

Development of multi-agent framework in JavaScript

Aleksandar Lukić*, Nikola Luburić*, Milan Vidaković*, Marko Hölbl**

*Faculty of Technical Sciences, Novi Sad, Serbia

**Institute of Informatics, Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia
 {lukic.aleksandar, nikola.luburic, minja}@uns.ac.rs
 marko.holbl@um.si

Abstract—Large scale and complex systems that require significant hardware resources are typically designed to utilize distributed computing. This paper presents an architecture of a lightweight multi-agent middleware, aimed to simplify distributed computing operations. This middleware is implemented in JavaScript programming language, specifically in the Node.js framework. It is designed to support simple load balancing, avoid a single point of failure and enable execution of computationally intensive tasks. Furthermore, by supporting standardized agent communication, our agents can transparently interact with agents in existing, third-party multi-agent solutions.

Keywords—Multi-agent platform, Software agents, Mobile agents, Distributed architecture, JavaScript

I. INTRODUCTION

Modern applications and systems have high computational complexity and require significant hardware resources to run efficiently. Applications which require the most computational power utilize distributed computing, where machines and devices are grouped and managed in clusters, in order to distribute the workload. The main challenges involved in the design and implementation of distributed systems are hardware and software heterogeneity, resource access and availability, scalability, fault tolerance, concurrency, security, etc [1, 2]. The problem addressed in this paper is the development of distributed system based on the agent paradigm, which can easily distribute agents to a wide variety of devices.

The basic motivation for our work is to simplify distributed computing operations, where the underlying distributed system consists of heterogeneous software and hardware. We do this by developing a multi-agent middleware in JavaScript programming language. The JavaScript was chosen because it is the language of browsers, which means that agents that were written for our middleware can easily be executed in every major browser. This also means that our agents can be executed on any device that can open a Web browser, without any previous installation or configuration.

On the back-end, our system uses Node.js [3] application servers, and because of this design choice we've simplified the implementation of one of the most important mechanisms in the agent paradigm, the agent migration. Our agents can easily migrate between servers and between a browser and a server. This paper focuses on the server side of the whole system, which is designed to enable execution of computationally intensive tasks.

Furthermore, our system supports standardized agent communication defined by The Foundation for Intelligent Physical Agents (FIPA) [4]. This standardized communication is defined by the Agent Communication Language (ACL). By creating FIPA ACL-compliant agent middleware we've enabled communication between our agents and agents from different systems that support the same standard.

The rest of the paper is organized as follows. First, we give a brief overview of the related work in section 2. Our solution is presented in section 3. In section 4. we present some performance tests that we've done and made some comparisons. We discuss the current state of our work and our goals for the future in section 5. Finally, in section 7., we draw our conclusions.

II. RELATED WORK

Use of Web, JavaScript, and HTML as an agent platform is not very common. The Radigost [5, 6] system uses Web and JavaScript as an implementation platform for multi-agent systems. It has many interesting features like support for standard agent communication and yellow-pages service for agent directories. However, it does not support running agents outside the browser. Because of that, Radigost has been merged with the JavaEE-based agent framework into one system named Siebog [7, 8]. Siebog allows execution of JavaScript agents on the server side with the use of adapters, but only one-way mobility is currently supported. Agents can only migrate from a browser to a server.

The framework described in our paper follow up previous research and development done in Siebog. Because the server-side of Siebog is based on JavaEE technologies, it cannot be hosted on small devices, such as those used in Internet of Things (IoT) applications. Taking that, agent migration limits, and agent distribution mentioned above into consideration, we have decided to create a lightweight Node.js based middleware. This middleware, conveniently named SiebogJS, currently doesn't support all functionality that Siebog does. Use of enterprise technologies and industry standard servers enabled Siebog to support automatic load balancing, agent's state persistence and replication, and fault-tolerance.

Alongside Radigost, currently, there exist a few more additional Web, more specifically HTML5 based agent middlewares. Most similar to ours is described in [9]–[11]. The

middleware proposed in this paper is primarily focused on using mobile agents to support the IoT platform. This middleware enables agent execution in the browser, and on a server. Browser agents have JavaScript part and HTML part, while server versions of those agents are pure JavaScript objects. From their description, it is not clear how multiple agents could be started within the same host, nor if server-side agents could execute computationally intensive tasks, which is one of the main goals we wanted to achieve.

Another Web-based agent framework is described in [12]. This paper describes agent middleware that supports document based client-side and server-side agents. The system model is based on two components. Pneuma, the document that encapsulates data together with an execution state and logic, the agent. The second component is named Soma, and it is the execution environment that is available on Web servers and Web browsers. In this approach, Soma hides the differences between browser and server environments and creates a completely new application platform for mobile agents. These two components are based on standard Web technologies and protocols like URIs, HTTP, as well as JavaScript.

III. SOLUTION

The topology of our middleware consists of a set of interconnected hosts, which represent agent centers, as depicted in Fig. 1. The minimum number of hosts in the set is one. One of the hosts is defined as a master, while others are defined as slaves. The master host manages the state of the whole cluster in order to preserve consistency. To avoid a single point of failure, one of the connected slave hosts is set to become the next master when the current one crashes. The master host also receives and load balance client requests in order to distribute the load.

Every agent center can send messages directly to another agent center or it can publish messages to all agent centers at once. Our current solution uses ZeroMQ [13], asynchronous messaging library, to handle agent center interconnection.

Fig. 1 also shows that the clients can connect to the master agent center via HTTP and/or via WebSockets protocols.

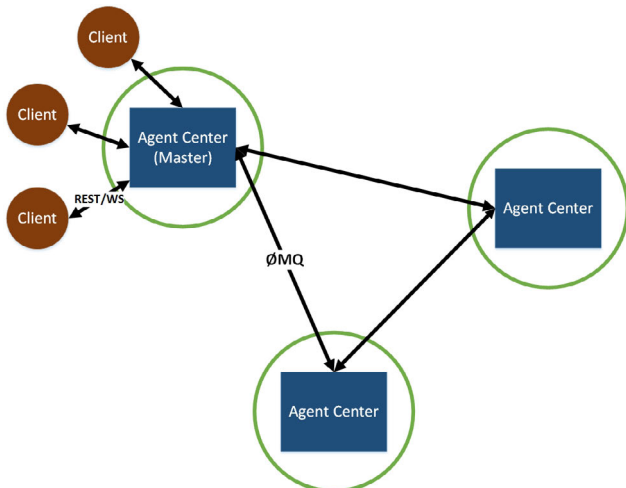


Fig. 1: SiebogJS Architecture

A. Agent center architecture

Every agent center has a layered architecture, which is depicted in Fig. 2. Each layer relies on the functionality of the previous layer.

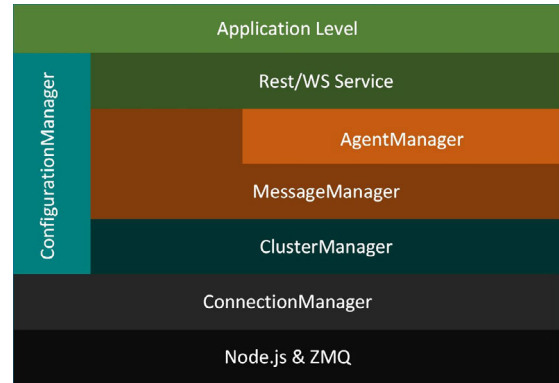


Fig. 2: Layered architecture of the agent center.

The first layer represents the technologies that we've used to implement our system.

The connection manager represents an abstraction layer that hides concrete network protocols used for connection establishment. As mentioned above, our system currently relies on ZeroMQ for connection establishment and message routing, but for example, it can easily be replaced by WebSockets.

The cluster manager is used to create a specific topology. The main functions that it handles are endpoint binding, host registration, and connection management. When the agent center starts, this module uses the connection manager to bind two endpoints, one for the peer-to-peer communication pattern and one for the publish-subscribe pattern. This means that through one endpoint agent center sends messages directly to the other agent centers, and through the other, it sends messages to all agent centers at once. In the current solution, every slave agent center on its start connects to the master and sends its data in order to register to the cluster. Master then notifies other slaves, so that they can also connect to the new slave.

The cluster manager also uses heartbeat mechanism to detect connection breaks, so that it can take appropriate measures for keeping the cluster consistent. Also, because this module is on top of the connection manager, every message goes through it, and those that were not intended for it will get passed to the message manager.

The message manager handles message routing. Messages can be passed between clients, agents and other hosts. This component is central in our design because it connects the REST/WebSocket services, the agent manager, and the cluster manager.

The agent lifecycle, messaging and migration is handled by the agent manager. Every agent gets executed in a new Node.js child process. The reason for this design choice is the single-threaded nature of JavaScript and Node.js. In order to handle concurrent request agent manager needed to start agents in at least one child process, because long agent execution can block request handling. With the presupposition that every agent

will process received messages fast, multiple agents could be executed inside one process. But because we can't guarantee that this will be the case and also because we wanted to enable execution of computationally intensive tasks, every agent had to be executed inside a new Node.js child process. Drawbacks of this design choice will be discussed in the next section. Because agents are executed in a new process, agent manager had to handle inter-process communication. Furthermore, there is no direct connection from the message manager to the agent, so all agent messages first go to the agent manager, and then get passed to the agents.

The agent migration is a very important mechanism in the agent paradigm, and it is one of the main reasons we have chosen JavaScript in the first place. Our system offers this mechanism to agents, and the agent itself can initiate the migration process. In the current solution, agent's code gets uploaded to the master host and when the upload finishes it gets distributed to all other hosts. Because every agent center has a code of every agent, in the migration process it is only necessary to send the agent's state to the targeted agent center.

Fig. 2, shows that some layers are configurable. The configuration manager uses configuration data received from JSON files to setup appropriate functionality.

REST and WebSocket (WS) services provide the endpoints of our system to which the client applications can connect. These services provide endpoints for creating and removing agents, sending messages to agents etc. The only difference between endpoints they provide is that WS service has one additional endpoint where the client can get the agent activity log. This endpoint exposure enables our system to support the Web, mobile or desktop based client applications.

B. Programming agents

From a developers point of view, system functionality is encapsulated in reusable Agent class. Every agent has a unique identifier represented by the AID class, Listing 1. The AID consists of the agent's runtime name, the id of the host on which agent will be executed and the agent class. The value property of AID class is just its string representation.

```
function AID(name, host, agentClass) {
  this.name = name;
  this.host = host;
  this.agentClass = agentClass;
  this.value = name + divider + host +
    divider + (agentClass.value ||
    agentClass);
}
```

Listing 1: AID class

Besides a unique identifier, every agent has a type, which is represented by the AgentClass class, Listing 2.

```
function AgentClass(name, module) {
  this.name = name;
  this.module = module;
  this.value = module + divider + name;
}
```

Listing 2: AgentClass class

The AgentClass consists of the type name, and the module name. The value property is just its string representation.

The agent also has two other properties that are injected into it upon its creation. These two properties are agent manager and message manager objects. The agent manager object provides methods for getting information about currently available agent types in the system, as well as information about running agents. Besides that, agent manager object exposes the migration method, which agent can invoke and start the migration process. On the other side, message manager exposes methods for sending messages.

In order to create distributed applications, developers only need to create their own specific agents, subclasses of Agent class. The specific subclass can override the following methods:

- `handleMessage(ACLMessage)`, this method is automatically called whenever agent center receives a message addressed to that agent. As a parameter agent gets the received ACL message.
- `postConstruct()` and `preDestroy()`, these methods are automatically called after the agent is created, and before the agent is removed from the system, respectively. Some initialization can be done within a `postConstruct` method, while used resources could be released in the `preDestroy` method.
- `postMigration(ACLMessage)`, when the migration process finishes this method gets called instead of `postConstruct`. ACL message is the message sent via `migrate` method. The `migrate(hostAlias, ACLMessage)` method initiates the migration process, it can be invoked by an agent itself. This method accepts two parameters, alias of the host on which agent should move and an optional ACL message which can be used to pass additional information.

C. Example of an agent

In the Listing 3, we gave a simple agent example. This agent has overridden the `handleMessage`, `postConstruct`, and `preDestroy` methods. In the last two methods, the agent just logs to the console that it has started and that it will be destroyed. In the `handleMessage` method, the agent asks the agent manager for the identifier of an agent of Ping type, and if there is at least one Ping agent running in the system this agent will send the message to it with the message managers `post` method.

```
function TestAgent() {
  var self = this;
  Agent.call(self);

  self.handleMessage = function (aclmsg) {
    console.info('Agent with aid: ' + self.
      aid.value + ' received a message: '
      + JSON.stringify(aclmsg));

    self.agentManager.getAgent('siebogjs.
      pingpong@ping', false, function (
      pingAid) {

      if (!pingAid)
```

```

return;

var msg = ACLMessage(self.aid.
    value, ACLPerformatives.
    REQUEST, [pingAid]);
msg.content = 'Ping';
self.messageManager.post(msg);
};

self.postConstruct = function () {
    console.info('Created agent with aid:
        '+ self.aid.value );
};

self.preDestroy = function () {
    console.info('Destroying agent with
        aid: '+ self.aid.value);
};
}

```

Listing 3: Test agent example

IV. EVALUATION

During development, we have performed a couple of performance tests and compared the results of Siebog and our solution. The test environment consisted of a laptop computer with a quad-core Intel i7 CPU with hyper-threading, 16GB of RAM and an SSD. Besides that, Siebog was executed on Wildfly 8 [14] server, while our solution was executed on Node.js v6.10.0 application server.

Two performance tests were taken. First, we tested the limit of the number of running agents and also measured the time needed to create a certain number of agents. The rate of change of time given the number of created agents is depicted in Fig. 3.

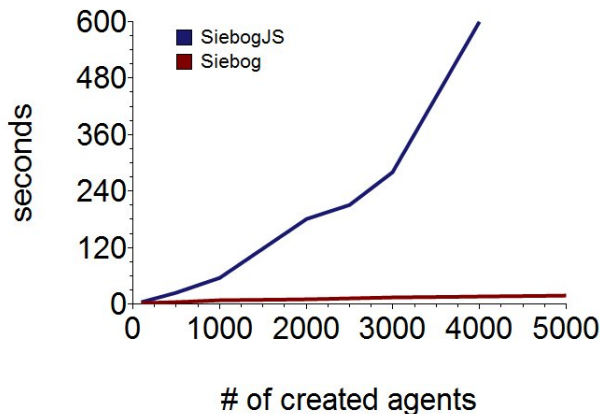


Fig. 3: Creation of agents

It is obvious that our solution performs much worse than Siebog. The reason behind this behavior is the fact that every agent is executed in a new Node.js child process, while Siebog agents are executed in threads. For the same reason, our agents consume much more hardware resources. The maximum number of running agents and the time needed for their execution can be observed in Fig. 3. The limit of our system is about

4000 agents, while creation and execution of them took about 10 minutes. On the other hand, Siebog can execute more than 15K agents, and for example, the creation and execution of 4K agents took about 20 seconds. The important thing to mention is, when both systems were executing, for example, 2K agents, in the case of Siebog the laptop was usable, but that can't be said for the our solution.

In the second test, we measured the time that two agents took to exchange a certain number of messages. The results of these measurements are depicted in Fig. 4.

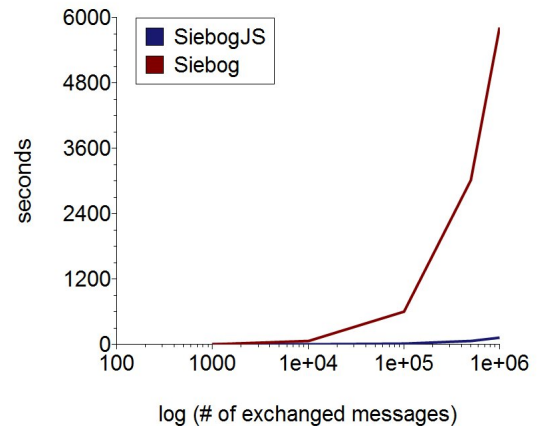


Fig. 4: Message exchange between two agents

The x - axis is presented using a logarithmic scale because we measured the exchange of messages in the range from 1K to 1M. Here we can see that our solution performs much better than Siebog. For example, exchange of 1M messages took our system about 2 minutes, while Siebog took about an hour and a half. The slow message exchange process is a known Siebog issue, and it could and should surely be optimized in the future.

V. DISCUSSION AND FUTURE WORK

In the previous section, we have shown strengths and weaknesses of our system. We have shown that the biggest drawback of our design is the fact that every agent gets executed in a new process. This can be modified to work similarly like JaveEE-based servers handle session beans. Instead of thread pool that is used in Java, we could create a pool of processes. Then multiple agents should be executed inside one process. To enable this execution model passivization of inactive agents and activation of passivated agents should be implemented in our system. This would enable our system to dynamically allocate processes, and execute multiple agents inside one process, which should reduce resource consumption.

Currently, our system only solves a single point of failure problem, but if one of the agent centers crashes, all of its agents will be gone. In the future, we would like to add agent state persistence and replication which would enable high availability and fault-tolerance. Furthermore, we would like to add a new type of agents and persist them only.

Load balancing is another topic that we would like to address in the future, currently, our system only supports

round-robin load balancing, but we would like to add load balancing that depends on the load of the agent center.

Our system currently doesn't support any security mechanisms. This is a big issue because we support agent upload, so malicious code could easily be executed on our servers. Since Node.js does not have built-in security mechanisms, we will need to implement our own. One of our primary goals is to prevent security problems while agents execute their code, and we will do this by implementing two security mechanisms: static code analysis and run-time security based on policies. This will enable us to potentially detect problematic code and to prevent agent to execute that code. Furthermore, Node.js depends on multiple modules which are used by the code that is executed on it. The way that a module is required and used could pose a security threat. Besides that, JavaScript has an eval function, that can be used to parse and evaluate a string as an expression. This can also pose a security threat, and so it should be detected and discarded with the static code analysis.

Previously we have only talked about how to improve our current solution, but in order to complete the whole system, we also need to implement the client-side. Only with the completed server-side and client-side parts, we can fully support agent distribution mentioned in section I. The client-side will be very similar to the Radigost, for example, it would probably use WebSockets to communicate with the server and WebWorkers to run agents.

VI. CONCLUSION

In this paper, we have presented our server-side JavaScript-based agent middleware. Also, we have presented our solution's test results and compared them to the Siebog's. Those tests have shown strengths and weaknesses of our design. In the previous section, we have suggested some techniques that can be used to solve the main performance problems. We have also discussed our future goals involving security and more advanced clustering techniques, as well as the future development of the middleware's client-side part. In order to support the distribution of agents without previous installation or configuration, the client-side part must be implemented. The server-side part can run on multiple platforms, but it requires installation.

Like we mentioned before Web and JavaScript based agent middlewares are rare, and there exists a lot of room for future research and improvements.

ACKNOWLEDGMENT

Results presented in this paper are part of the research conducted within the Grant No. III-47003, Ministry of Education, Science and Technological Development of the Republic of Serbia.

REFERENCES

- [1] Andrew S. Tanenbaum and Marteen van Steen. *Distributed Systems. Principles and Paradigms*. Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2007.
- [2] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems. Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.
- [3] Node.js foundation. Node.js. <https://nodejs.org/en/>.
- [4] Foundation for intelligent physical agents. <http://www.fipa.org/>.
- [5] Dejan Mitrović, Mirjana Ivanović, Zoran Budimac, and Milan Vidaković. Radigost: Interoperable web-based multi-agent platform. *Journal of Systems and Software*, 90:167–178, 2014.
- [6] Dejan Mitrović, Mirjana Ivanović, and Costin Bădică. Delivering the multiagent technology to end-users through the web. In *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14)*, page 54. ACM, 2014.
- [7] Dejan Mitrović, Mirjana Ivanović, Milan Vidaković, and Zoran Budimac. Extensible java ee-based agent framework in clustered environments. In *German Conference on Multiagent System Technologies*, pages 202–215. Springer, 2014.
- [8] Milan Vidaković, Mirjana Ivanović, Dejan Mitrović, and Zoran Budimac. Extensible java ee-based agent framework—past, present, future. In *Multiagent Systems and Applications*, pages 55–88. Springer, 2013.
- [9] Jari-Pekka Voutilainen, Anna-Liisa Mattila, Kari Systä, and Tommi Mikkonen. Html5-based mobile agents for web-of-things. *Informatica*, 40(1):43, 2016.
- [10] Laura Jarvenpaa, Markku Lintinen, Anna-Liisa Mattila, Tommi Mikkonen, Kari Systs, and Jari-Pekka Voutilainen. Mobile agents for the internet of things. In *System Theory, Control and Computing (ICSTCC), 2013 17th International Conference*, pages 763–767. IEEE, 2013.
- [11] Kari Systä, Tommi Mikkonen, and Laura Järvenpää. Html5 agents: mobile agents for the web. In *International Conference on Web Information Systems and Technologies*, pages 53–67. Springer, 2013.
- [12] Marius Feldmann. An approach for using the web as a mobile agent infrastructure. In *Proceedings of the International Multiconference on ISSN*, volume 1896, page 7094, 2007.
- [13] CEO of iMatix Pieter Hintjens. Ømq - the guide. <http://zguide.zeromq.org/page:all>.
- [14] Rad Hat. Wildfly homepage. <http://wildfly.org/>.