Synchronized Content and Metadata Management in a Federation of Distributed Repositories of Chemical Learning Objects

Sergio Tasso[1], Simonetta Pallottelli[1], Osvaldo Gervasi[1(✉)], Razvan Tanase[1], and Marina Rui[2]

[1] Department of Mathematics and Computer Science, University of Perugia, via Vanvitelli, 1, 06123 Perugia, Italy
{sergio.tasso,simonetta.pallottelli,osvaldo.gervasi}@unipg.it, tangent.jotey@gmail.com

[2] Department of Chemistry and Industrial Chemistry, University of Genoa, via Dodecaneso, 31, 16146 Genoa, Italy
marina@chimica.unige.it

Abstract. The paper deals with the synchronization mechanism among the servers of a federation of distributed repositories for the constant updating of the didactic-scientific material, its properties and its locations. A shared metadata database is the synchronization point of reference and it allows to improve performance in terms of searching and downloading. The proposed federation is meant to deal with a large variety of different contents though the discussed prototype implementation is concerned with scientific and educational subjects in particular. Additional elements of evaluation are the capability of enhancing collaboration and fault tolerance.

Keywords: Repository · Synchronization · Learning objects · Content sharing

1 Introduction

Thanks to their advanced tools and interoperable environments, the new technologies, make available, any time and any place, learning object [1, 2] and repository facilities needful both for training and information.

The content, repository platforms and e-learning systems can share in these environments, are manifold. For our job, it is particularly interesting to note that repository platforms, because of their powerful and flexible structure, allow a full management of Learning Object Metadata (LOM).

In the current view, the e-learning content design implies the application of a development model featured by material serialization, which has the ultimate goal of obtaining self-consistent learning units (typically learning objects) in SCORM format.

In this context it suits the Glorep system [3], which, in order to facilitate the learning objects (LOs) phases production, proposes a shared and distributed solution for the education content, aimed at reuse, after a suitable classification and indexing [4, 5].

O. Gervasi et al. (Eds.): ICCSA 2017, Part III, LNCS 10406, pp. 14–28, 2017.
DOI: 10.1007/978-3-319-62398-6_2

The project is part of other research carried out within the workgroup which has provided interesting ideas for the construction of the materials and for their treatment, also in view of an extension of the taxonomy so far considered [6–10].

2 The Glorep Project

The leaders of the Glorep project (Grid Learning Object Repository) [11–15] are the Mathematics and Computer Science Department and the Department of Chemistry, Biology and Biotechnology of the University of Perugia. This project involves other Chemistry Departments of Italian and foreign Universities, among which there are the ones of Genoa and Thessaloniki (Greece) and is focused at implementing a federation of repositories, supported by a cluster of SMEs (small and medium-sized enterprises) coordinated by ECTN, in which the information is processed by LOs.

The aim of this activity is to integrate different software tools in order to enhance the Glorep strength and efficiency for making it the reference product of the Virtual Research environment [16, 17] called for funding at the recent EINFRA-9-2015 Horizon 2020 call [18].

Then, Glorep, the federation of distributed and shared repository is especially builded for:

- Making available to a large community of teaching and learning the didactic *and* scientific content they produced.
- Offering an environment that ensures the dynamic improvement of materials available through the correct storage and cataloging of revised versions

The federation is composed by repositories with equal rank, responsibilities, duties and functions, which exchange among them the whole information about available LOs [19, 20].

Glorep assigns to the modular CMS Drupal [21–23], the role of collecting, managing and tracing the information on the distributed and collaborative networks.

Drupal is a free and open source content-management framework written in PHP and distributed under the GNU GPL. It is powerful, flexible and customizable according to the opinion of the Web community made of more than a million of users, with more than one hundred developers. As already said, it is modular and highly flexible, provided with more than 30 thousand modules to extend and customize its functionalities. Moreover, it is multiplatform.

It should note that the Drupal modularity is due to a bunch of modules that perform only primary functions, but complemented, by its own developers, by APIs (Application Programming Interface) which allow creating a lot of additional modules to extend its functionality according to needs.

2.1 The GLOREP Features

There are various modules of Glorep interacting in order to allow a simple and quick input and to find a LO. Whereas there are only two main features allowing the concerned

modules to be effective and efficient: the use of the Dewey Decimal Classification (DDC) [24] cataloging system and the use of IEEE LOM [25, 26].

Starting from the Drupal configuration standard, we implemented four new modules in order to manage the wide federation and its contents:

- *LinkableObject* - To manage the LO: it enables the LO creation and upload it to the servers. It also allows managing LO permissions, defining who can create and who can view and download the learning material.
- *SearchLO* - To manage a distributed search of LO on wide federation: it is a searching system of LO easy and intuitive.
- *TaxAssistant* - To manage LO classification step: it analyses the related metadata entered by the user in order to help with the selection of the category better related to the LO.
- *CollabRep* - To manage the federation: it can be used to create, join and quit a federation, and it performs synchronization recovery measures in case of communication issues during updates.

Recently, fundamental changes have been made about synchronizing federated servers. In particular, the old *Collabrep* module has been replaced with the new *GlorepSyncIO* stand-alone daemon (as you can see in Fig. 1).

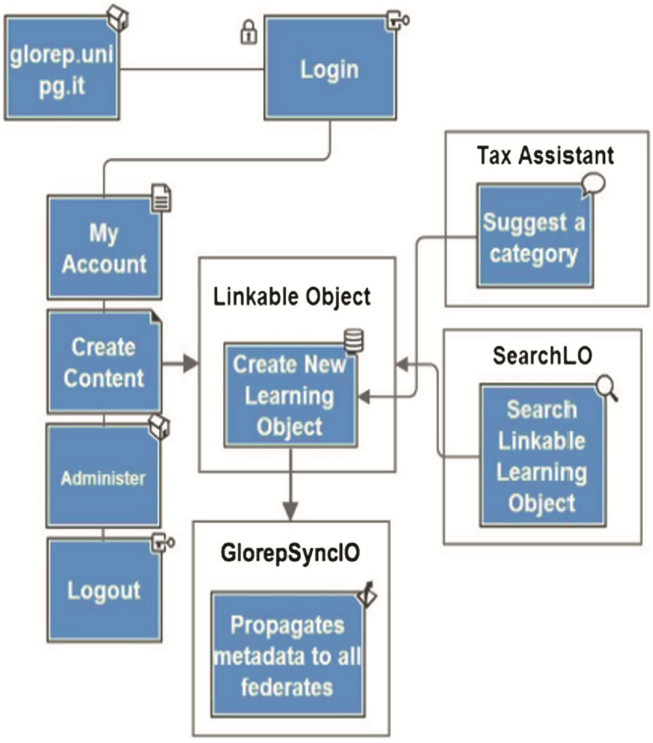

Fig. 1. The new schema of a Glorep server

3 The Synchronization Mechanism of GlorepSyncIO

3.1 Server.php

The main file of the entire daemon is called *server.php*, this is where everything comes together and where most of the defined methods are being put to use in order to achieve the main goal of the project: synchronization. This is the meat of the code [27, 28].

Before starting to talk about the mechanisms that are behind everything, we must first understand exactly what the exact problem is we are trying to solve at this point. We have a situation where a variable number of servers must provide data for a different number of applications (in this case a web application). Each server has their own database, containing data gathered from their users through forms, or any other kind of data input. At this point, each server has its storage of data. But, what if these servers would exchange data between them?

Proposition

One way of approaching this problem would be to directly transfer data from one server to another. Directly exchanging data sounds like a good idea, if the number of servers is not variable. We know, however, that our number of servers is variable, meaning that anyone can choose to leave or join the federation.

This implies other problems, such as allowing every server to detect and distinguish between the two events of leaving and joining the federation.

The resource cost of this method is also directly proportional to the number of servers of the federation, meaning that the more servers there are the more resources it will cost to keep everyone updated. This way of approaching is clearly a bad one.

Instead of using a direct point-to-point communication, a central database could serve as an easier way of communication between servers (see Fig. 2).

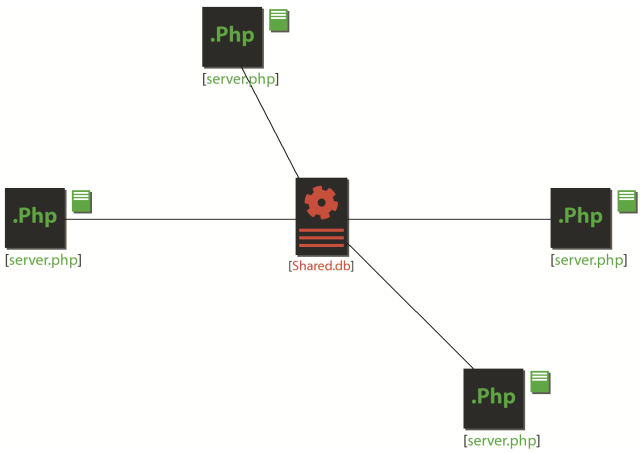

Fig. 2. The sharedBD based approach

The central piece is a database, nothing more. We will be calling that "shared database" from now on. In this mechanism, each server also has its own copy of the database stored locally.

3.2 Objects and Owners

The local database stores all kinds of objects gathered from around the federation, however we must distinguish between the ones created by the local federate and the ones created by other federates, to do so we define:

- *Owned Objects*:
  - The only objects the daemon will upload to the shared database are the ones belonging to the local federate.
- *Delegate Objects*:
  - The objects the local daemon must only download from the shared database.

3.3 Uploading

Let's say we are hosting one of these servers. Every N seconds, if the shared database is behind, the daemon would attempt to upload the local objects to the shared database. Understanding the meaning of *being ahead* or *being behind* is key.

We will say the local database *is ahead* of the shared database when the *owned objects* stored locally outnumber the *owned objects* stored on the shared database.

We will say the local database *is behind* the shared database when the *delegate objects* stored on the shared database outnumber the ones stored locally.

Note that this mechanism implies that the local database could be both *behind* and *ahead* of the shared database, in fact, the local database could contain more *owned objects*, and less d*elegate objects*.

However, how does the daemon know if the local database is *ahead* of the shared database?

It all comes down to how the objects are stored inside databases.

Every record has:

- an auto-incremented attribute *id*,
- a string i*d_fd* which indicates the name of the federate and
- an *id_lo*.

The *id_lo* attribute is a numeric attribute that when combined with the *id_fd* produces a unique key, this key identifies the object itself.

When working in the local database, the *id_lo* attribute and the *id* attribute, of the *owned objects*, overlap.

Selecting the local owned object with the highest *id_lo*, will return the newest *owned object* on the local database.

Selecting the *owned object* with the highest *id_lo* from the shared database, on the other hand, will return the newest *owned object* stored on the shared database.

The last step in finding out whether the local database is *behind* or *ahead*, is to compare the two objects.

If they both have the same *id_lo*, it means the local database is *not ahead*.

If the shared object has a lower *id_lo*, it means the local database is *ahead*.

If the shared object has a higher *id_lo*, there will never be an instance when the shared database will be *ahead* of the local database regarding the *owned objects*.

When the local database is ahead, simply passing along as offset the *id_lo* of the last known proprietary object from the shared database to a devoted method (Sync::upload_after_offset($offset, $local_db, $shared_db, $federate_name)), it will upload every missing object to the shared database.

3.4 Downloading

Once established that the local database is behind, the download routine can start.

The download routine uses the method Sync::download_after_offset($offset, $local_db, $shared_db, $federate_name) to download the missing *delegate objects*.

This time the last *delegate objects* must be compared.

If they both have the same *id_lo*, it means the local database is *not behind*.

If the shared object has a higher *id_lo*, it means the local database is *behind*.

If the shared object has a lower *id_lo*, the local database will never be *ahead* of the shared database regarding the *delegate objects*, that's because the *delegate objects* are downloaded from the shared database to begin with.

When the local database is *behind*, the download routine will download each object to the local database.

Each time an object is *being* retrieved from the shared database, a delete query is executed. This query will delete every object in the local database that uses the same *id_fd* and the same *id_lo* of the object that is about to be pushed in the database.

This mechanism allows the federates to overwrite their *own objects* and propagate the changes across the federation network.

3.5 Updating

Every iteration, after executing the upload routine and the download routine, a third routine is executed. In order to keep the shared database updated with every change that passes locally, the daemon iterates through a table stored in the local database called *lo_update_log*, this table is populated by a MySql trigger, which executes every time an object is updated locally.

Through the iteration, GlorepSyncIO, will retrieve the *owned objects* referenced in l*o_update_log*, which are not *draft*, and it will propagate the changes to the shared database.

The changes are propagated by simply sending the whole object with the new metadata, this object will be then stored in the shared database, and every other server that executes the download routine, will notice that the object is a duplicate and will overwrite the existing one/s (Fig. 3).

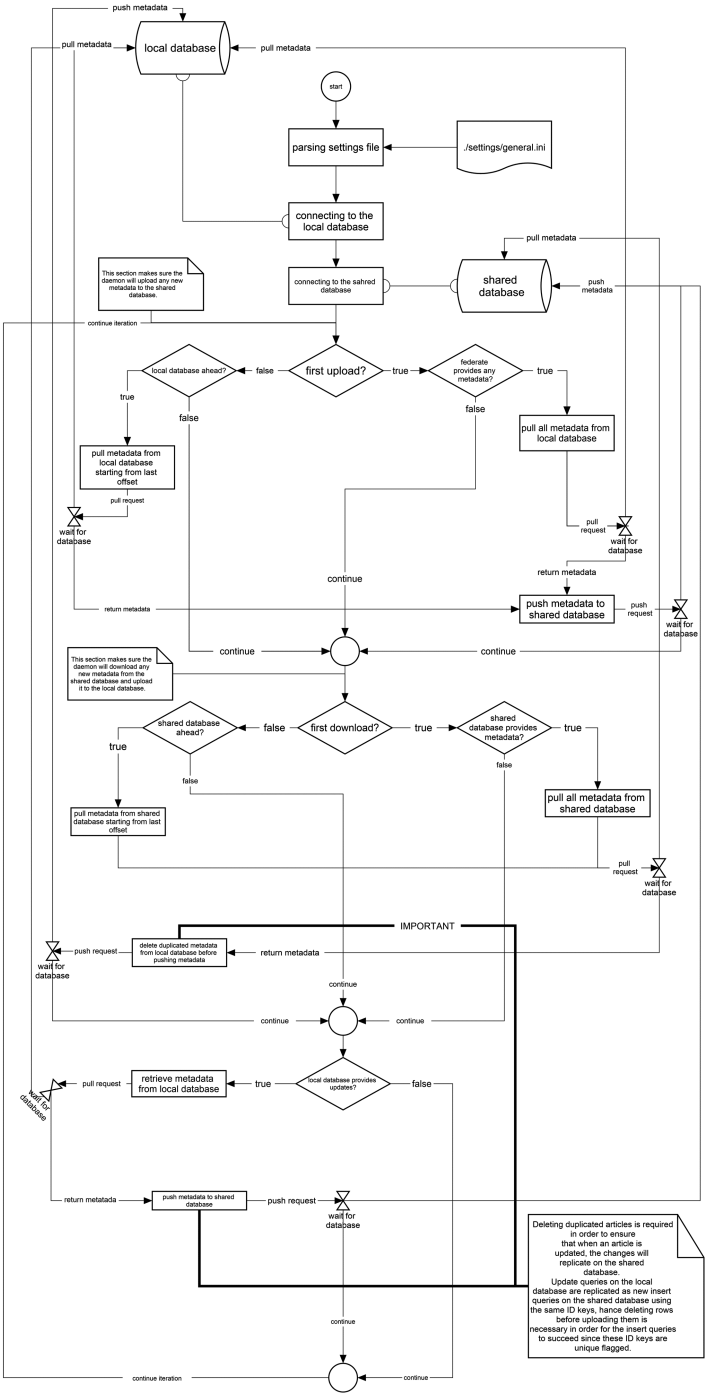

Fig. 3. The GlorepSyncIO schema

4 The GlorepSyncIO Components

The GlorepSyncIO daemon uses messages, readers and writers. Figure 4 shows the main features of each of these components used by GlorepSyncIO daemon.

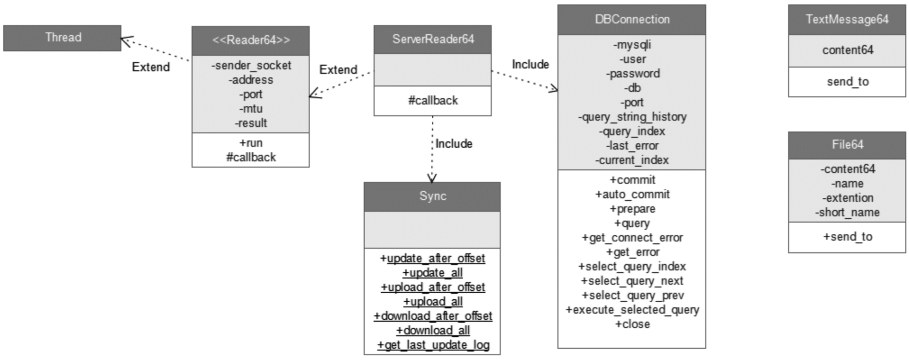

Fig. 4. The GlorepSyncIO class diagram

4.1 Messages

Once a machine is connected through a socket to our daemon, there are two types of messages that can be sent to us. The first one is a text message using the *TextMessage64* class, and the second type is a file using the *FileMessage64* class.

4.1.1 FileMessage64

The FileMessage64 class will create a FileMessage64 object, which will contain the data of the specified file. The data will not be stored as binary; instead, the binary string will be converted into a base64 string, and will be stored as such.

The reason behind storing a base64 string is quite simple: before sending our message, we want to wrap our file inside a JSON (JavaScript Object Notation) object [29], so we can send some side information about our file, such as the file's name, size, and type and we want to avoid JSON parsing errors in doing so.

JSON objects are very light and have a very simple syntax, thus are very easy to parse, however this is a double-edged sword.

Let's say I want to send a file as a binary string (which, again, we are not doing).

The actual string will not be stored as binary numbers in your variable; instead, it will be hashed using the Unicode standard.

Your file's contents will probably look something like this:

[…] &ÖS¥ýC´¤î#^#¤"#úP#Ü'{Ë[@#ÇŒh‰¾‹Tä□#–Rù5w'Î-'±V>·#OwjÑO„Õ […]

As you can see, the domain of the string contains some JSON wild characters such as "{", which could escape the JSON encoding algorithm.

One way to avoid this problem is to convert our binary string into a base64 string since the base64 alphabet table is much smaller and does not contain any wild characters that JSON might not like. Figure 5 shows the base64 characters table.

Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	F	47	V	63	/

Fig. 5. The table of base64 characters

4.1.2 TextMessage64

The TextMessage64 class acts almost identically to File64, they both send a stream of data to the other end of the channel, and they both convert the message in base64, to avoid the same parsing issue discussed above. The only difference here is the header of the package that is being sent. While File64 would send a JSON object containing the file's content, name, size and type, TextMessage64 would send an object containing the message and the type of its contents (which is set to "text-plain").

After the content of the message has been encoded (being either File64 or TextMessage64), it will then be wrapped inside a JSON Object, this object will then be also encoded into a base64 string. Finally, the string is then sent to the receiver.

4.2 Readers

Readers are a way for the server to read messages sent to their socket.

Reader64 is an abstract class which extends *Thread*, thus *Reader64* will define the method called *run*. Reader has a defined construct; it requires a socket (which will be

used to read the incoming data) and a Maximum Transmission Unit (MTU) (which is used to read chunks of bytes from the stream of data).

Reader will start reading chunks of data from the socket buffer, each chunk will be appended to a temporary variable until there's nothing left inside the buffer.

Once the message has been read, it's contents encoded in base64 will be passed to an abstract callback method, along with the sender's IPv4 address and the port of the sending application.

The message that is being received at this point is nothing more that a base64 encoded string. Before passing the string to the callback method the string is decoded from Base64 into a JSON object, so there's no need to decode it afterwards.

Reader64 is an abstract class, thus, it cannot be instantiated and hence *Server-Reader64* is needed.

ServerReader64 extends *Reader64* and it defines the body of the callback method, which is named *callback* which provides 3 parameters: the message, the IPv4 address, the port number of the sending application.

4.3 Writers

Writers are a simplified way to send JSON encoded messages through a socket.

Writer64 is an abstract class which extends *Thread*. It also have an abstract callback method, which will later be defined by the *TextWriter64* and *FileWriter64* classes, as they both extend *Writer64*.

4.3.1 TextWriter64 and FileWriter64

TextWriter64 and *FileWriter64* both extend *Writer64*, hence they both have to define a protected method called *callback*. This callback method takes one single attribute, and that is the socket to which the message is supposed to be written. Both these classes only exist to provide some "sugar syntax" [30] when sending data to the daemon. They are by no means required, in any situation, but they do help as they create a connection behind the scenes, and then they properly close it, all in one single line of code when creating an instance of the class itself.

4.4 DBConnection

DBConnection class does not extend *mysqli* (MySQL Improved Extension) [31], instead it uses a private *mysqli* object in order to connect to the database, and provides a limited number of methods to query the database in question.

When queries are sent to the database they are also saved to *DBConnection* inside an array, this allows *DBConnection* to quickly move backwards and forwards through the queries and execute them multiple times. Prepared statements are not saved when executed. *DBConnection* makes use of the *mysqli* construct and has the user to pass along the same parameters they would pass to *mysqli* (hostname, username, password, database, port), however, instead of providing these parameters hard-coding them, the

invoker can also only provide the first two parameters, referencing the *./settings/general.ini* file.

Using the second method might prove a bit more flexible, since the strings are fetched from a file which can be easily edited at runtime.

The first parameter passed to the *DBConnection* construct must indicate which of the two available databases it must connect to: the local database or the shared database.

The following keywords indicate the local database: *local*, *localhost*, *127.0.0.1*.

These strings are not treated as IP addresses; they are just convenient, easy to remember predefined strings, nothing more.

The following keywords indicate the shared database: *shared*, *sharedhost*.

4.5 Sync

The *Sync* class provides the methods used in the synchronization mechanism in *server.php*, and *ServerReader* class.

4.5.1 Available Methods

```
Sync::upload_after_offset(
        int             $offset,
        DBConnection    $local_db,
        DBConnection    $shared_db,
        string          $my_fed
        )
```

Parameters description:

$offset
: Offset from which objects will start being selected from database $local_db.

$local_db
: Database from which the data will be selected.

$shared_db
: Database to which the data will be uploaded.

$my_fed
: Name of the local federate.

Selects objects from $local_db starting from $offset and uploads them to $shared_db.

```
Sync::download_after_offset(
        int             $offset,
        DBConnection    $local_db,
        DBConnection    $shared_db,
        string          $my_fed
        )
```

Parameters description:

$offset
: Offset from which objects will start being selected from database $local_db.

$local_db
: Database from which the data will be selected.

$shared_db
: Database to which the data will be uploaded.

$my_fed
: Name of the local federate.

Selects objects from $shared_db starting from $offset and uploads them to $local_db.

After the remote object has been obtained and saved in memory, this method executes one more step before writing the object in the local database.

Before writing the object in the local database, every other object with the same *id_lo* and *id_fd* will be deleted regardless the circumstances.

```
Sync::get_last_update_log(
DBConnection    $local_db
)
```

Parameters description
$local_db
: Database from which the data will be selected.

Returns the last row from table *update_log* in database $local_db. Table *update_log* is used as a support table; no application should insert records in this table directly. The local database uses a trigger in order to insert records in *update_log*. The trigger will listen for an update event on the general table and insert a row in *update_log* afterwards, using the value of the *id_lo* attribute of the new object as the *local_id* attribute of the new record in the *update_log* table.

```
update_after_offset(
        int             $offset,
        DBConnection    $local_db,
        DBConnection    $shared_db,
        string          $my_fed
        )
```

Parameters description

$offset
: Offset from which updates will start being selected from database $local_db.

$local_db
: Database from which updates will be uploaded from.

$shared_db
: Database to which the updates will be uploaded.

$my_fed
: Name of the local federate.

Selects rows from *update_log* starting from $offset. For each fetched row the relative object from the general table is retrieved. If the object's status is *draft* or the object's *id_fd* is not the same as the local federate's name then do nothing.

Otherwise, the object will be uploaded to the shared database.

Finally, delete the currently selected record from update_log table and skip to the next record.

5 Conclusion and Future Work

The new stand-alone synchronization daemon GlorepSyncIO discussed in the present paper allows an efficient synchronization among the servers of the federation of distributed repositories of chemical learning objects Glorep.

The change from the previous configuration is due to the separation of the synchronization mechanism from the CMS hosting the local server.

The GlorepSyncIO daemon executes the synchronization routine once every N seconds. This way of proceeding might not be enough. In some cases we might want to directly query the daemon and force the synchronization routine using a RPC. Sockets allow the daemon to accept incoming (and request) connections from other machines and read the incoming message, which in this case would be a Remote Procedure Call (RPC).

Acknowledgements. The authors acknowledge ECTN (VEC standing committee) and the EC2E2 N 2 LLP project for stimulating debates and providing partial financial support. Thanks are due also to EGI and IGI and the related COMPCHEM VO for the use of Grid resources.

References

1. Raju, P., Ahmed, V.: Enabling technologies for developing next-generation learning object repository for construction. Autom. Constr. **22**, 247–257 (2012)
2. Sampson, D.G., Zervas, P.: Learning object repositories as knowledge management systems. Knowl. Manage. E-Learn. **5**(2), 117–136 (2013)
3. G-Lorep, March 2016. http://glorep.unipg.it
4. Xu, H.: Faculty use of a learning object repository in higher education. VINE J. Inf. Knowl. Manage. Syst. **46**(4), 469–479 (2016)
5. Nejdl, W., Tochtermann, K. (eds.): EC-TEL 2006. LNCS, vol. 4227. Springer, Heidelberg (2006). doi:10.1007/11876663
6. Franzoni, V., Mencacci, M., Mengoni, P., Milani, A.: Heuristics for semantic path search in wikipedia. In: Murgante, B., et al. (eds.) ICCSA 2014. LNCS, vol. 8584, pp. 327–340. Springer, Cham (2014). doi:10.1007/978-3-319-09153-2_25
7. Franzoni, V., Milani, A.: Semantic context extraction from collaborative networks. In: Proceedings of the 2015 IEEE 19th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2015. IEEE Press (2015). doi:10.1109/CSCWD.2015.7230946
8. Franzoni, V., Milani, A.: PMING distance: a collaborative semantic proximity measure. In: Proceedings of 2012 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2012, pp. 442–449. IEEE Press (2012). doi:10.1109/WI-IAT.2012.226
9. Franzoni, V., Milani, A.: Heuristic semantic walk for concept chaining in collaborative networks. Int. J. Web Inf. Syst. **10**(1), 85–103 (2014). doi:10.1108/IJWIS-11-2013-0031. Emerald
10. Franzoni, V., Chiancone, A., Milani, A.: A multistrain bacterial diffusion model for link prediction. Int. J. Pattern Recogn. Artif. Intell. **31**(9), 12–24 (2017). doi:10.1142/S0218001417590248. World Scientific
11. Pallottelli, S., Tasso, S., Pannacci, N., Costantini, A., Lago, N.F.: Distributed and collaborative learning objects repositories on grid networks. In: Taniar, D., Gervasi, O., Murgante, B., Pardede, E., Apduhan, B.O. (eds.) ICCSA 2010. LNCS, vol. 6019, pp. 29–40. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12189-0_3
12. Tasso, S., Pallottelli, S., Bastianini, R., Lagana, A.: Federation of distributed and collaborative repositories and its application on science learning objects. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011. LNCS, vol. 6784, pp. 466–478. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21931-3_36
13. Tasso, S., Pallottelli, S., Ferroni, M., Bastianini, R., Laganà, A.: Taxonomy management in a federation of distributed repositories: a chemistry use case. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012. LNCS, vol. 7333, pp. 358–370. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31125-3_28
14. Tasso, S., Pallottelli, S., Ciavi, G., Bastianini, R., Laganà, A.: An efficient taxonomy assistant for a federation of science distributed repositories: a chemistry use case. In: Murgante, B., Misra, S., Carlini, M., Torre, C.M., Nguyen, H.-Q., Taniar, D., Apduhan, B.O., Gervasi, O. (eds.) ICCSA 2013. LNCS, vol. 7971, pp. 96–109. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39637-3_8
15. Tasso, S., Pallottelli, S., Rui, M., Laganá, A.: Learning objects efficient handling in a federation of science distributed repositories. In: Murgante, B., et al. (eds.) ICCSA 2014. LNCS, vol. 8579, pp. 615–626. Springer, Cham (2014). doi:10.1007/978-3-319-09144-0_42

16. Mariotti, M., Gervasi, O., Vella, F., Cuzzocrea, A., Costantini, A.: Strategies and systems towards grids and clouds integration: a DBMS-based solution. Future Gener. Comput. Syst. (2017). doi:https://doi.org/10.1016/j.future.2017.02.047
17. Costantini, A., Gervasi, O., Zollo, F., Caprini, L.: User interaction and data management for large scale grid applications. J. Grid Comput. **12**(3), 485–497 (2014)
18. Laganà, A.: Horizon 2020 proposal for Research and Innovation actions Chemistry, Molecular & Materials Sciences and Technologies Virtual Research Environment (CMMST-VRE) Call EINFRA-9-2015 (2015)
19. Pallottelli, S., Tasso, S., Rui, M., Laganà, A., Kozaris, I.: Exchange of learning objects between a learning management system and a federation of science distributed repositories. In: Gervasi, O., Murgante, B., Misra, S., Gavrilova, M.L., Rocha, A.M.A.C., Torre, C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2015. LNCS, vol. 9156, pp. 371–383. Springer, Cham (2015). doi:10.1007/978-3-319-21407-8_27
20. Tasso, S., Pallottelli, S., Laganà, A.: Mobile device access to collaborative distributed repositories of chemistry learning objects. In: Gervasi, O., et al. (eds.) ICCSA 2016. LNCS, vol. 9786, pp. 443–454. Springer, Cham (2016). doi:10.1007/978-3-319-42085-1_34
21. Olteanu, C.: Learning Management System and Shareable Content Object Reference Model. Manag. J. **6**(1), 106–109 (2007). Faculty of Business and Administration - University of Bucharest
22. Drupal, January 2016. http://drupal.org
23. System requirements, January 2016. https://drupal.org/requirements
24. Ahn, J., Lin, X., Khoo, M.: Dewey decimal classification based concept visualization for information retrival. In: CEUR Workshop Proceedings, vol. 1311, pp. 7–14 (2014)
25. 1484.12.1 IEEE Standard for Learning Object Metadata, March 2017. http://ieeexplore.ieee.org/document/1032843
26. McClelland, M.: Metadata standards for educational resources. Computer **36**(11), 107–109 (2003)
27. PHP Threads, March 2017. http://php.net/manual/en/book.pthreads.php
28. PHP Thread class, March 2017. http://php.net/manual/en/class.thread.php/
29. Introducing JSON, March 2017. http://www.json.org/
30. March 2017. http://www.quora.com/What-is-syntactic-sugar-in-programming-languages
31. PHP Manual-MySQL Extension, March 2017. http://php.net/manual/en/book.mysqli.php

http://www.springer.com/978-3-319-62397-9

Computational Science and Its Applications – ICCSA 2017
17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part III
Gervasi, O.; Murgante, B.; Misra, S.; Borruso, G.; Torre, C.M.; Rocha, A.M.A.C.; Taniar, D.; Apduhan, B.O.; Stankova, E.; Cuzzocrea, A. (Eds.)
2017, XXXVI, 766 p. 219 illus., Softcover
ISBN: 978-3-319-62397-9