OpenLSD, OpenLSM and OpenR66 projects

Author: Frédéric Brégier under LGPL

Website: http://openlsd.free.fr/en/OpenLSD.html

History of the project

One of the biggest projects I had from 2003 was about the Management of Document in Electronic format, and to be more specific it was about Documents Archiving and not about content management. In fact, my employer had a goal to store about 2 PB of documents (2 000 TB or 2 Millions of GB), that is to say more than 200 billions of documents with an average size of 10 KB by document.

This project was given to my team in 2003 and in 2004 2 guys came into my team, and specifically M Vincent Castella as the Project Chief in my team. IBM had also greatly participated in our studies.

My employer had made a choice of software (proprietary software) in the beginning of 2002.

Where problems begin

During our studies and development, it appears that this solution has some limitations, like its competitor softwares. Among them, I can quote those:

- There is a limitation concerning the storage space saying how many TB can be managed: most of those softwares were created during the 90's where the storage would not go further than 1 Tera Byte. We have a factor at least of thousand to consider for our project.
- There is a limitation concerning the storage usage saying how many files can be stored in one storage space : again, due to the limitation to 1 TB from one side, and to the usage in the 90's of 32 bytes programming model on the other side, the number of document are therefore limited to some billions. We again have a factor at least of hundred but that could easily tend to thousand to consider for our project.
- There is a real weak point of the security of the document based on the replication on two sites: in the 90's, the data security could be established using tapes or using slow on-line replication. But today, according to the number of documents to save (our estimation is about 1 million of document in one hour in heavy load), the methods of the 90's are no more compatible with those facts. The replication speed we measured is about 4000 documents by hour so we have again a factor of a thousand to consider for our project and the tape capacity are not compatible with the speed of saving and reading (up today).

Moreover we have to face another problem which concerns the file transfer. Indeed, to get this million of document by hour, this implies that over 100 000 files must get in buy file transfer protocol every hour (each file containing about 10 documents). However the secure file transfer protocol software cannot guarantee such volume, again for the same reason they were developed in the 90's.

- We cannot use FTP since it is not secured neither in crypto neither in quality of transfer.
- SFTP, the FTP protocol over SSH, answers yes to the first criteria (crypto) but not to the quality of transfer such as the guarantee the transfer was ok, should it be redo from the beginning or from another check point, what kind of transfer we've got, from who and when, can we set some automatic actions in pre or post transfer operation...
- This is the kind of functions we need, functions that runs in file transfer monitors like CFT software. But those softwares suffer when too many transfers get into the system.

How OpenLSD is born

When I focused on those two problems, I decided to develop myself at home two systems, quite close in conception but really different regarding their usage, completely in JAVA (1.5 at that time) and I have the project to develop a third opus, deduced from the first one specifically to handle the legal email archiving :

- OpenLSD : Software from Document Archiving : Open Legacy Storage Document
- OpenLSM : EMail Archiving : Open Storage Mail
- OpenR66 : Monitoring File Transfer Software : Open Route 66

Overview of OpenLSD

This project has for goal the numerical archive and wants to be generic, that is to say it should not be dependant on the archived data, neither to their format or their index. This project is therefore not a complete solution by itself, but a complete framework where we can specify business by only specifying the type of index and create the associated consultation application. It is framework for archiving and retrieving large volumes of static documents.

This project has the following properties:

- 100% full Java

- Allows to handle in theory up to 2^192 documents split in 2^64 logical application spaces (Legacy), each of them split in 2^64 storage spaces (Storage) each of them containing up to 2^64 documents (Documents)

- Each storage space (Storage) can reach up to 2^64 bytes (16 000 Peta bytes, or 16 millions de Tera bytes or 16 billions of GB)

- In practice, a physical server should not have more than 256 storages (filesystems) for management reasons (talk to you SAN and System administrators to see what they think), so each physical server can handle up to 2^72 documents (4 722 billions of billions of documents). The number of documents in each storage depends on the filesystem limitation and the size of the documents. Generally, a filesystem is limited around 2^54 bytes and if you consider an average of 256 KBytes for each document, then you've got up to 2^36 in each storage/filesystem, so up to 2^44 documents in one OpenLSD physical server (17 592 billions of documents). Now the limit of the number physical servers is up to your money you can invest… Let say you can afford 32 servers like that, then you can have 2^49 documents (562 949 billions of documents).

- Allows to create multiple copies (from 1 to n) on distinct localizations to ensure either the data security (replication) and either the speed of access to the documents in terms of network latency (closeness)

- Allows the use of cache functions (both in read and write) to speed up the read of a document for a final user but also to speed up the import processes of documents inside the system in the case where the user (or the application) is not closed to one of the LSD localization.

- Use a data base (JDBC : MySQL for small CRM up to some millions of entry, Oracle or PostGreSQL for bigger CRM)

- Allows using a Java module in Tomcat (or other servlet software) to implement the reading interface for the documents ready in the CRM system.

OpenLSD is not self sufficient. Indeed, it does not take into account the business specificities (business data) which are used to index documents, but it does give the necessary functions to handle them with some few extra codings. OpenLSD is a framework which must be extended (an example is given in the source) to get the application wanted by the user. Mainly, only one Java class is needed to be written. OpenLSD gives the following functions:

- Index and data instantiations necessary to OpenLSD in a database in several tables (MySQL, PostGreSQL, Oracle : porting to another database software should not take more than 2 days)

- Once the business logic implemented in the database and the external software adapted (should be done in 5 to 10 days), the system is ready

- Import of documents: from the server that gives the physical storage (faster) or from the network (the speed will depend on the network); this import is secured (controlled, duplicated if in clone mode), validated (using a MD5 like algorithm), unique (the identification) and optimized (storage and network).

- Extraction of documents : from the server that gives the physical storage (to realize for instance export for burning on external media), from the network (Java client or J2EE client from Tomcat or equivalent)

- Integrity Control and Repair functions

- Statistical Functions

Concept and Logic

OpenLSD stands for Open Legacy Storage Document, i.e. a framework that enables document archiving in a secure and powerful way and able to handle very huge amount of documents. Document can be of any types.

The main ideas are from me but, as anyone, I could not achieve so much work without the help and great discussion of several guys, and specifically M Vincent Castella, the chief of « generics » (private joke).

OpenLSD, a framework from which you build your specific archive software for your needs

OpenLSD is a framework and is not a software as a final product can be, even if some examples are included. In this logic, the type of document can be of any type since it is the business logic to handle any specific format and to know what to do with it. For instance, OpenLSM (Legacy Storage Mail) is based on OpenLSD and allows the archiving of email from email client software (thunderbird or any others if we can implement something as an interface). OpenLSM has the responsibility to handle the email format. The example within OpenLSD is a simple archive application using one string as document identifier or index and using a web interface (JSP using Tomcat) to get or put documents.

Starting from examples, one can implements his specific archive software, spending times on business approach and not on implementing what OpenLSD stands for.

The main idea is that they are some "Impl" packages that must be adapted to fit the business needs. This construction enables fast implementation from current example. One can of course modify, extend or create a new implementation from the example.

Multiple Clients-Server approach

OpenLSD has two parts, server and client, but cannot be reduced only to a client-server approach.

OpenLSD can be used in a strict server-client protocol, but also in a web services approach (such as using the JSP examples) or in a distributed approach, with or without centralization of the documents. It can be used for small business as for big entities as it was intend to be able to handle huge amount of documents and connections but also it was done as simple as it can be.

Server on Archival process

The server part does nothing with the business properties and is only responsible to store document and to retrieve them when asked. The server only knows the three indexes: L, S and D (Legacy, Storage and Document), each of them in 64 bits so 2^(64*3) as a maximum number of documents in one OpenLSD Service. The size of each Storage can be up to 2^64 bytes, so each Legacy can be of size up to 2^128 bytes. One OpenLSD service can handle in theory up to 2^192 bytes and up to 2^192 documents. Every action is performed after receiving a communication using the NIO socket MINA framework.

The server part has no direct relation with the business and so it does not have any relation with the database. This is intend to enable an efficient implementation, stable and is not supposed to be modified whatever the business logic that the clients will implement. All indexes are handled by the clients, using a database to enable unique index, to handle the storage capacity and to allow business access and logic.

Three levels: Legacy, Storage, Document

The logic of archiving in OpenLSD server is using three levels:

1. Legacy – This is the highest level, each Legacy can be considered as one Business archive application. Each Legacy is composed by many Storages (up to 2^64 storages by Legacy). For instance, archiving emails and pictures can be stored in two different Legacies.

   Another example, if two groups of people want to use different storage and business logic, they can use two different Legacies.

   Each Legacy has properties such as a specific crypto key (if any) to encrypt documents on storage, or the size of each storage, or the status as open or closed. One OpenLSD Server can handle up to 2^64 Legacies.

2. Storage – This is the medium level, each Storage should be considered as one storage space. Each Storage includes many Documents (up to 2^64 documents by Storage). In general, a Storage is one filesystem

with the highest size that a filesystem can be. Size of filesystems are allowed up to 2^64 Bytes.

All storages from one Legacy will have the same limit of size for the underlying filesystem. The size is accurate from the document sizes modulo the size of one block in the underlying filesystem implementation (4K by default but can be changed) but it does not take care about the extra bytes when the file is stored using a crypto function.

So the size defined inside the Legacy must take care of the average file size that are intended to enter the system, usually defining a size 5% less than the real size of the filesystem should be sufficient.

For instance, let say one filesystem is 16 Tera bytes (2^44 bytes), then in the Legacy, the size should be around 16 Tera bytes in decimal order (so 16*10^12).

3. Document – This is the lowest level, each Document can be of any type and any size (limited to 2^64 Bytes, which can not really be considered for now as a limit ;-)).

Client on Business

The client part is the only one to know the business properties (for instance, what to do with documents, who is able to see one specific document, how can we find or import one specific document using which index). The proposed framework client is using a database as permanent data storage for index and other properties.

OpenLSD is written using JDBC and was tested with PostGreSQL, MySQL and Oracle. Of course, the biggest the number of documents to be in the system, the more robust the database must be. As for our own tests, Oracle was the more stable and efficient. PostGreSQL should be close (at least for the less than 1 TB database). Our tests with MySQL show MySQL was OK with small and medium office size, but not with biggest office (more than millions of documents and thousands of final users). Of course, any specific business can have its own conclusion!

In fact, there are several types of clients:

- Importer or auto-importer: those clients know about the business, the database and how to communicate with the OpenLSD server (or even several servers, see later). Importer can work both in local way (as usual archival software) and through the network.
- Get (using Tomcat or simple java batch): those clients know about the business, the database (read only generally) and how to communicate with the OpenLSD server (or several servers) in read only.

- Admin: those clients handle some specific functions as starting or stopping some or all OpenLSD services.
- Check: those clients handle some check as consistency check (database and OpenLSD consistency) or some connections checking functions.
- Check Similar: this client handles a check of similarity of a new document not already imported to see if one document in one specific Legacy is already imported elsewhere based on a binary comparison.
- Advanced services: here we can found some new services, as the one we plan as the cache function. The cache function would have effect both from the read point of view and from the write point of view, enabling low bandwidth network to work correctly with the system. It will allow also restricting the database access to a secure point and not in a distributed way.

Security aspects

The security of the system is partially taking into account by OpenLSD by using different protocol and technical aspects. For example, the user authentication is not on the OpenLSD area because it is on the final software responsibility to implement this kind of security. Here are the security points that OpenLSD take into account:

- The message passing is using a proprietary protocol in order to not open directly the access (read or write) to the documents for the final users. The physical storage is completely handled by a single (multithreaded) java process, ensuring security of access (read or write) through the OpenLSD protocol. Therefore, each access to the documents is controlled by the OpenLSD system.
- Documents can be stored using a crypto logic. The key is relative to the Legacy and can be different for each Legacy. This encryption ensures that if someone access to the physical files, it will not be able to read the document natively.
- A MD5 key is computed for each document and is stored in the database. This ensures that the document inside the system is still the same than the one that was inserting in the beginning. Of course, this security depends on the database security also.
- Security of storage (replication to protect on physical corruption) can be done using several ways:
  - A physical mirror between storage unit (example using asynchronous mirror between disk storage): the main advantage of this solution is the easiness of the implementation, the main disadvantage is the price (should be the network link or the software part of the disk storage). The propagation of user's mistakes (such as one deletes a directory, then implying the same delete of the same directory on the mirror part) should be minor as it is unlikely that one user can access to the disk storage and moreover with the ability to delete anything. Generally, this kind of mirror is limited to one copy only.

    - An application mirror between two instances of the same application: the main advantage is that the application is fully responsible of the replication so of the consistency, the main disadvantage is the potentially high latency to be introduced inside the application in order to validate the insertion of the document in both applications. If the main application wants to trace everything, this should be the best choice since the application is responsible of everything so it knows what it does.
    - An OpenLSD mirror between two instances of the same Legacy: the main advantage is that the application doesn't have to take care about the replication since it is taken by OpenLSD, the main disadvantage is the potentially hole of replication since OpenLSD will give its acknowledgement of insertion after the document is inserted inside one Legacy among those existing, thus implying that the replication is not finished yet but in progress. If the replication process is not mandatory in the trace, then it should be the best choice.

  One of the advantages of the mirror is that it enables main applications to access documents through the closest and ready repository. OpenLSD is able to handle up to 2^64 replication hosts.

- Some functions have specific security techniques such as key or password checking.
- Even if the legal people tends to say to keep everything in archive, such that one will never delete a document theoretically from the system, considering very large archive database, we cannot assume to store everything for eternity (at least with the current technology). So we implement a solution to delete a document when its time living is over but in secured way (using key or password checking, TCP/IP security such as filtering and MD5 double checking).
- The database is also to be taken into account in the security aspects. Your Database administrator should help you to improve the security and the reliability such as using a database replication, whatever the way you use. For OpenLSD, the necessary right are very simple (select, insert, delete, update rows, truncate table – on one temporary table only - and execute procedures on the OpenLSD schema).

Secure and efficient deleting document support

Considering the last point (deleting documents), in the early 90's, deleting a document from an archive system was considering as unacceptable. But now considering huge systems, most of the people cannot afford to store everything forever. It does not mean that keeping forever a document in OpenLSD is not possible. In fact, by default, this is the normal way. One document will never be deleting from the system if a specific action is not taken. This specific action is based on a triple check: first we can set a TCP/IP filtering, then a key or password protection, and finally it checks the given MD5 according to the real MD5 of the file. Once everything is ok, the file is deleted.

One can say, why all of this about simply deleting a file?

Well, if you recall, we take as example huge amount of documents. What if the system brings in huge amount of documents every day and takes out (deletes) a relatively close number of documents, for instance archive with 6 months of living status? All archival systems from the 90's will have one big problem: the internal index will not be able to handle a long term of the system living since their indexes are always increasing. To be more explicit, take the following numbers as an example:

- 2^24 documents enter the system each day (16 millions a day, 700 000 by hour)
- Every 6 months, the documents are deleted, so there are at most 2^24 * 6 * 30 equals around 3 billions of documents in the system.
- The internal index is based on 32 bytes, so able to store more than 4 billions of documents, which is superior to the number of documents that should be stored.
- But after 2^8 days (256 days so less than one year), the index reach its maximum value (2^24*2^8 = 2^32), so the system is KO.

So what OpenLSD proposes is to take care about this deleted index and to re-use them when one wants to insert a new document.

Keep storage low in usage

A side effect, but as important as the first goal to handle correctly the index when deleting a document, deleting a document will free some storage space. OpenLSD will try to fill as much as possible the free spaces and not asking for a new storage so as to keep your investments in disk storages as low as possible.

Technical aspects

File Storage Interface

The file storage interface is done through the Legacy, Storage and Document classes. The Legacy can be crypted and controls therefore the crypto key. The Legacy defines two paths:

- base: where the Legacy takes its place for its Storages and Documents
- outbase: where the Legacy will store copy of files when the COPY function will be called

Each Storage should be considered as mount point of an independent filesystem. So the Legacy is a multi-filesystem aggregation.

The size of each Storage in one Legacy must have the same size. Storages are created automatically when it is necessary. **However, even if they would be automatically created when needed, The Storage administrator should handle filesystems creations and mounts and so LSDStorage creations using the LSDAdmin function (or Web interface)**.

Warning: **The size in Legacy must be less than real size since encryption should add some bytes**. In general, depending on average files size and crypto mode or not, 5 percent less than the real size should be enough.
OpenLSD takes care of block size in filesystem but not of crypto block size, but later it could be if we think it could be usefull.

Network Protocol

OpenLSD framework message passing concept is based on serialization of objects between clients and servers. One shall say it could be inefficient but as OpenLSD uses a NIO framework (MINA), it can handle serialization / deserialization with high performance and bandwidth (please see our benchmarks) using non blocking socket and a multi-threading approach. Moreover, this enables also a secure way to access to the Legacies, the front entry of the system by not allowing access them by standard protocols such as ftp, http, ...

It does not mean that ftp and http can not be used as client. In fact, the application example shows how to do using servlet to store or retrieve document from HTTP protocol. But this is done by something we can call a «LSD proxy». The Tomcat server (for the example) implements a proxy for OpenLSD using a HTTP client from one side (get or put) and LSD tunnel to LSD server on the other side.

From version 1.0, OpenLSD supports an HTTP connector for short functions on OpenLSD Server and OpHandler Server (Statistics, Shutdown):

- Info on Sessions : shows some numbers statistics on global sessions by Handler
- Info on Storage : shows some statistics on Physical Storages (short description) (OpenLSD Server only)
- Full info on Storage : show more detailed statistics on Physical Storages but could be very long (depend on disk speed access and number of files) (OpenLSD Server only)
- Clean : force the JVM to run garbage collection dynamically (on OpHandler, clean also the Pool of connections)
- Shutdown : shutdown the OpenLSD Server or OpHandler Server (protected by a password)

With the option CSV, the data are returned in CSV format.

Message Protocol

Each message is a command (client to server) or an answer (server to client). Each command can be of two types: (**unique** field in message or session objects)

1. Unique: only one command and one answer will be done during this session. After the command and answer are done, the session will be immediately and automatically closed.
2. Non Unique: multiple commands and answers can occur, so the session must be closed explicitly or will be closed after a time out of inactivity occurs.

Some message can be multiple blocks based, such as document sending or receiving protocol. For instance, when a document is bringing through the network, it is split according to the size of the document (**fileblocksize** field for the size of one block, **rankblock** for the current block rank, **bytesblock** for the current block of bytes in message or session objects, and **filesize** for the global file size in the session object). For instance, a document of 1MB could be split in 64 blocks of 16 KB each. The size of one block is a parameter of OpenLSD. Some messages have no need of multiple block support (such as administration message to stop or start a service).

For each multiple block message, two implementations have been made:

1. Acknowledging protocol: each block is to be confirmed prior to get the next block. The advantage of this protocol is that we can limit the memory usage at one block at a time by session. The disadvantage is that if the network is slow or if the latency is quite high, the performance can be limited to the network aspect.
2. Non-Acknowledging protocol (NAck): only the first and the last blocks are acknowledge, so the sender sends all of the blocks without waiting the acknowledge of the receiver. The advantage is the performance shall not be limited by the network but now by the memory since one file can be fully in memory when sent or received, which can be a disadvantage.

However the memory usage is for a short time only, just until the block is consumed by the receiver or really sent by the sender.

I implement a final blocking function that wait until the asynchronous underlying command is done (**waitForAllBlocks** and **endedAllBlocks** in session objects). This one is needed to finalize some actions (client side mainly).

For importer local to the OpenLSD service, I implement a specific message that does not transmit the file to be imported but only its full access path. This implementation is obviously the most efficient. Nevertheless, importers from network have also good performance but limited to the network bandwidth.

A KeepAlive filter has been inserted in the message protocol (inspired from Mina's KeepAlive filter) which allows maintaining connections even on long term (Web pool of connections, OpHandler pool of connections …). However after a long time of inactivity (around 10 minutes), the connection is closed in order to not maintain very long term connection.

Database relative

OpenLSD uses a JDBC interface to access to database. In order to be efficient, after making some tests, it appears that using some big framework such as Hibernate could be a problem in applications that do not already used Hibernate objects. Moreover, the performances were very bad when considering specific tasks compared to « manual SQL code ». So my implementation uses direct JDBC connection and not object for persistence. However, I decided to make class for each kind of objects, such as Hibernate objects, but using my own underlying implementation (**LSDDb**X classes). By this way I was able to introduce a special class (**LSDSpecific**) where I put specific functions according to the underlying database really used. Indeed Oracle, PostGreSQL and MySQL do not have the same specific SQL abilities, neither the same performance according to the way one write the SQL code. So for very specific codes, I made this specific class which calls specific subclasses according to the current used database (**LSDMySQLSpecific** and **LSDOracleSpecific** and **LSDPostGreSQLSpecific**).

Specific Java problems

I had to face also some specific and quite odd problems in Java.

One was that when you start to work in 64 bytes, you assume that everything is ok with size up to 2^64, but it isn't. Almost all objects in Java (even 1.6) are limited to 2^32 bytes long. Specifically I had to create a List that allows more than 2^32 objects in it, the LinkedLongList. However I try to limit as possible its usage since the problem would be the memory (only to get the list of Storages for one Legacy). However, at the time I am writing those words, I suspect that it is not a problem since having more than 2^32 real servers is probably not a real common fact…

I create a Lock that is not reentrant (contrary to the ReentrantLock of Concurrent package) meaning that if the same thread tries to lock it again, it will block. Of course, this is OK only if another thread has the right to unlock it from elsewhere (if not, it brings to a dead lock!). Perhaps it exists something better to do that, but the ReentrantLock was not the good option.

Another problem was when you launch an external process from a Runtime.exec command. The API was not really well documented but now there is at least a warning (as of the 1.6 version) in the Process API:

« All its standard io (i.e. stdin, stdout, stderr) operations will be redirected to the parent process through three streams (getOutputStream(), getInputStream(), getErrorStream()). The parent process uses these streams to feed input to and get output from the subprocess. Because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block, and even deadlock. »

What does this means? Well, if you launch a process but you don't care about its ErrorStream for instance and so you don't try to read anything from it, you likely will have a chance to block the subprocess and even the main JVM, so having a deadlock. To avoid this, I use a specific thread to read from error stream and input stream (read anything without action), the **LSDStreamGobbler** class. Some few web sites present quite the same code as it is inspired from several blogs or forums. To use it, when you don't care about outputs of the subprocess, just call the **waitForProcess(Process p)** static function. If you take care about the InputStream, then create a new **LSDStreamGobbler** object with the Process as parameter and start the new created **LSDStreamGobbler** thread.

Optimizations

I try several ways to optimize as much as I found, with the help of Vincent and also Brice Carriere Montjosieu.

One of course was to use as much as possible the efficiency brings by the NIO support of MINA framework. Non blocking socket protocol was a good way to implement with a few threads a scalable server for OpenLSD.

Another one was to optimize as much as possible the SQL request, and even to create procedures inside the database so as to be as efficient as possible.

And finally, I try to make OpenLSD as general as possible. I cannot say that this is a final version or even a complete version since I have some other plan of improvements but I think it was time to bring it to the open source community and let this project lives free (free as free beer too !).

Future plan

I plan several improvements. The order is not defined since I change my way according to my will. But here are some of them:

- Implements all functions of import, get, check and admin for Multiple OpenLSD support. Right now, import, delete, check and admin are supported on Multiple OpenLSD support, that is to say a mirror of the documents handled by OpenLSD for each Legacy defined inside the Database.

  What is ready: Import, delete, check and admin for ML OpenLSD support.

  One goal is to allow of course the replication and therefore the security of the archiving process.

  Another goal is to allow more availability of the concerned Legacy by allowing one user to ask documents from any OpenLSD servers that implements this Legacy in parallel or in the reverse way to import documents to any OpenLSD servers that implements this Legacy in parallel. These two last points are not yet implemented but are not difficult to do.

- Create a full cache support, both in read and write. The main idea is that a user can use a client that communicates with a local (or close) cache OpenLSD server.

  If the user asks for a document, this document will first try to be retrieving from the local cache, and if not present, it will be retrieve from one of the OpenLSD servers and store also in the cache server for a next request until a certain time. The storage in the cache server is of course less than the OpenLSD Server and is supposed to be up to 2^64 documents, but we recommend to limit it to a number as small as possible. Perhaps some optimization will break up this limitation. The storage can be also crypted (using a different key than the Server one) and documents are stored for a short time only (there will be a parameter standing for valid hour delay like a web cache).

  If a user (or program) ask to import a document into the system, the cache will be as a write cache, that is to say it will be first written to the local (or close) cache OpenLSD server and immediately returns the control to the user (or the program). Then the import of the document is done in an asynchronous way between the cache server and the final OpenLSD server (or one of them in case of multiple OpenLSD servers).

  One of the interests of the caching service is obviously to improve network latencies between the main system and the users.

Another interest is that for the import part it will allow to not give the access to the central database from any client. Indeed, the cache server will act as a simple cache for the final process of import of the document. The only process to access to the central database will be the final importer process and not the cache or client process.

- Improve OpenLSM (email archiving based on OpenLSD) since its first version is running OK but should have some improvements on business part... ;-)
- And of course continue to clean the code, refactorize when needed and debugging any bug that I didn't found yet...