UDC 519.718.7:681.3.069

S. S. Surkov,
O. N. Martynyuk, PhD.

IMPROVEMENT OF SECURITY FOR WEB SERVICES BY RESEARCH AND DEVELOPMENT OF OAUTH SERVER

Abstract. *With the crucial growth of information-technologies is rapidly growing sociability. The clearest example - there are many social networks. To solve the problem of authorization for third-party web services use OAuth protocol, which asks user to enter his credentials in Web-browser or native application and as a result gives to application access token is used for authorization with REST-API to identify user. All social networks such as Facebook, Twitter, Google+, Foursquare, Evernote, VK provide REST-APIs protected by OAuth 1.0a or OAuth 2.0.*

For implementation of web services developers use custom frameworks. In this work, we researched, developed and analyzed OAuth library for creation of web-services built on JAX-RS 2.0 framework.

For verification and comparison our solution with others we plan test all the solutions by test suite which allows to achieve high number of concurrent connections. To reduce workload on test server we are going to launch test suite on a different PC. After achieving the results, we will compare them to existing solutions and analyze our solution for weak points.

Keywords**:** *Server, Java, OAuth, REST API, Load Testing*

С. С. Сурков,
А. Н. Мартынюк, канд. техн. наук

ПОВЫШЕНИЕ ЗАЩИЩЕННОСТИ ВЕБ-СЕРВИСОВ ПУТЕМ ИССЛЕДОВАНИЯ И РАЗРАБОТКИ OAUTH СЕРВЕРА

Аннотация. *Веб-сервисы без какой-либо защиты REST API очень уязвимы для многих видов сетевых атак. Для того, чтобы надежно защитить веб-сервис от сетевых атак был создан протокол OAuth. Эта статья описывает наше исследование и разработку серверной библиотеки OAuth, построенной на фреймворке JAX-RS 2.0 и нагрузочное тестирование вновь созданной библиотеки и других существующих решений.*

Ключевые слова: *сервер, Java, OAuth, REST API, нагрузочное тестирование*

С. С. Сурков,
О. М. Мартинюк, канд. техн. наук

ПІДВИЩЕННЯ ЗАХИЩЕНОСТІ ВЕБ-СЕРВІСІВ ШЛЯХОМ ДОСЛІДЖЕННЯ І РОЗРОБКИ OAUTH СЕРВЕРА

Аннотація. Веб-сервіси без будь-якого захисту REST API дуже уразливі для багатьох видів мережевих атак. Для того, щоб надійно захистити веб-сервіс від мережевих атак був створений протокол OAuth. Ця стаття описує наше дослідження і розробку серверної бібліотеки OAuth, побудованої на фреймворку JAX-RS 2.0 і навантажувальне тестування новоствореної бібліотеки та інших існуючих рішень.

Ключові слова: *Сервер, Java, OAuth, REST API, Тестування навантаження*

Introduction. With the crucial growth of information-technologies is rapidly growing sociability. To solve the problem of authorization social networks OAuth protocol is used, which is an open authorization protocol that allows third party applications to provide limited access to protected resources without transferring to server username and password. For example, a social network developer who wants to provide an access to user's friend list to third party developer is not required to share with third party developer email and password.

Instead, user authorizes directly to the social network, which (with the permission of the user or administrator of the web-service) provides permissions to the friend list. All social networks such as Facebook, Twitter, Google+, Foursquare, Evernote, and VK provide REST-APIs protected by OAuth 1.0a or OAuth 2.0.

Analysis of existing solutions of OAuth server implementations showed that for HTTP POST request method request body is not used in the calculation of OAuth signature. This vulnerability allows modifying the content of the request during transmission. On the other hand, many third party libraries are not open-source and many have licensing problems for commercial use. Furthermore, there's not much Java libraries which wouldn't require any specific framework and bring a lot of dependencies and have a good performance.

This paper is a further development of our previous article [1] and fully implements proposed in that work solution. In [2] we've proposed a solution which would require browser on smartphone side. As this library supports XAuth communication smartphone approach would be no longer required for developers who would develop client and server side using proposed library in this work. This library is fully compatible for migration from single server to server cluster [3 – 4] approach what we created.

Aim of the work: increase security of web services. To achieve this aim we investigated plenty of authorization protocols and found protocol OAuth 1.0a as the most secure. In work for modification of OAuth protocol [2] we significantly increased security of OAuth 1.0a protocol.

In this paper we improve security of OAuth protocol by developing library which would be easily implemented by web service developers. To achieve this aim we develop a feasible library, which implements OAuth protocol for JAX-RS 2.0 framework, which should be: easy to be modified, easily to be integrated in project, ensure compatibility with protocols OAuth 1.0a XAuth. The library allows creating secured REST API, which are used for creation of dynamic web sites and desktop or mobile applications.

The main advantage of OAuth [5 – 6] protocol is that username and password are sent to server from web-browser window and are not exposed to third party app. To enter username and password web browser window or native mobile application is used to ensure that third-party developers would not have unauthorized access to user's credentials especially when web service provides REST-API to public. For trusted applications, web service developer could allow using xAuth protocol to pass directly login and password from user's authentication and this will significantly easier for application developer and still secure. But in this case there's a potential possibility that third party application developer could collect user's credentials.

This gives user more reasons to trust the application, as user can be sure that his credentials wouldn't be stolen from web browser window by third party apps. In modern mobile apps to get access token instead of browser window could be used native mobile applications. In this case application which use Facebook API first try to use official Facebook app tries to open native application first and then if there's no application then browser window is opened.

Responsibility to trust applications, which use xAuth, depends on how much web service developer trusts third party app developer. After successful authentication in web browser third-party app retrieves access token with is meant to be used for REST API requests.

If user is already authorized in web-service, he doesn't need to enter his credentials the second time in order to user the app. Necessity of authorization depends on access token lifetime. If access token is about to expire then application refreshes this token to continue work with REST service. In this case authorization will be required if user didn't use the app for access lifetime.

For development of OAuth[5,6] protected web-services, typically are used two or three legged approach. The main difference between them is that in two-legged implementation does not involve user and only application is authorized. In this case, if the user wants to access the data by Twitter, it uses a three-legged server as access token needs to be requested for user in third party application, instead of application token for Twitter application. The library will provide both two and three-legged server, since it is more practical for everyday use [7].

The sequence of OAuth protocol includes the following steps:

• consumer requests a token from the server;

• consumer then is redirected to authorization page;

• consumer logs in and is redirected back to the consumer with access token;

• consumer receives an access token, and requests OAuth token, which will be used in future to make secured REST API requests [8].

General structure. We created OAuth server library for wide used JAX-RS framework. To integrate the library with third application interface OAuthProvider which describes work with database needs to be implemented.

In test application, we use Grizzly application server, MySQL and JDBC in implementation of each API and OAuthProvider interface. To work with MySQL server library Apache DBCP is used, which manages a pool of connections MySQL and manages release of the connection.

The library supports plain text, HMAC-SHA1 and RSA-SHA1 signatures of request. To sign, verify or generate OAuth request class OAuthSignature is used.

The class diagram of implementation of OAuth signatures is shown in Fig. 1.

This web container accepts HTTP[9] connections and processes them in its own thread pool. After registered method according to its annotation by JAX-RS is called.

To register a method annotation should be added which describe HTTP method such as GET, POST, PUT, etc. After this annotation for API PATH, should be also added e.g. @Path("/path/to/api").

To access the database for test application software design pattern DAO (Data Access Object) is used. To use this software design pattern in JAX-RS there are DAO factory instances.

DAP is injected into class when annotation is present. After JAX-RS finds annotation in class then DAO factory matches class type and provides DAO object.

To make the library truly easily used with JAX-RS framework annotation @OAuthImplementationRequired was implemented and registered in JAX-RS.

With it web service developer indicates REST APIs which are meant to be secured with OAuth protocol.

Implementation of signature of request for OAuth server. A particularly important part of OAuth 1.0a implementation is signature of request. We implemented OAuth signature by specification in three algorithms: RSA SHA1, HMAC SHA1 and plain text. The last option is not safe as an attacker might capture keys.

However, when using SSL 2.0 encryption and certificate validation on client side, from performance point of view this option is the best because it takes less computing resources.

The main class OAuthSignature allows to third party developer to sign and verify the OAuth requests. It can work with any signature classes, which that implement OAuthSignatureMethod interface that requires the implementation of three methods: name, sign, verify.

Objects of these classes are created in the factory Methods, which guarantees they will be created only once.

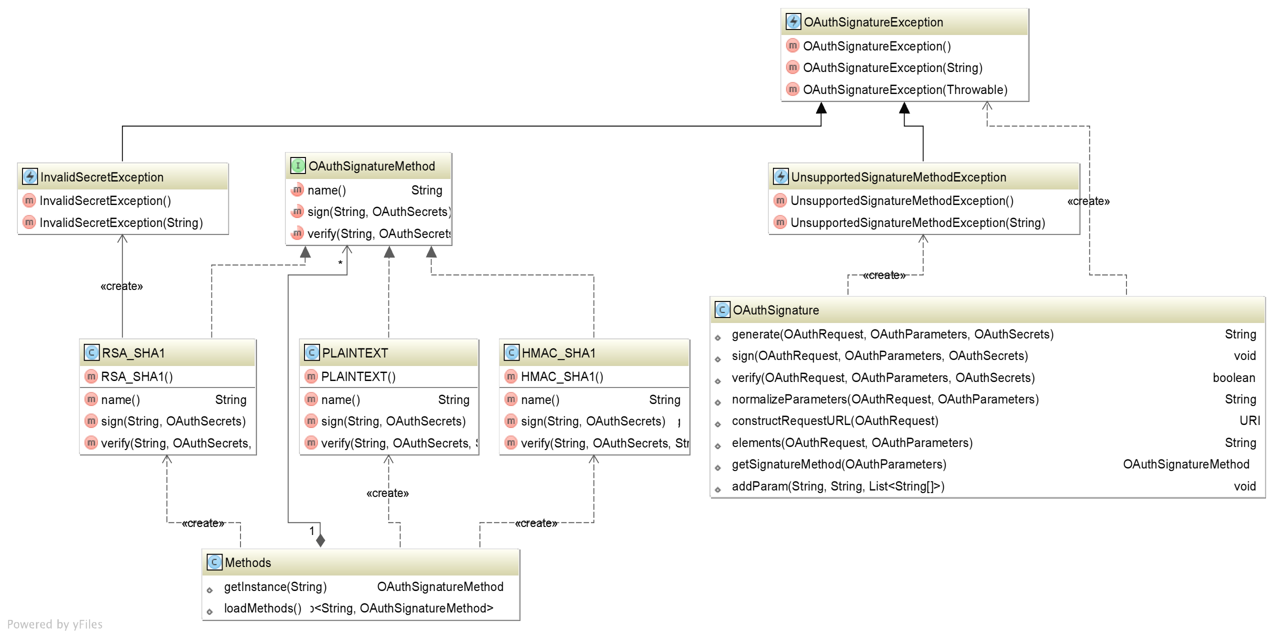

OAuthSignatureException
OAuthSignatureException()
OAuthSignatureException(String)
OAuthSignatureException(Throwable)
InvalidSecretException
InvalidSecretException()
InvalidSecretException(String)
OAuthSignatureMethod
name() String
sign(String, OAuthSecrets)
verify(String, OAuthSecrets
UnsupportedSignatureMethodException
UnsupportedSignatureMethodException()
UnsupportedSignatureMethodException(String)
«create»
«create»
«create»
OAuthSignature
generate(OAuthRequest, OAuthParameters, OAuthSecrets) String
sign(OAuthRequest, OAuthParameters, OAuthSecrets) void
verify(OAuthRequest, OAuthParameters, OAuthSecrets) boolean
normalizeParameters(OAuthRequest, OAuthParameters) String
constructRequestURL(OAuthRequest) URI
elements(OAuthRequest, OAuthParameters) String
getSignatureMethod(OAuthParameters) OAuthSignatureMethod
addParam(String, String, List<String[]>) void
RSA_SHA1
RSA_SHA1()
name() String
sign(String, OAuthSecrets)
verify(String, OAuthSecrets,
PLAINTEXT
PLAINTEXT()
name() String
sign(String, OAuthSecrets)
verify(String, OAuthSecrets,
HMAC_SHA1
HMAC_SHA1()
name() String
sign(String, OAuthSecrets)
verify(String, OAuthSecrets, Str
«create»
«create»
«create»
*
1
Methods
getInstance(String) OAuthSignatureMethod
loadMethods() ›<String, OAuthSignatureMethod>
Powered by yFiles

Fig. 1. Test Server UML diagram

Generation of Request Token. The first step of authorization is retrieving valid request token, which authorizes client application. To do this we developed API, which generates request token.

To get request token application validation is required the same way as in registration API except in this API no OAuth token is issued.

After successful verification, new request token is returned to user and kept in temporal database in server.

Third party application should open login page after receive request token. In case of successful authorization parameter oauth verifier will be received.

The difference between request token and oauth token is that request token authorizes application and oauth token authorizes user.

After OAuth verifier is received it's needs to be exchanged to oauth token and oauth token secret. After successful response is received user can make requests to web service.

On server side in database entry which associates user and OAuth token and OAuth token secret is created.

To exchange request token and OAuth verifier we created API which checks request token and verifier and issues oauth token.

The retrieved OAuth token and OAuth token secret user's application must store and use for future requests.

Implementation details of the validation request. Once the user OAuth token, every request which is protected by OAuth needs to be validated on server side.

The difference for verifying request token and access token is access token authorizes user and is taken from different database.

From the algorithm there are three important steps: verify that oauth token is valid, calculate signature string and hash of it and verify the request. Algorithm validation of request shown in Fig. 2.

On the first step the library simply looks into oauth tokens database and finds if this token is valid.

On the second step, if the token is valid then it constructs signature string with oauth token secret part from database.

In the third step hash of signature string with database token is compared to hash in request and if it doesn't match then not successful response is sent to client.

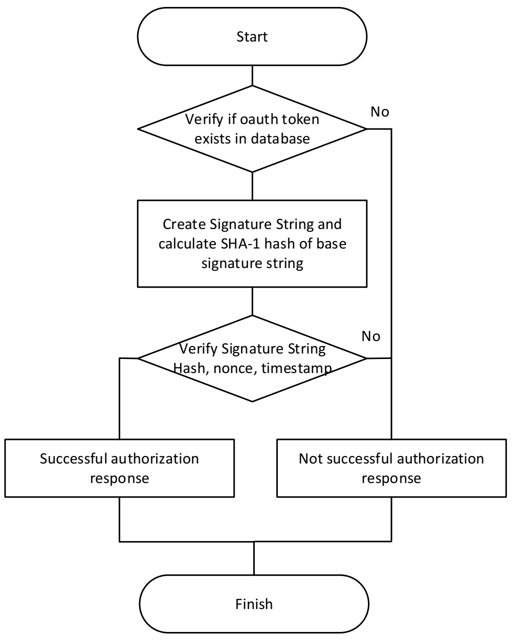

Start
Verify if oauth token exists in database
No
Create Signature String and calculate SHA-1 hash of base signature string
Verify Signature String Hash, nonce, timestamp
No
Successful authorization response
Not successful authorization response
Finish

Fig. 2. The algorithm of validation OAuth request

Testing OAuth library. To test performance and workload of the library we used Apache JMeter for REST API with OAuth authorization and without it.

To separate server and test application workload and get more precise results we ran Apache JMeter on different PC in the same local network. For test we used 50 concurrent threads.

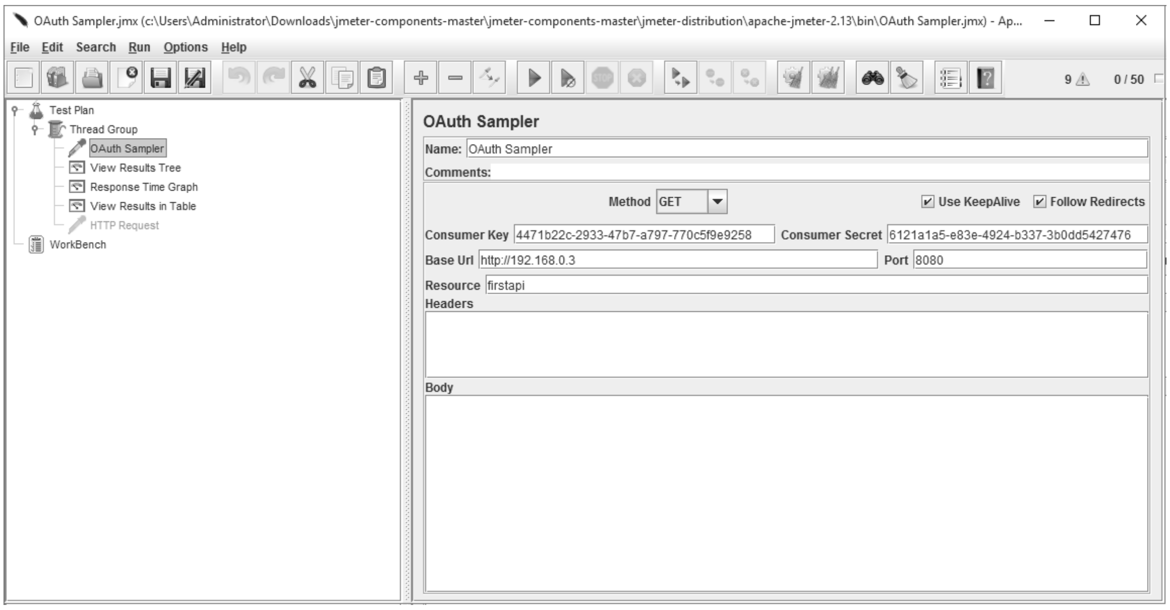
Configuration of Test PC and Server PC is shown in Table below.

Configuration of Test PC and Server PC

Parameter	Server PC	Test PC
CPU	Core i7-3770K (4C/8T)	Core i5-3570K (4C/4T)
RAM	16Gb	16Gb
Main Storage	SSD 256G	SSD 256G
OS	OS X 10.11 El Capitan	OpenSuse Linux 42.1

As client software we used Apache JMeter with OAuth 1.0a authentification plugin and during the each test get a table with processing time for every request request.

After the results are retrieved we analyze them by calculating averare processing and after that we create plot, In Fig. 3 configuration for API secured OAuth is shown.



OAuth Sampler.jmx (c:\Users\Administrator\Downloads\jmeter-components-master\jmeter-components-master\jmeter-distribution\apache-jmeter-2.13\bin\OAuth Sampler.jmx) - Ap...

OAuth Sampler

Name: OAuth Sampler

Comments:

Method GET

[✓] Use KeepAlive [✓] Follow Redirects

Consumer Key 4471b22c-2933-47b7-a797-770c5f9e9258 Consumer Secret 6121a1a5-e83e-4924-b337-3b0dd5427476

Base Url http://192.168.0.3 Port 8080

Resource firstapi

Headers

Body

Fig. 3. Configuration for OAuth request

As optimal amount of requests we choose 100000 requests based on average test time to show us full picture.

For OAuth protected API and not protected API we analyzed results and created plots.

For API which is not protected by OAuth average execution time are 2.391 milliseconds.

Plot that shows execution time for every request for not-protected API is shown in Fig. 4.

For protected API by OAuth average processing time is 4.946 milliseconds. This increasing of processing time is because of calculation of base signature string and comparing hashes. But security of REST API in this case is significantly improved.

Plot that shows execution time for every request for OAuth-protected API is shown in Fig. 5.

Analyzing the results, we created plot, which shows requests per minute for OAuth-protected request and not OAuth-protected plain request.

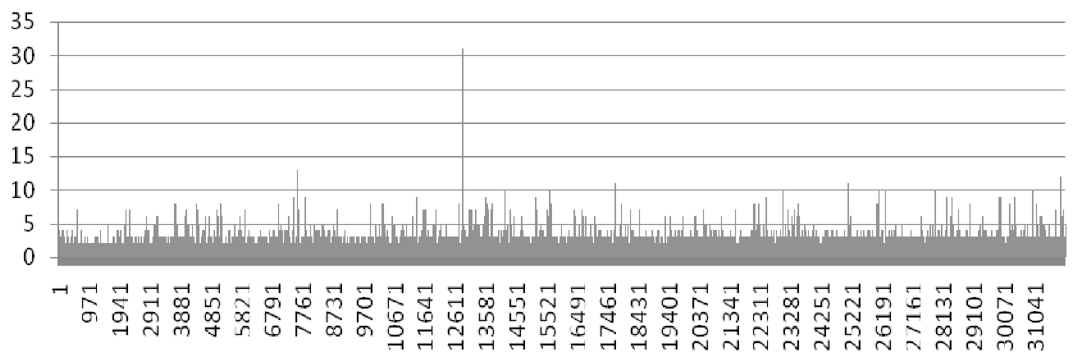


Fig. 4. Execution time for non-protected API

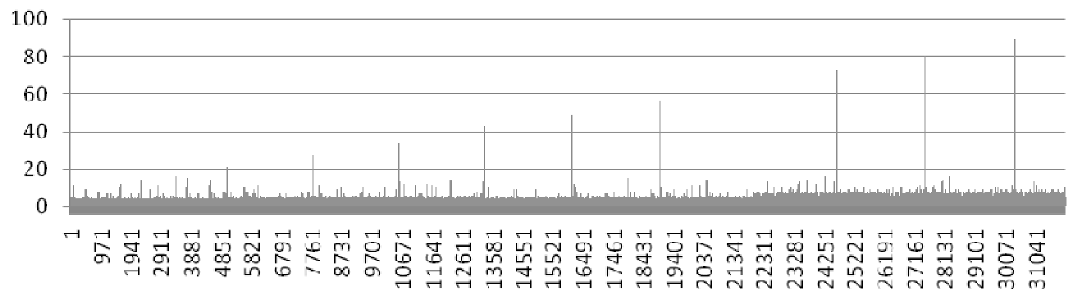


Fig. 5. Execution time for protected API

For OAuth-protected API average request per minute (rpm) is 8323 and for non-protected API is 11393 rpm.

The plot which shown requests per minute for protected and not-protected API in Fig. 6.

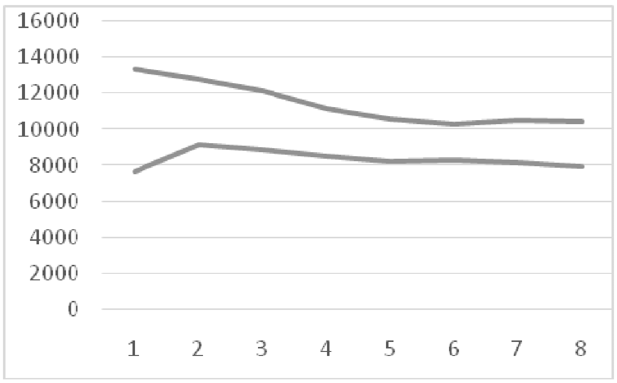


Fig. 6. Requests per minute for OAuth and plain request

Results tell us that to protect REST API by OAuth protocol requires more processing power. But in exchange web service developer guarantees high security of his REST API.

In comparison with other solutions which give us a bit worse result 7998RPM for protected API we achieved extra security, solved licensing problems. And we achieved a small speed boost because of the difference in implementation of the libraries which is 3,86 %.

Requests per minute are heavily depend on PC configuration as we changed roles of test PC and server and we got 6370RPM for Core i5 3570K at the same frequency 3.5GHz but it has 4 cores and 4 threads and does not support Hyper Threading as Core i7 3770K does, which has 4 cores and 8 threads (8 virtual cores). In our case having CPU with Hyper Threading technology with additional 4 virtual cores gave us 20 % boost for RPM.

To confirm that Hyper Threading gave us such boost we tested again with 3770K and with disabled hyper threading and results are identical to 3570K with the same frequency.

Analyzing the result, we can conclude that to choose the hardware to use our OAuth server library you need to clarify first how many requests per minute you need to achieve. Then you need to run the same tests as we did on your server and make sure it satisfies your requirement. In case you want to upgrade your server our recommendation is to choose CPU with many cores as possible to handle as many as possible concurrent requests.

If upgraded server still doesn't satisfy your expectations, then we recommend following our method of migration from single server to server cluster [3;10 –11].

Conclusion. The result of our research and development is convenient and easily integrated OAuth server library, which uses JAX-RS framework and fully implements OAuth protocol server side.

To integrate library to existing project developer only needs to implement OAuthProvider interface to work with database.

The library meets the aims set in the work: easily modifiable, easily integrable and supports protocols OAuth 1.0a and xAuth.

Testing showed that library can survive very high workloads but requires more processing power in comparison with plain non-protected request.

Comparing our library to existing solutions we achieved extra security, solved licensing problems and a speed boost of 3,86 %. Other than this the library showed pretty good optimization for multi core CPUs and gave 20 % boost for CPU with enabled Hyper-Threading.

In future, this library will be optimized for performance and memory. Other than this we'll add built-in support for migration of single server to server cluster and analyze the dependency of requests per minute to number of server in cluster.

References

1. Surkov S.S., Martynyuk O.M., and Mileiko I.G., (2015), Modification of Open Authorization Protocol for Verification of Request, *Electronic and Computer System, Special Edition,* No. 19 (95), Odessa, Ukraine, pp. 178 – 181.

2. Surkov S.S., and Martynyuk O.N. Avtomatizatsiya avtomobilnogo kompyutera bez podderjki brausera posredstvom Bluetooth [Authorization for Automobile Headunit without Browser Support with Mobile Devices through Bbluetooth], (2015), *Holodilnaya Tehnika I Tehnologiya*, No. 2, *Kviten,* Odessa, Ukraine, pp. 65 – 71 (In Russian).

3. Surkov S.S., and Martynyuk O.M., (2015), Method of Migration from Single Server

System to Server Cluster, *Proceedings of the 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications* (IDAACS'2015), 24-26 September 2015, Warsaw, Poland, pp. 808 – 811.

4. Surkov S.S., and O.N. Martynyuk, Avtomatizatsiya avtomobilnogo kompyutera bez podderjki brausera posredstvom Bluetooth [Authorization for Automobile Headunit without Browser Support with Mobile Devices through Bbluetooth], (2015), *Holodilnaya Tehnika I Tehnologiya*, No. 2, *Kviten*, 2015. Odessa, Ukraine, pp. 65 – 71 (In Russian).

5. Hammer-Lahav E. (ed.), (2010), The OAuth 1.0 Protocol, *IETF RFC 5849 (Informational)*, April 2010, (In English) [Electronic resource], Available at: URL: http://tools.ietf.org/html/rfc5849 (accessed 23.06.2016).

6. Basney Jim, and Gaynor Jeff, (2016), An OAuth Service for Issuing Certificates to Science Gateways for TeraGrid Users, *National Center for Supercomputing Applications University of Illinois at Urbana-Champaign 1205 West Clark Street*, Urbana, Illinois 61801, Article No. 32, (In English) [Electronic resource], Available at: URL: http://dl.acm.org/citation.cfm?id=2016776 (accessed 23.06.2016).

7. Richardson Leonard, and Ruby Sam, (2011), RESTfull Web Services Web services for the real world, *O'Reilly Media*, May 2011, pp. 188 – 205.

8. Mark Masse, (2013), REST API Design Rulebook, *O'Reilly Media*, pp.23 – 35.

9. Fielding R., and Reschke J. (2014), Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, *IETF RFC 7230*, June 2014, (In English) [Electronic resource], Available at: URL: https://tools.ietf.org/html/rfc7230 (accessed 23.06.2016).

10. Siewert Sam B., (2013), Cloud Scaling, Part 1: Build a Compute node or Small Cluster Application and Scale with HPC, *University of Alaska Anchorage*, (In English) [Electronic resource], Available at: URL: http://www.ibm.com/developerworks/cloud/library/cl-cloudscaling1-hpcondemand/ (accessed 23.06.2016).

11. Webber Jim, Parastatidis Savas, Robinson Ian, (2012), REST in Practice Hypermedia and Systems Architecture, *O'Reilly Media*, September 2012, pp. 285 – 351.

Received 08.06.2016

Surkov
Sergey Sergeevich,
Post-graduate student
Computer Intellectual systems and networks
Odessa National Polytechnical University,
tel .: +38 (091) 916-40-91.
E-mail:
k1x0r@ukr.net

Martynyuk
Oleksandr Nikolaevich,
PhD., Associate Professor
Computer Intellectual systems and networks
Odessa national Polytechnic. University,
tel .: +38 (067) 489-81-69.
E-mail:
anmartynyuk@ukr.net