approximately 3.5 times, which is an excellent outcome (the ideal would be 4 times).

These achieved and presented experimental results are very encouraging for the performances of our framework, and work in favor of the intended usage of it. They confirm the effectiveness of the inherent load-balancing capability. Along with other clustering features offered by the modern enterprise application server, Siebog represents an excellent framework for applications that require and support larger populations of agents.

6 Performance evaluation of client-side agents

The newly proposed architecture of Siebog needs to be able to serve large numbers of running agents, which are concurrently, and at high frequencies, storing and retrieving their respective internal states. To evaluate this feature, we used the open-source Yahoo! Cloud Serving Benchmark (YCSB) [35] tool. YCSB is designed for load-testing of (primarily) NoSQL databases, and can be configured through a range of parameters, including the desired number of operations per second (throughput), the number of concurrent threads, maximum execution time, etc.

The experiments were performed using two machines, each with 8 virtual CPUs and 28 GB of RAM, running 64-bit version of Ubuntu 14.04 LTS. One machine was hosting the Apache Cassandra datastore and the Siebog server (deployed on the Wildfly 9.0 application server), while the other one was used to launch YCSB-simulated external devices. Each external device was represented by a separate WebSocket

that the server needed to maintain. We have used different server machines for the client-side test, since we only had these two virtual machines. For the server-side clustering (described in the Sect. 5), we have had entire computer lab, so we were able to increase the number of computers in the cluster.

In a realistic use-case, there will be many more writes to the store than reads. That is, the internal state of an agent will usually be read only once: when the web page is loaded and the agent is started. On the other hand, the state can be stored multiple times during the agent's execution, e.g. after each processed message or after each computational sub-step. Therefore, the YCSB workload was set up as write-heavy, so that 90% of all operations are writes.

The goal of the experiment was to determine the maximum number of external devices as well as client-side agents that our system can support. For this goal, several test-cases were executed. With each successive test-case, the total number of connected devices was increased. Then we would try to find the maximum throughput (i.e. the number of operations per second) that the Siebog server can support. A test-case was executed for one hour, and the maximum throughput value that was stable during this period was taken as the end-result.

We started with 100 external devices, increased the number for each test case, and finally reached the limit of approximately 16,000 devices. This number actually represents the maximum number of open connections that the operating system could support. Nonetheless, being able to support 16,000 external devices using a single-node Siebog cluster is an excellent result, given the fact that the cluster can easily be extended with more nodes as the demands

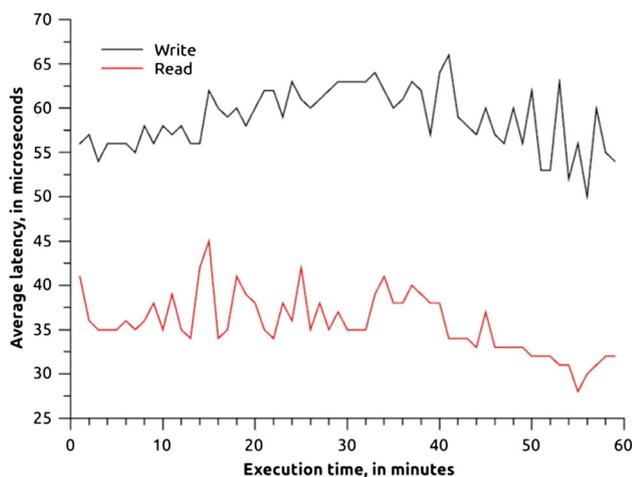


Fig. 7 Average read and write latencies of the test-case simulating 16,000 external devices and 6000 operations per second, during the one hour period. The latencies are calculated at one minute intervals

grow. The results of this test-case are shown in Fig. 7. More concretely, the figure shows average read and write latencies during the one hour period, calculated at one minute intervals. The latencies are very low (expressed in microseconds), due to the WebSocket protocol's support for asynchronous I/O.

For each test-case, through trial-and-error, we determined that the value of approximately 6000 operations per second is the maximum throughput that remained stable during the one hour period. Our system is capable of serving much larger numbers than this (i.e. up to 100,000 operations per second), but only in "short bursts," after which the backend datastore needed some time to manage all the write operations.

Although the 6000 operations per second might not seem as a large number at first, it is worth noting that an agent is not supposed to store its internal state at every second. So even if agents store their respective states at every 10 s, we reach the conclusion that our Siebog multiagent middleware can manage 60,000 agents distributed across 16,000 devices, using only one server side node.

7 Conclusions and future work

Siebog is a web-based, enterprise-scale multiagent middleware. It combines the enterprise technologies on the server with HTML5 and related standards on the client to support multiagent solutions whose functionalities meet the expectations of modern software systems. This combination of server-side and client-side technologies is the unique up to now and represents the main contribution of this paper, since it is (to the best of our knowledge) the only agent middleware which scales well on both server and client sides. With the usage of well-proven industrial solution for the JavaEE application server, we have managed to provide server-side agents with the possibility of operating in clusters with features like load-balancing and safe-failover working out of box. On the other hand, the usage of HTML5 concepts like WebWorkers and WebSockets gave the opportunity for client-side agents to "spread" over enormous number of client browsers, with the number of instances greatly surpassing the number of server-side instances. In this paper, we have, therefore, presented how Siebog was updated to support dynamic clusters of heterogeneous client-side devices, as well as to support server-side agents in a clustered environment.

The two new components of our system are the load-balancer, which is in charge of distributing agents across the connected devices, and a highly-scalable backend datastore used for persisting the internal states of client-side agents. Evaluation tests proved that the Siebog can scale up in a clustered environment and that it can handle increased load on the server-side. Server-side agents are deployed as EJB beans, so they are multiplied on the server in case of stateless

beans. In case of stateful beans, they can be reconstructed on other nodes, in situation of failure of the node they reside in. On the client side, the cluster is actually replaced with multiple browsers, having multiple JavaScript-based agents, who are capable of saving their state to the server, in case user closes the browser. Client-side agents can exchange messages between them, but they also can exchange messages with client-side agents on other browsers, as well as with server-side browsers.

For any meaningful application of Siebog, its client-side agents need to become “detached” from their host environments (e.g. web pages). As shown in the paper, thanks to the use of Dynamo architecture and the WebSocket protocol, on just one server node the state persistence system in Siebog can support thousands of external devices hosting tens of thousands of client-side agents, which is an excellent result.

Future developments of Siebog will be focused on an even tighter integration of client-side and server-side agents. Also, the system will be extended with an interoperability module, allowing it to interact with third-party multiagent solutions. Although Siebog already supports BDI agents on the server, the work is underway to develop a unique architecture for intelligent agents.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Michael, M., Moreira, J.E., Shiloach, D., Wisniewski, R.W.: Scale-up x scale-out: a case study using Nutch/Lucene. In: IEEE International Parallel and Distributed Processing Symposium, pp. 1–8 (2007)
2. Mitrović, D., Ivanović, M., Vidaković Budimac, Z.: The Siebog multiagent middleware. *Knowl.-Based Syst.* **103**, 56–59 (2016)
3. Mitrović, D., Ivanović, M., Bădică, C.: Delivering the multiagent technology to end-users through the web. In: Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS), pp. 54:1–54:6 (2014)
4. Mitrović, D., Ivanović, M., Budimac, Z., Radigost Vidaković, M.: Interoperable web-based multi-agent platform. *J Syst Softw* **90**, 167–178 (2014)
5. Mitrović, D., Ivanović, M., Vidaković, M., Budimac, Z.: The Siebog multiagent middleware. *Knowl.-Based Syst.* **103**, 56–59 (2016)
6. Vidaković, M., Ivanović, M., Mitrović, D., Budimac, Z.: Extensible java EE-based agent framework—past, present, future. In: Ganzha, M., Jain, L.C. (eds.) *Multiagent Systems and Applications*, Intelligent Systems Reference Library, vol. 45, pp. 55–88. Springer, Berlin (2013)
7. Xanthopoulos, S., Xinogalos, S.: A comparative analysis of cross-platform development approaches for mobile applications. In: Proceedings of the 6th Balkan Conference in Informatics (BCI), pp. 213–220. ACM, New York (2013)
8. Bordini, R.H., Braubach, L., Dastani, M., El, A., Seghrouchni, F., Gomez-sanz, J.J., Leite, J., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. *Informatica* **30**, 33–44 (2006)
9. Bădică, C., Budimac, Z., Burkhard, H.D., Ivanović, M.: Software agents: languages, tools, platforms. *Comput Sci Inf Syst ComSIS* **8**(2), 255–298 (2011)
10. Bellifemine, F., Caire, G., Greenwood, D.: *Developing multi-agent systems with JADE*. Wiley, New York (2007)
11. Minotti, M., Santi, A., Ricci, A.: Developing web client applications with JaCaWeb. In: Omicini, A., Viroli, M. (eds.) *Proceedings of the 11th WOA 2010 workshop, Dagli Oggetti Agli Agenti*, Rimini, Italy, September 5–7, 2010. CEUR Workshop Proceedings, vol. 621. CEUR-WS.org (2010)
12. HTML5: a vocabulary and associated APIs for HTML and XHTML (2014). <http://www.w3.org/TR/html5/>. Accessed 29 April 2016
13. Gartner identifies the top 10 strategic technology trends for 2014 (2013). <http://www.gartner.com/newsroom/id/2603623>. Accessed 29 April 2016
14. Agent Developing Framework Homepage. <http://adf.sourceforge.net/index.html>. Accessed 29 April 2016
15. Voyager Homepage. <http://www.recursionsw.com/voyager-intro/>. Accessed 29 April 2016
16. FIPA Homepage. <http://www.fipa.org>. Accessed 29 April 2016
17. Badica, C., Ilie, S., Muscar, A., Badica, A., Sandu, L., Sboru, R., Ganzha, M., Paprzycki, M.: Distributed agent-based online auction system. *Comput Inf* **33**(3), 518–552 (2014)
18. Such, J.M., Alberola, J.M., Mulet, L., Espinosa, A., Garcia-Fornes, A., Botti, V.: Large-scale multiagent platform benchmarks. In: *Proceedings of the MultiAgent Logics, Languages, and Organisations—Federated Workshops, Languages, Methodologies and Development Tools for Multi-agent Systems (LADS)*, pp. 192–204 (2007)
19. Jurasovic, K., Jezic, G., Kusek, M.: A performance analysis of multi-agent systems. *Int Trans Syst Sci Appl* **1**(4), 335–342 (2006)
20. Ilie, S., Bădică, A., Bădică, C.: Distributed agent-based ant colony optimization for solving traveling salesman problem on a partitioned map. In: *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, pp. 23:1–23:9. WIMS '11, ACM, New York (2011)
21. Aiello, F., Fortino, G., Gravina, R., Guerrieri, A.: A java-based agent platform for programming wireless sensor networks. *Comput J* **54**(3), 439–454 (2011)
22. Fortino, G., Guerrieri, A., Russo, W.: Agent-oriented smart objects development. In: *16th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 907–912 (2012)
23. Verma, P., Gupta, M., Bhattacharya, T., Das, P.K.: Improving services using mobile agents-based IoT in a smart city. In: *International Conference on Contemporary Computing and Informatics (IC3I)*, pp. 107–111 (2014)
24. Jarvenpaa, L., Lintinen, M., Mattila, A.L., Mikkonen, T., Systa, K., Voutilainen, J.P.: Mobile agents for the internet of things. In: *17th International Conference on System Theory, Control and Computing (ICSTCC)*, pp. 763–767 (2013)
25. Systa, K., Mikkonen, T., Jarvenpaa, L.: Html5 agents: mobile agents for the web. In: Krempels, K.H., Stocker, A. (eds.) *Web Information Systems and Technologies, Lecture Notes in Business Information Processing*, vol. 189, pp. 53–67. Springer, Berlin (2014)
26. WildFly Homepage. <http://wildfly.org/>. Accessed 29 April 2016
27. Mitrović, D., Ivanović, M., Vidaković, M., Budimac, Z.: A scalable distributed architecture for web-based software agents. In: *7th International Conference on Computational Collective Intelligence (ICCCI)*, pp. 67–76 (2015)

28. Ilie, S., Bădică, C.: Multi-agent approach to distributed ant colony optimization. *Sci. Comput. Program.* **78**(6), 762–774 (2013)
29. Cao, J., Spooner, D.P., Jarvis, S.A., Nudd, G.R.: Grid load balancing using intelligent agents. *Future Gener. Comput. Syst.* **21**(1), 135–149 (2005)
30. Nehra, N., Patel, R.: Towards dynamic load balancing in heterogeneous cluster using mobile agent. *Int. Conf. Comput. Intell. Multimed. Appl.* **1**, 15–21 (2007)
31. Zhang, Z., Zhang, X.: A load balancing mechanism based on ant colony and complex network theory in open cloud computing federation. In: 2nd International Conference on Industrial Mechatronics and Automation (ICIMA), vol. 2, pp 240–243 (2010)
32. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon’s highly available key-value store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. SOSP ’07 (2007)
33. Mitrović, D., Ivanović, M., Vidaković, M., Budimac, Z.: Extensible Java EE-based agent framework in clustered environments. In: Mueller, J., Weyrich, M., Bazzan, A.L.C. (eds.) 12th German Conference on Multiagent System Technologies. Lecture Notes in Computer Science, vol. 8732, pp. 202–215. Springer, Berlin (2014)
34. Pérez-Carro, P., Grimaldo, F., Lozano, M., Orduña, J.M.: Characterization of the Jason multi-agent platform on multicore processors. *Sci Program* **22**(1), 21–35 (2014)
35. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC), pp. 143–154. ACM, New York (2010)