GlorepSync IO

1. Php ZTS – why implementing it?
2. MySql
3. Socket
4. Messages
    1. FileMessage64
    2. TextMessage64
5. Readers
    1. Reader64
    2. ServerReader64
6. Writers
    1. Writer64
    2. FileWriter64
    3. TextWriter64
7. DBConnection
8. Sync
9. Server.php

Php ZTS

Php ZTS is an easy to use, quick to use learnreading API for PHP7.
It allows you to execute any and all predefined and user declared methods and functions, including closures.
Ready made synchronization is also included.

For PHP7, pthreads has been almost completely rewritten to be more efficient, easier to use and more robust.

Supported versions

Pthreads v3 requires PHP7 or above.

Windows support

Yes, pthreads v3 does have Windows support. It is offered thanks to the pthread-w32 library.

Windows releases can be found at http://windows.php.net/downloads/pecl/releases/pthreads/

It is very important to note that this release does not use forks in order to simulate threads in php, the threads that will be created will be true threads that are compatible and safe with php.

For more information please visit http://php.net/manual/en/class.thread.php/

MySQL

This project uses MySQL.
MySQL is an open-source relational database management system (RDBMS).
MySQL's triggers functionality is necessary for proper synchronization of the metadata we're using.
GlorepSyncIO uses triggers to detect events and to synchronize the local database with the remote database.

Socket

Sockets are currently not a core part of GlorepSyncIO.
They are, however, a mean fo a future implementation of RPC listeners.
As it is right now, our daemon (GlorepSyncIO) is passive, it executes the routine once every N seconds. This way of proceding might not be enough. In some chases we might want to directly query the darmon 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).

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 File64 class.

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.
Te reason behind storing a base64 string is quite simple: before sending our message, we want to wrap our file inside a json object, so we can send some side information about our file, such as the file's name, size,and type and we want do 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 hased using the unicode standard.

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

[…] &ÖS¥ýC´¤î#^#¤"#úP#Ü'❺Ë[@#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.
Here's 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	

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 wild characters issue. 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.

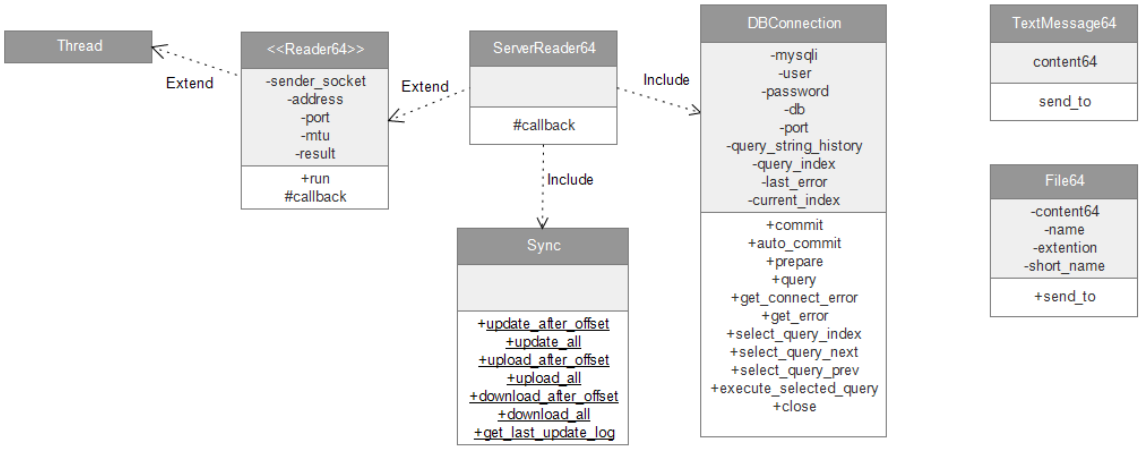
Readers

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 body of a method called “run”. Reader has a defined construct, it requires a socket (which will be used to read the incoming data) and a Maximum Transimission 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 untill there’s nothing left inside the buffer.
Once the message has been read, it’s contents encoded in base64 will be passed to a abstract callback method, along with the sender’s IPv4 address and the port of the sending application.

NOTE: The message that is being received at this point is nothing more that an base64 encoded string. Before passing the string to the callback method the string is decoded into a Json Object, so there’s no need to decode it afterwards.

Reader64 is an abstract class, thus, it cannot be instantiated, hence ServerReader64 is needed. ServerReader64 extends Reader64 and it defines the body of the callback method, which is named “callback”, it provides the 3 parameters that has been passed along in Reader64 after the message has been read: the **message**, the **IPv4 address**, the **port number** of the sending application.



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 FilewWriter64 classes, as they both extend Writer64.

TextWriter64 & 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” when sending data to the daemon. They are by no means required, in any situation, but they do help out 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.

DBConnection

DBConnection class does not extend **mysqli**, 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 query 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.

NOTE: 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 be 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**".
NOTE: 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**".

Available methods

- commit()

Commits the current transaction for the database connection.

- autocommit()

Turns on or off auto-commit mode on queries for the database connection.
To determine the current state of autocommit use the SQL command *SELECT @@autocommit*.

- prepare(**string $query**)

Prepares the SQL query, and returns a statement handle to be used for further operations on the statement. The query must consist of a single SQL statement.

- query(**string $query**)

Performs a query against the database.

- get_connect_error()

Returns any error during the connection attempt.

- get_error()

Returns the error message from the database.

- select_query_index(**int $index**)

Selects a specified query index.

- select_query_next()

Increases the query index by 1.

- select_query_prev()

Decreases the query index by 1.

- execute_selected_query()

Executes the current selected query, regardless if autocommit is set to false or true.

- close()

Closes the connection with the database.

Sync

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

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, there is one more step this method executes 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 delete regardless the cirumstances**.

- 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** 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 **insert event** on the general table and insert a row in update_log afterwards, using the id_lo of the new general record for the local_id attribute of 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 row fetches the relative object from the general table, if the object's status is "draft" or the object's id_fd is not the same as the local federate's name then delete the row and skip to the next one.
Otherwise the object will be uploaded to the shared database and the record in update_log in the local database will be deleted.

Server.php

The main file of the entier 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.

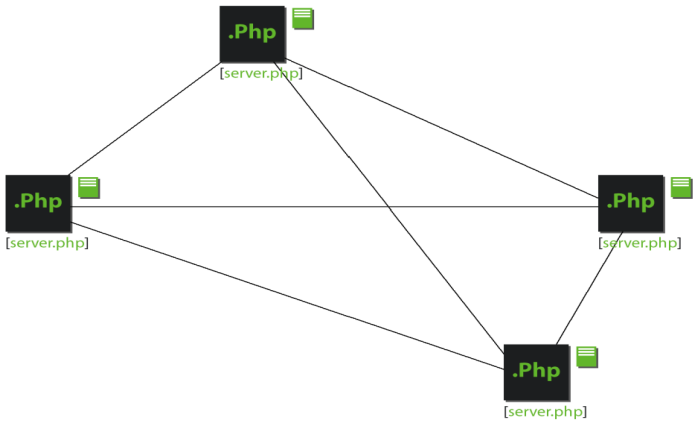
Before starting to talk about the mechanisms that are behind everything, we must first understand exactly what is the exact problem weìre trying to solve at this point.

We have a situation where a variabile 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 gather 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?

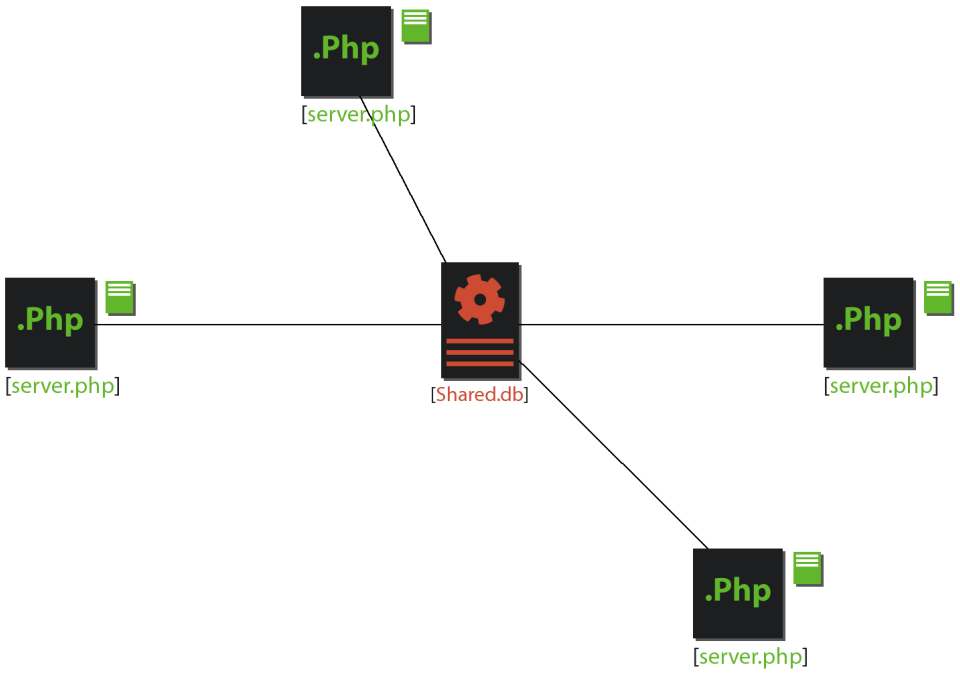
Proposition

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 collaboration.

This implies other problems, such as allowing every server to detect and distinguish between the two events of leaving and joining the collaboration.
The resource cost of this method is also directly proportional to the number of servers of the collaboration, 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 comunication, a central database could serve as an easier way of comunication between servers.



The central server is a database, nothing more. We will be calling that server “shared database” from now on.
We will also be refering to the database of the server (it doesn’t matter which one of the servers)
In this mechanism each server has its own database stored locally, and the shared database also has an empty copy of the database.

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.

The only objects the daemon will upload to the shared database are the ones belonging to the local federate: **owned objects**.

On the other hand the local daemon must only download metadata which does not belong to the local federate: **delegate objects**.

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 “behind 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.

IMPORTANT : Note that this mechanism implies that the local database could be both **behind and afead** of the shared database, infact, the local database could contain more owned objects, and less delegate objects.

But 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 **id_fd** which indicates the name of the federate, and **id_lo**.
The id_lo attribute is a numeric attribute that when combined with the id_fd a unique key is obtained, this key identifies the object itself.

NOTE: 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 wheter the local database is behind or ahead, is to compare the two objects.

- **I**f 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 the id_lo of the last known proprietary object from the shared database to Sync::upload_after_offset($offset,$local_db,$shared_db, $federate_name) as offset, will upload every missing object to the shared database.

Downloading

Once enstablished that the local database is behind, the download routine can start.
The dounload routine uses 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.

- **I**f 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 from the local database.

IMPORTANT: 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.

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, GSIO, will retrieve the owned objects referenced in lo_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.

Diritti, autori, ecc...