Piotr Nawrocki
Aleksander Mamla

DISTRIBUTED WEB SERVICE REPOSITORY

Abstract *The increasing availability and popularity of computer systems has resulted in a demand for new language- and platform-independent ways of data exchange. This demand has, in turn, led to significant growth in the importance of systems based on Web services. Alongside the growing number of systems accessible via Web services came the need for specialized data repositories that could offer effective means of searching the available services. The development of mobile systems and wireless data transmission technologies has allowed us to use distributed devices and computer systems on a greater scale. The accelerating growth of distributed systems might be a good reason to consider the development of distributed Web service repositories with built-in mechanisms for data migration and synchronization.*

Keywords Web services, repository, heterogeneity, data replication, node balancing

Citation Computer Science 16 (1) 2015: 55–73

1. Introduction

There has been a very rapid development of computer systems architecture in recent years. Furthermore, the role of computer systems in business and everyday life has increased significantly. Along with the decrease in hardware prices came a greater availability of computer systems that support modern business.

Many corporations started to replace their expensive dedicated mainframe solutions with those based on microcomputers, personal computers, and clouds [12]. Companies developed special computer systems to support their production and research. Those systems were required to communicate and exchange data with one another. However, the increasing number of participating systems made this collaboration ever more complicated. Moreover, these systems not only needed to communicate with the remaining systems within the company but also had to be compatible with those used by the companys business partners. Such requirements resulted in a need for a new method of data exchange that would be independent of programming languages, operating systems, or hardware platforms. All of this has led to the development of Web services in a number of areas, such as Smart Building, Internet of Things, and Cloud Computing [16].

Web services offer programmers entirely new possibilities when designing and developing distributed systems or Web applications. They introduce a new layer of abstraction above application servers, operating systems, and hardware platforms.

With the growing popularity of systems using Web services emerged the demand for special tools that could manage Web service metadata and allow effective Web service discovery. Increasingly often, special Web service repositories are used for that purpose. Figure 1 illustrates the role of Web service repositories in the Web service architecture and the lifecycle of a single service.

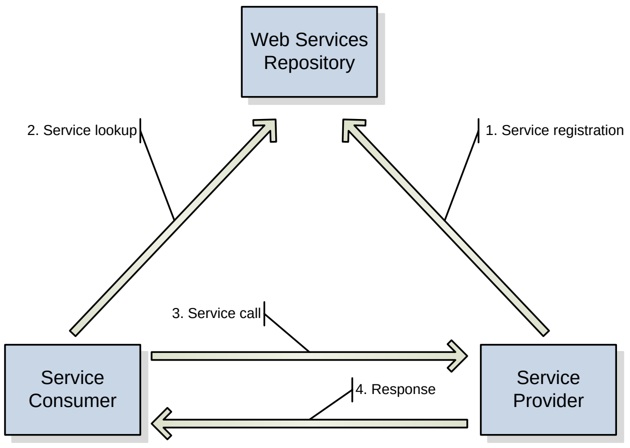


Figure 1. Role of a Web service repository in the Web service architecture.

The rapid development of mobile devices and wireless data transmission technologies has also accelerated the spread of Web services and the metadata that describe them. This spread has, in turn, resulted in the need for a distributed repository of Web service metadata with additional aspects of distributed systems that must be considered, such as data persistence, consistency, and availability.

2. Related work

Numerous studies and publications have appeared on Web service repository issues. However, few of them deal with the replication of data between repositories.

The best-known solution for public repositories used to be UDDI (Universal Distribution, Discovery and Interoperability). This was developed by an international consortium founded by Microsoft, IBM, and Ariba. UDDI [8] was meant to deliver a public repository of Web services and a unified standard for publishing and discovering the Web services available. However, the solution did not see widespread adoption; and so, Microsoft and IBM decided to shut down their public UDDI registries in 2006.

Many articles discuss the problem of efficient Web service discovery mechanisms in repositories [11][7]. In [13], the authors propose a Semantic Web services Clustering (SWSC) method that extends the semantic representation of Web services. With the introduction of this method, the Web service discovery process has been improved, allowing for the precise location of Web services with expected properties. In another article [5], the authors propose a heuristic approach to the semi-automatic classification of Web services. It allows consumers and repository administrators to categorize services manually, thus improving the automatic organization and discovery of Web services. Neural networks may also be used in the discovery of Web services. In [2], the authors propose a framework for enabling the efficient discovery of Web services using Artificial Neural Networks (ANN).

As suggested by Atkinson, Bostan, Hummel, and Stoll [3], a new approach to the design and implementation of Web service repositories is required; one that limits human maintenance of repositories and increases the role of automated management mechanisms. Other papers [6, 15] state that new kinds of Web service repositories could be helpful in the area of service composition and providing QoS for Web services.

An important aspect concerning the information collected in a Web service repository is ensuring that it is up to date. A situation may occur where a Web service stops working, changes its QoS and other parameters (such as cost or URI). There are different approaches to solving this problem. One of them is checking whether the entry in the repository is up-to-date. However, in cases where it is found that the Web service in question has stopped working or its parameters have changed, the user is only informed that the Web service is not working properly. Therefore, a better solution, proposed in article [1], is an integrated approach for dynamically adapting web service compositions based on non-functional requirements (NFRs). The proposed solution provides methods for identifying semantically-equivalent services to allow the selection of the best possible Web service to be invoked. This approach focuses more on providing Web services with a specific functionality and appropriate QoS than on continuously checking whether a particular Web service is available.

Recent publications have attempted, inter alia, to combine the concept of clouds and repository services [9], with issues related to storage, replication, and migration of data across multiple repository storage providers [17].

Referring to the solutions already developed, the authors of this article have sought to respond to contemporary challenges of storage and automatic replication of information, particularly with respect to services in distributed data repositories.

3. Web service repository

In the design and development of a distributed Web service repository, the following issues had to be considered:

- **data replication** – due to the distributed nature of the repository being developed, the system has to provide reliable and effective mechanisms for data replication between cooperating instances;
- **error detection** – the repository should detect and handle errors affecting single nodes;
- **data security** – the system developed ought to implement security mechanisms with respect to stored data.

The designed system must incorporate special mechanisms that address all of these requirements.

3.1. System architecture

The developed Web service repository has been designed and implemented as a distributed system. Other features of this system include:

- **platform independence** – the repository is intended to operate with different operating systems;
- **fault tolerance** – the repository is supposed to detect and handle any outages or errors affecting integrated hosts;
- **self-organization** – cooperating hosts organize themselves in a hierarchical structure to reduce the number of connections maintained and improve the speed of data synchronization across the repository;
- **extensibility** – the data model used by the repository offers users considerable discretion with respect to storing Web service description entries.

A single system instance (node) may work as a standalone repository or may connect with other instances to create a distributed repository. The system introduces the concept of a *group* – a set of data on Web services and the users with rights to access them, which is stored on at least one node.

Each node may maintain multiple groups at the same time (see Figure 2). All group data (i.e., Web service entries and user information) are synchronized across all of the nodes that maintain the group in question.

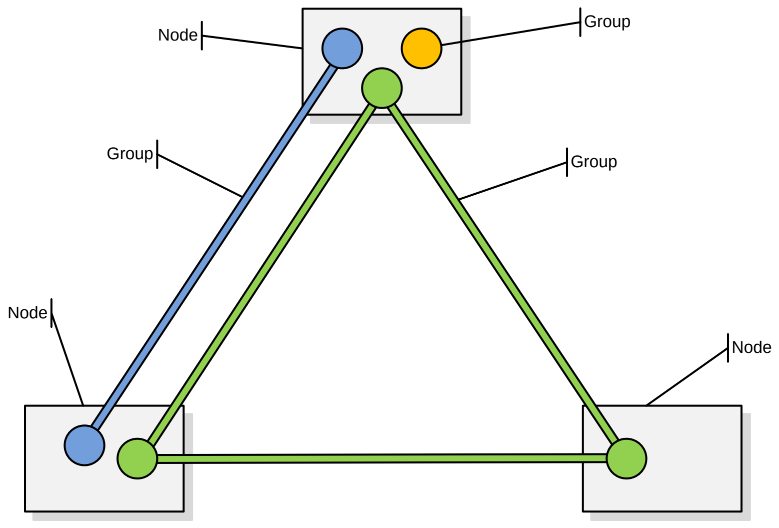


Figure 2. Concept of groups in the implemented repository.

Each node may create a new group, connect to an existing group, or disconnect from a group. If group data has only been stored on a single node, disconnecting that node from the group is tantamount to permanently deleting this groups data.

Groups have unique identifiers and *private keys* that are used when new nodes connect to the group and during communication between connected nodes.

Figure 3 presents the architecture of the implemented system. The roles of individual modules are:

- **network communication module** – responsible for receiving and sending messages from/to other nodes;
- **group management module** – responsible for creating new groups and connecting to/leaving existing groups. When a node creates a new group or connects to an existing one, this module is responsible for maintaining contact with other nodes organized into the group and for ordering data replication between nodes;
- **data replication module** – performs data replication between cooperating nodes;
- **database access module** – provides an interface to the data persistence layer;
- **user interface module** – makes the system functionality accessible to the user through the HTTP interface.

The entire network communication between subgroup members and neighbors is encrypted with the groups private key (requested from the node at the time it connects to the group). If any node receives a message encrypted with an invalid key, this message is disregarded. This mechanism gives system administrators control over access to the repository.

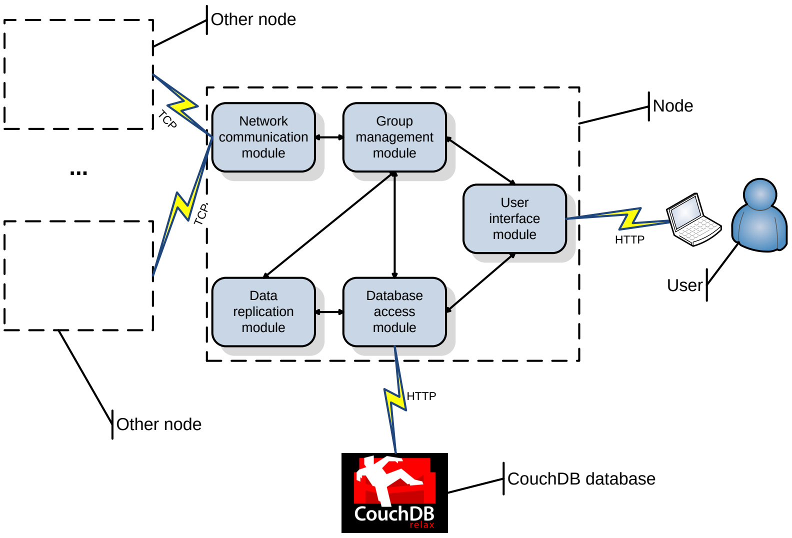


Figure 3. System architecture.

3.1.1. Hierarchical structure of the repository

Where multiple nodes connect to the same group, they work together as a single distributed repository and, thus, require mechanisms for mutual communication and data replication. The implemented system takes advantage of the built-in data replication features provided by CouchDB. However, to use these features effectively, a repository has to maintain information about cooperating nodes and formulate the best strategy for effective data replication.

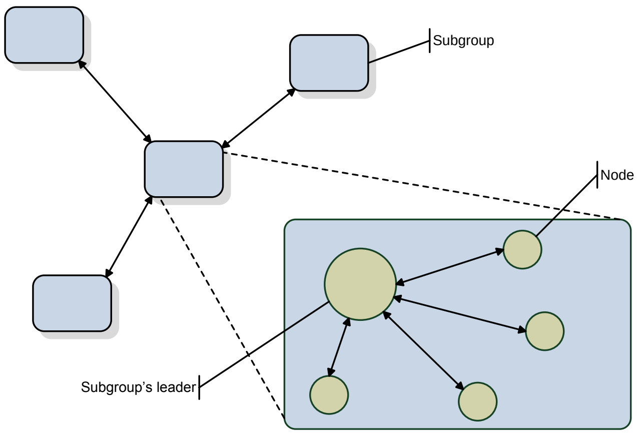


Figure 4. Hierarchical structure created by nodes.

In order to ensure the fastest and most-effective data replication, cooperating nodes automatically organize themselves into a hierarchical structure (see Figure 4). Individual nodes connected to the group divide into special *subgroups*. Each subgroup may contain up to five nodes, with one designated subgroup leader. Every subgroup may have up to three neighbors (adjacent subgroups). A subgroup leader is responsible for:

- checking the availability of other subgroup members;
- checking the availability of leaders of adjacent subgroups;
- ordering data replication between subgroup members and subgroup neighbors.

A subgroup leader continuously monitors the availability of subgroup members and sends them information on current subgroup status; i.e., a list of active members and neighbors. As a result, all subgroup members have up-to-date information about subgroup structure. Regular members also monitor the availability of the leader. If the leader is inactive for an excessively-long period, the remaining members elect a new leader.

Leaders of adjacent subgroups also maintain continuous communications with one another, exchanging information about other neighbors and overall group structure.

The hierarchical structure of the repository is managed upon the connection of new nodes (Figure 5 contains a detailed diagram of the algorithm used for that purpose). Cooperating nodes tend to achieve a well-balanced graph structure (see Figure 6).

3.1.2. Data replication

Each node monitors changes to the database using the CouchDB *changes feed* mechanism. Where a change in data is detected, the node has to notify other nodes about this in order to synchronize the data across the repository. The notification process depends on the type of the node in question:

- if the node is a subgroup leader, it replicates its database to all subgroup members and neighbors;
- if the node is a regular member, it notifies the subgroup leader about the change. The leader replicates data from the node in question and then replicates it to other members and neighbors.

As a result, all changes are propagated across the entire repository. The time required for complete replication reaches maximum where a change is made on a node that is not a subgroup leader and its subgroup is the first or last vertex of the graph diameter. Thus, the maximum time required to propagate changes across a repository consists of:

- the time required to replicate changes from a member of the first subgroup on the graph diameter to its subgroup leader;
- replication times between leaders of subgroups that belong to the graph diameter;
- the time required to replicate changes to a member of the last subgroup on the graph diameter from its subgroup leader.

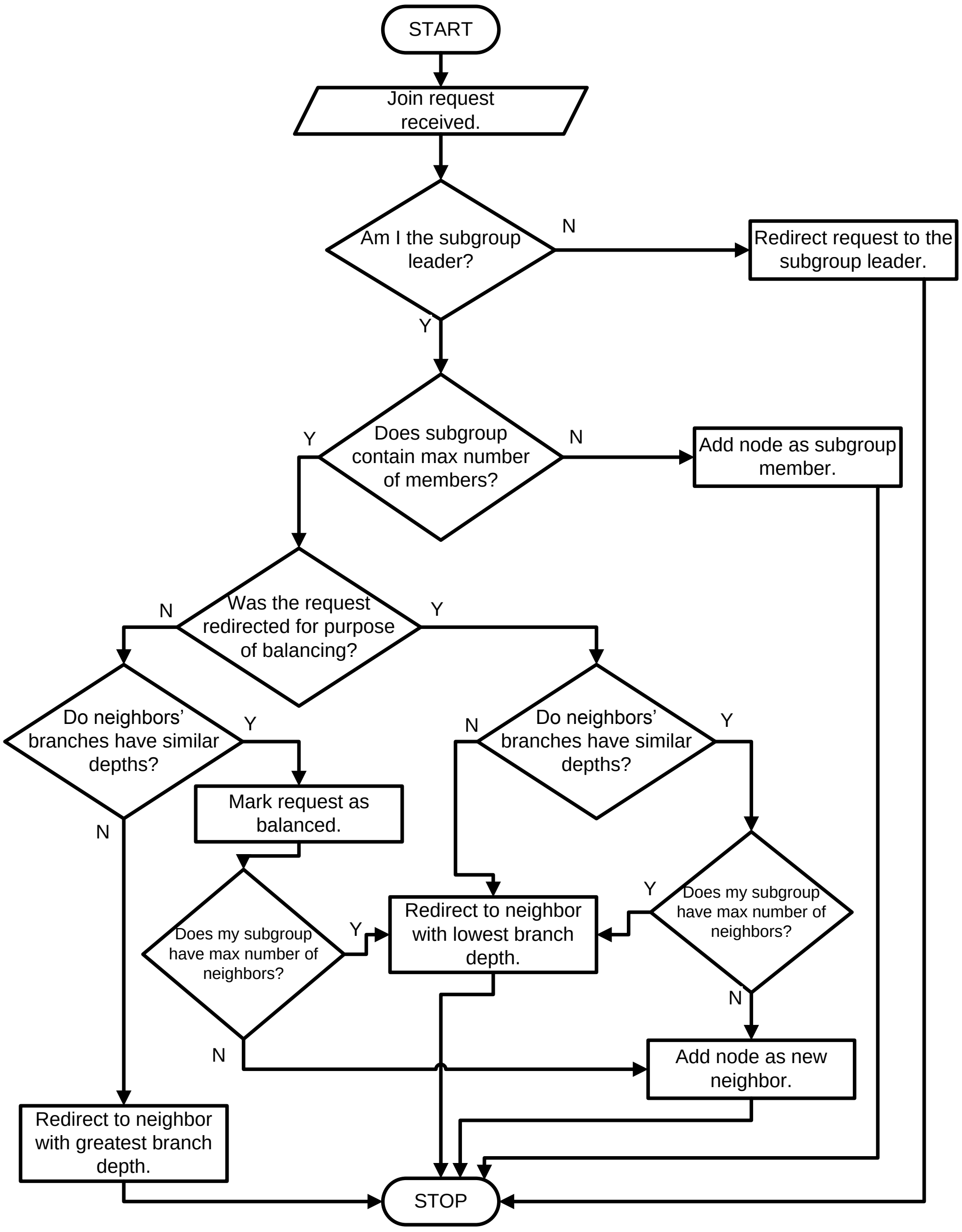

START
Join request received.
Am I the subgroup leader?
N
Redirect request to the subgroup leader.
Y
Does subgroup contain max number of members?
Y
N
Add node as subgroup member.
Was the request redirected for purpose of balancing?
N
Y
Do neighbors' branches have similar depths?
Y
Mark request as balanced.
N
Do neighbors' branches have similar depths?
N
Y
Does my subgroup have max number of neighbors?
Y
Redirect to neighbor with lowest branch depth.
Does my subgroup have max number of neighbors?
Y
N
N
Add node as new neighbor.
Redirect to neighbor with greatest branch depth.
STOP

Figure 5. Algorithm for adding a new node to the repository.

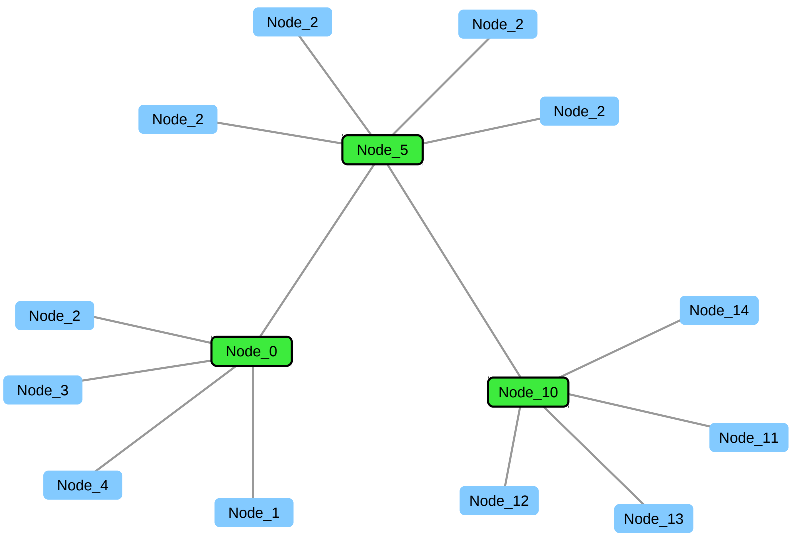


Figure 6. Graph structure created by nodes (subgroup leaders are framed).

The time required to propagate changes can be expressed by the following formula:

$$\sum_{i=1}^{diam(G)+2} (T_{w_i} + T_{r_i}) \tag{1}$$

where:

$diam(G)$ – the diameter of the graph created by subgroups,

T_{w_i} – the time required to discover data change on node i; $\forall i\ T_{w_i} \in \langle 0\,\mathrm{s}, 30\,\mathrm{s}\rangle$,

T_{r_i} – the time required to replicate changes between node i and node $i+1$.

Replication from the subgroup leader to other subgroup members and neighbors is conducted simultaneously, and thus, the delays that might occur due to the replication sequence have been disregarded.

3.2. Implementation

The main criteria adopted when selecting technologies to implement the repository were as follows:

- **multi-platform** – the repository was intended to work in heterogeneous environments, and thus, the need for technologies that can be used on multiple computer platforms;
- **support for distributed processing** – the technologies used should enable operation in distributed environments;
- **high level of mutual integration** – the technologies selected should be easy to integrate with one another.

Apache CouchDB^{TM}is a *NoSQL*, document-oriented, and schema-free database. Data is stored in the form of documents saved in the JSON format (a sample document is shown in Listing 1). Each document has a unique identifier. Documents are stored in a flat address space, and there are no correlations between them. CouchDB also offers the ability to attach files to saved documents. This allows the database to store data that cannot be represented in the JSON format (e.g., images, binary files).

Listing 1. Sample Apache CouchDB^{TM} document.

```
{
   "_id": "58e045044a00783cfc73a8c8c5a658f4",
   "_rev": "15-e28f2dc7b36a769df2a369ee683a93d4",
   "title": "Sample document",
   "description": "Example of CouchDB doc",
   "create_date": 1401895317183,
   "version": 15
}
```

The aggregation and representation of stored data is achieved using the special *views* mechanism. Views are functions written in JavaScript and saved in special *design documents*. When generating the result of a view, the database iterates through all documents and passes them to the view function. The function must determine if the document in question should be considered part of the result and then provide its representation as a *key-value* pair. Passing documents to views does not affect them, and thus, multiple views may operate on database content and provide different data-representation models depending on current requirements.

Among the most-important features of CouchDB are its replication mechanisms. The database offers incremental, fault-tolerant master-master replication of data with conflict management. This means that the same database can be stored on multiple servers, and all of them can query for data as well as create, edit, and remove documents from their local copy of the database. Later, those changes can be bi-directionally replicated among all instances.

All interactions with the CouchDB database occur via the HTTP interface.

Node.js is a software platform created on the basis of the V8[1] JavaScript engine developed by Google. It makes it possible to build fast, scalable, data-intensive network applications written in JavaScript. Node.js uses an asynchronous, non-blocking, event-driven model of I/O operations.

Since its debut in 2009, Node.js has become a very popular programming platform [4]. It has a broad set of built-in libraries and an enormous base of user-created modules available through the NPM[2] tool.

Both the Apache CouchDB^{TM}and Node.js are multi-platform systems available for the Windows, OS X, and Linux operating systems [14, 10].

[1]V8 – created by Google, open source JavaScript engine. It was originally developed for the purpose of Google Chrome web browser.

[2]NPM (Node Package Manager) – package management tool for Node.js.

4. Tests

This section presents the results of the conducted test cases. All test cases can be divided into two categories. The first category encompasses performance tests conducted in order to analyze application stability and its ability to handle heavy data traffic. The second category includes tests concerning the organization of autonomous repository nodes. These tests cover cases related to the hierarchical structure of the repository, such as the correct creation of repository structure, the impact of node organization on repository performance, and node-rebalancing mechanisms.

4.1. Performance tests

Performance tests have been performed on a single node running on a machine with the following hardware specifications:
Processor – Intel® Core™i7-2630QM CPU, 8x2.00 GHz, 8 GB RAM
Operating system – Ubuntu 12.04 LTS
Node.js version – v0.10.10
Apache CouchDB™version – 1.3.0

The tests have been performed with the JMeter[3] tool calling the RESTful interface, provided by the repository.

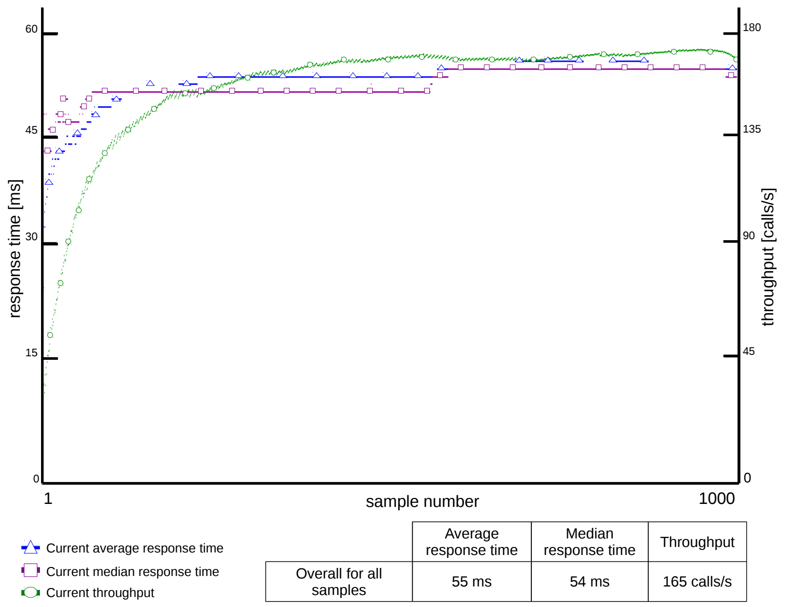
The test case included creating a new Web service entry in the repository and attaching a wsdl file to it. The test was performed in three different configurations. Each configuration resulted in a total of one thousand calls to the application, but individual configurations differed in the number of users simultaneously using the repository. The first configuration simulated ten users making one hundred calls each, the second one simulated one hundred users with ten calls each, and the last one simulated one thousand users with one call each.

Figure 7 presents test results for a simulation of ten simultaneous users performing 100 requests each. After about 200 requests, server response time stabilized, and the average response time was 55 ms. The achieved system throughput level was 165 calls/s.

In Figure 8, results for a simulation of 100 users performing ten calls each are shown. The average response time was equal to 316 ms and throughput amounted to 273 calls/s. The final configuration simulated 1000 simultaneous users performing one request each. The average response time reached 1587 ms and throughput amounted to 257 calls/s (Fig. 9).

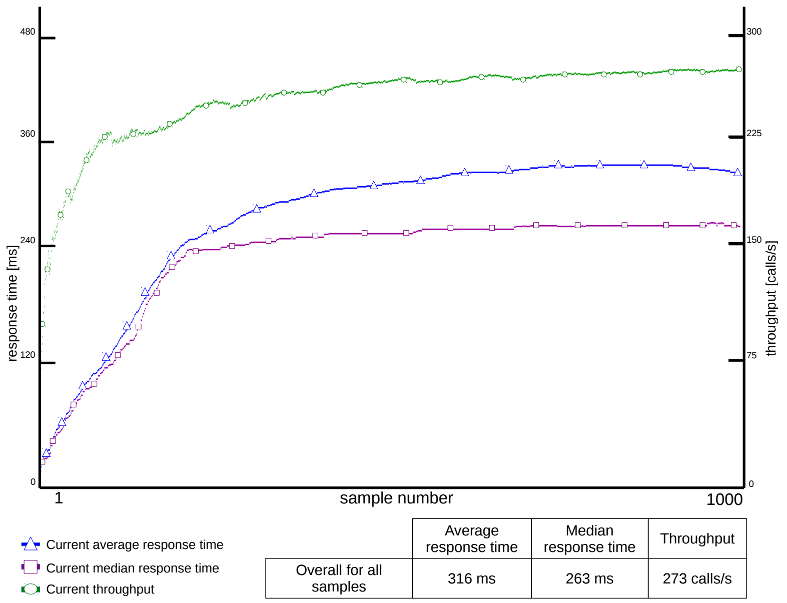
The results of the tests performed demonstrate that the repository created is able to handle increased load without a significant performance hit. Even with a large number of users simultaneously interacting with the repository, response times were satisfactory, and the application did not experience any unexpected issues.

[3]Apache JMeter™– tool written in Java, allows to execute load tests and performance tests of various types applications.



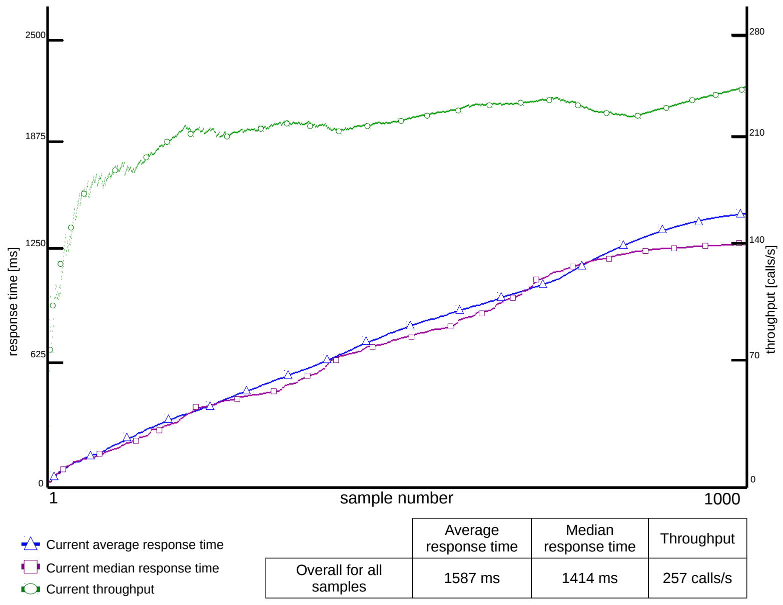
	Average response time	Median response time	Throughput
Overall for all samples	55 ms	54 ms	165 calls/s

Figure 7. System response time for creating a new repository entry with a file attached (10 users, 100 requests each).



	Average response time	Median response time	Throughput
Overall for all samples	316 ms	263 ms	273 calls/s

Figure 8. System response time for creating a new repository entry with a file attached (100 users, 10 requests each).



	Average response time	Median response time	Throughput
Overall for all samples	1587 ms	1414 ms	257 calls/s

Figure 9. System response time for creating a new repository entry with a file attached (1000 users, 1 request each).

These results demonstrate that the implemented solution is able to maintain satisfactory performance while operating under heavy user load.

4.2. Hierarchical repository structure

Tests of hierarchical structure of the repository were performed using multiple cooperating system instances. Single instances were run on virtual machines.

In order to test if the repository maintains a correct structure while adding new nodes, a new group was created. Subsequently, a total of 23 nodes connected to the newly-created group. All of the joining nodes sent requests to the same node, so it was possible to test whether the join requests would be properly redirected. Listing 2 shows the redirection information captured in the application logs of one of the nodes.

Listing 2. Join request redirection noted in application logs.

```
[22:16:47][group_manager][3d365c236e057c6ef04477dab4502cac] Join
    request from 192.168.1.107:8725 redirected to 192.168.1.107:8125
```

All log entries have the following form:

[*time*][*module name*][*group identifier*] *message*

Figure 10 presents the graph created by the connecting nodes, which indicates a successful test result.

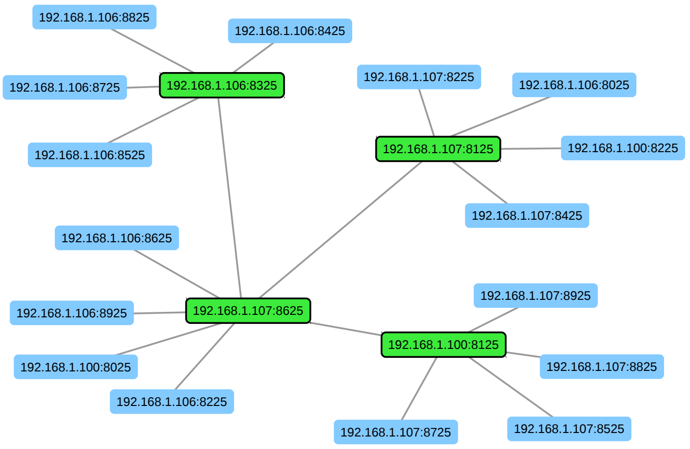


Figure 10. Graph created by repository nodes (subgroup leaders are framed).

In the test of new subgroup leader election, six nodes were connected to cooperate within a group. As expected, the nodes created two subgroups (one with five members and the second with a single node). Both subgroups had a designated subgroup leader. During the test, the leader of the first subgroup was disabled. As expected, the other subgroup members discovered that the leader did not communicate, elected a new one, and notified their subgroup neighbors. The election of the new leader is reflected in the application logs shown in Listing 3. Individual lines are taken from the logs of the new leader(1), a subgroup member(3), and a subgroup neighbor(5).

Listing 3. New subgroup leader election noted in application logs.

```
[14:22:20][group_manager][f83487eafe9e7d44a6f8e1e44a0d4c06] Elected for
      leader of group test_group
[14:22:20][group_manager][f83487eafe9e7d44a6f8e1e44a0d4c06]
     192.168.1.106:8025 elected for leader of group test_group
[14:22:20][group_manager][f83487eafe9e7d44a6f8e1e44a0d4c06] Neighbour
     group leader 192.168.1.100:8125 changed to 192.168.1.106:8025
```

The conducted tests have demonstrated that an automatically-created repository maintains the proper node structure. Already at the stage of connecting a new node, the other nodes cooperate to select the right location for that node so the structure of the repository remains balanced. During subsequent collaboration as well, nodes tend to preserve the correct structure of the repository. The most important observation is that the structure of the repository is maintained automatically without any intervention from users. Owing to that fact, the repository is self-organizing in order to improve its performance.

4.3. Rebalancing repository structure

The objective of the next test was to demonstrate the impact of node structure on the time required to replicate data across the repository. For the purpose of testing, two repository configurations were prepared. Both consisted of 23 cooperating nodes, but only in one of them were the nodes arranged into subgroups of five, and every subgroup tended to have three neighbors (adjacent subgroups). The configurations used in the test described are shown in Figure 11.

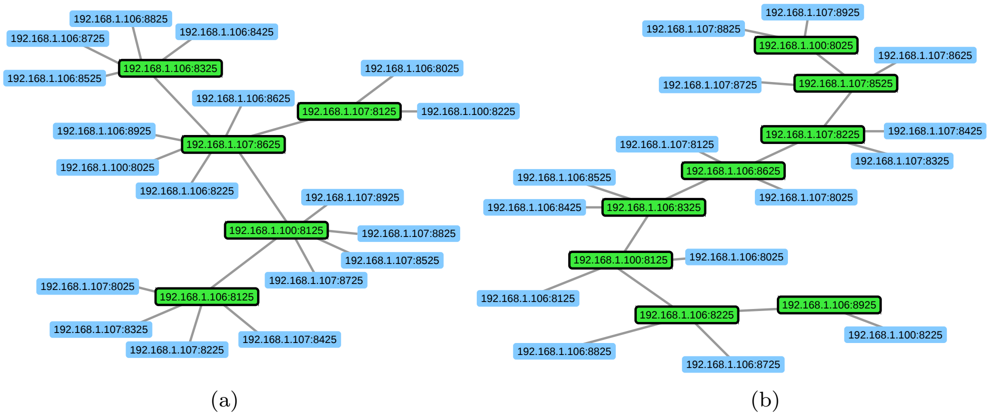


Figure 11. Repository configurations used in the replication time test – balanced tree (a) and unbalanced tree (b).

During the test, two nodes that were the most distant from each other were selected. Then, a new Web service entry was added on one of them, and the time required to replicate this change to the second node was measured.

For each configuration, ten trials were performed. The average time of change propagation across the repository with a balanced node structure amounted to 89.3 s. When compared to the unbalanced structure (where the average propagation time was 152.2 s), a considerable impact of the structure created by the nodes can be observed. In this particular scenario, propagation time for the unbalanced tree was 1.7 times greater than for the balanced tree (see Figure 12).

As shown in the previous test, the tree structure plays a key role in replication time within the repository. The implemented system is able to detect an invalid node structure and rebalance it.

For the purpose of tree-rebalancing tests, an unbalanced tree was prepared. As expected, the system discovered the unbalanced structure and initiated the rebalancing process. Tree structures before and after rebalancing are shown in Figure 13.

Repository balancing is performed by sending requests to single nodes with commands to reconnect to the group. Owing to that, rebalancing uses the same algorithm as when a new node is connected. As a result, nodes are once again connected to

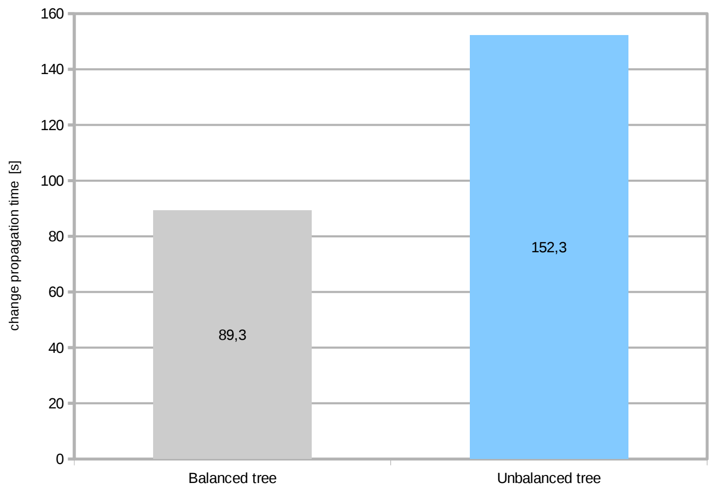
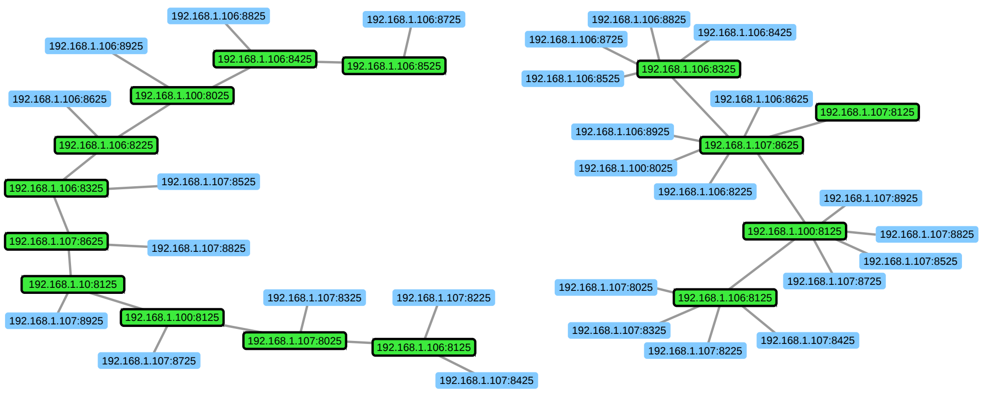


Figure 12. Comparison of change propagation times in the balanced and unbalanced tree.



(a) (b)

Figure 13. Tree structure before (a) and after (b) rebalancing.

form a balanced structure. Listing 4 shows a log entry from one of the nodes that was ordered to reconnect to the group.

Listing 4. Reconnect request noted in application log.

```
[22:10:24][group_manager][3d365c236e057c6ef04477dab4502cac] Rebalancing
     request received from 192.168.1.106:8325. Rejoining group at:
    192.168.1.106:8325
[22:10:42][group_manager][3d365c236e057c6ef04477dab4502cac]
    successfully reconnected to group: 3
    d365c236e057c6ef04477dab4502cac
```

The results of the tests performed have demonstrated the crucial role of proper repository structure for its performance. As demonstrated in Figure 12, the repository structure has a tremendous impact on the time required to propagate data changes across all connected nodes. This has been one of the main reasons to implement the rebalancing mechanism, which is responsible for detecting irregularities in the structure of the repository and restoring the balance between cooperating nodes.

5. Conclusions and further work

The implemented system is a very flexible tool for storing Web service information. The created repository can be used as a self-maintained, distributed system composed of autonomous nodes that can work in heterogeneous environments.

The nodes cooperating within the distributed repository organize themselves automatically into a hierarchical, balanced-graph structure. This structure, alongside the powerful data-replication mechanism provided by CouchDB, results in a fault-tolerant, fast, and safe data-synchronization strategy.

The system provides a rich user interface in the form of a Web application and a convenient RESTful Web service.

Multiple sets of test cases have been performed to analyze application behavior. They have demonstrated that the created repository is able to withstand substantial load without a significant decrease in performance. The tests have also demonstrated that the nodes cooperating in order to create the repository form a self-organized and self-balancing collective of autonomous units. Future work may include further analysis of different node tree structures that could additionally improve data replication performance within the repository.

It is worth pointing out that the technologies used for the implementation of the system and its modular architecture allow for the further development of the repository in order to store data other than Web service information.

Acknowledgements

The research presented in this paper was partially supported by the Polish Ministry of Science and Higher Education under AGH University of Science and Technology Grant 11.11.230.124 (statutory project).

References

[1] Agarwal V., Jalote P.: From Specification to Adaptation: An Integrated QoS-driven Approach for Dynamic Adaptation of Web Service Compositions. In: *Web Services (ICWS), 2010 IEEE International Conference on*, pp. 275–282, 2010. `http://dx.doi.org/10.1109/ICWS.2010.39`.

[2] Al-Masri E., Mahmoud Q. H.: Discovering the best web service: A neural network-based solution. In: *Systems, Man and Cybernetics, 2009. SMC 2009.*

IEEE International Conference on, pp. 4250–4255, 2009. ISSN 1062-922X. `http://dx.doi.org/10.1109/ICSMC.2009.5346817`.

[3] Atkinson C., Bostan P., Hummel O., Stoll D.: A Practical Approach to Web Service Discovery and Retrieval. In: *ICWS*, pp. 241–248. IEEE Computer Society, 2007.

[4] Cantelon M., Holowaychuk T., Harter M., Rajlich N.: *Node.js in Action*. Running Series. Manning Publications Company, 2013. ISBN 9781617290572.

[5] Corella M., Castells P.: Semi-automatic Semantic-Based Web Service Classification. In: *Business Process Management Workshops*, J. Eder, S. Dustdar, eds, *Lecture Notes in Computer Science*, vol. 4103, pp. 459–470, Springer, Berlin Heidelberg, 2006. ISBN 978-3-540-38444-1. `http://dx.doi.org/10.1007/11837862_43`.

[6] Dustdar S., Schreiner W.: A Survey on Web Services Composition. *Int. J. Web Grid Serv.*, vol. 1(1), pp. 1–30, 2005, ISSN 1741-1106. `http://dx.doi.org/10.1504/IJWGS.2005.007545`.

[7] Garofalakis J., Panagis Y., Sakkopoulos E., Tsakalidis A.: *Web Service Discovery Mechanisms: Looking for a Needle in a Haystack?* In: *International Workshop on Web Engineering*, 2004.

[8] Kreger H.: *Web Services Conceptual Architecture (WSCA 1.0)*. Tech. rep., IBM Software Group, 2001.

[9] Li S., Xu L., Wang X., Wang J.: Integration of hybrid wireless networks in cloud services oriented enterprise information systems. *Enterprise IS*, vol. 6(2), pp. 165–187, 2012.

[10] Malcontenti-Wilson A.: *Node.js for Raspberry Pi Project*, 2013.

[11] Mukhopadhyay D., Chougule A.: *A Survey on Web Service Discovery Approaches*. In: *Advances in Computer Science, Engineering and Applications*, D. C. Wyld, J. Zizka, D. Nagamalai, eds, *Advances in Intelligent and Soft Computing*, vol. 166, pp. 1001–1012, Springer, Berlin Heidelberg, 2012, ISBN 978-3-642-30156-8. `http://dx.doi.org/10.1007/978-3-642-30157-5_99`.

[12] Nawrocki P., Soboń M.: Public cloud computing for Software as a Service platforms. *Computer Science*, vol. 15(1), 2014. ISSN 2300-7036.

[13] Nayak R., Lee B.: Web Service Discovery with additional Semantics and Clustering. In: *Web Intelligence, IEEE/WIC/ACM International Conference on*, pp. 555–558, 2007, `http://dx.doi.org/10.1109/WI.2007.82`.

[14] Niec M., Pikula P., Mamla A., Turek W.: Erlang-based Sensor Network Management for Heterogeneous Devices. *Computer Science*, vol. 13(3), 2012. ISSN 2300-7036.

[15] Ran S.: A Model for Web Services Discovery with QoS. *SIGecom Exch.*, vol. 4(1), pp. 1–10, 2003. ISSN 1551-9031. `http://dx.doi.org/10.1145/844357.844360`.

[16] Soliman M., Abiodun T., Hamouda T., Zhou J., Lung C. H.: Smart Home: Integrating Internet of Things with Web Services and Cloud Computing. In: *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International*

Conference on, vol. 2, pp. 317–320, 2013.
`http://dx.doi.org/10.1109/CloudCom.2013.155`.

[17] Waddington S., Zhang J., Knight G., Jensen J., Downing R., Ketley C.: Cloud repositories for research data addressing the needs of researchers. *Journal of Cloud Computing*, vol. 2(1), 13, 2013.
`http://dx.doi.org/10.1186/2192-113X-2-13`.

Affiliations

Piotr Nawrocki
AGH University of Science and Technology, Krakow, Poland `piotr.nawrocki@agh.edu.pl`

Aleksander Mamla
AGH University of Science and Technology, Krakow, Poland `alek.mamla@gmail.com`

Received: 26.09.2014
Revised: 12.12.2014
Accepted: 14.12.2014