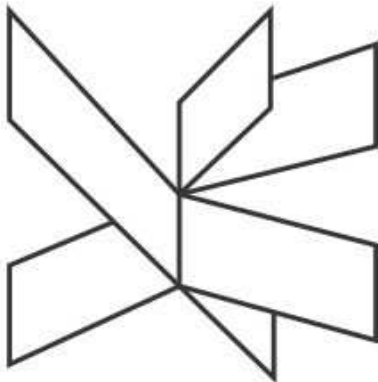


Centralized Library Management System



VIA University
College

2. Semester, February 2018 - June 2018
30326 characters

Supervisors

- Ib Havn (IHA)
- Joseph Chukwudi Okika (JOOK)
- Knud Erik Rasmussen (KERA)
- Mona Wendel Andersen (MWA)

Group 5

Angel Petrov	266489
Daniel Bergløv	220951
Diyar Hussein	266352
Kenneth Petersen	269379

Abstract

The purpose of this project, was to develop a library system that offered some improvements in certain areas, in which the competitor products on the market lacked.

The analysis showed that a problem with many of the existing products was their incompatibility with each other due to proprietary system designs.

The delivered service was designed to be open for third party compatibility and to rely on other open third party resources to improve its functionality. A central feature is the ability for customers to use software from any developer to communicate with the server. Access to and use of the server are the selling product. The default client software is shipped for free and is open-source.

The implementation was done in Java and the communication between client and server relies on java's RMI package. Ultimately this does limit the openness of the product. A future improvement would be to change this communication to a different set of protocols.

Table of Contents

Abstract	1
Table of Contents	2
1. Introduction	3
2. Requirements and Backlog	4
2.1 Functional requirements	4
2.2 Non-functional requirements	4
3. Analysis	5
3.1 Use-case Modeling	6
3.2 Activity Diagrams	8
3.3 Model Class Diagram	10
3.4 SWOT Analysis	12
4. Design	13
User Privileges	13
GUI Separation	14
RMI connection and interfaces	15
Sequence diagram	16
5. Implementation	17
User Authentication	17
RMI and ServiceManager	18
6. Testing	20
White Box testing	20
Black box testing	21
Test conclusions	24
7. Results and Discussion	25
8. Conclusion	26
9. Project future	27
10. References	28
11. List of appendices	29

1. Introduction

Currently the market for library management systems is dominated by a few companies, who all have the problem of being proprietary and incompatible with other solutions or systems. This offers limited flexibility for the customer and makes it difficult to tailor the system to the needs of the individual client.

Libraries on the market have faced immigration problems, when changing from older systems to newer systems, which in part have been caused by lack of openness in the data formats and other proprietary restrictions.

2. Requirements and Backlog

2.1 Functional requirements

- Global administrator can notify about updates
- Loans can be extended by users
- Global administrators can do everything that librarians can but universally, instead of for on library.
- Local administrators can add librarians to their library.
- Users can manage their own profile.
- Librarians can list past due loaners.
- Users are warned about due books when trying to rent.
- Global administrators can add/remove clients.

2.2 Non-functional requirements

- The system should use a database for data storage.
- The system should have graphical interface.
- Requests should use some sort of validation.

3. Analysis

In a time where the world has been connected through various technologies, such as the internet, which allows people to share information across world boundaries within seconds, libraries need a way of centralizing their data, such as inventory and users, and need a system that can be handled centrally.

Previously, libraries have faced the problem of utilizing different proprietary systems, which were incompatible with one another. This led to problems when trying to transfer information from one library to another and hindered centralization. (Version2, 2016)

Currently, there is already a system on the Danish market called Cicero, made by Systematic, that tackles most of these problems. The centralized part of the software provided by Cicero uses a closed proprietary data model which decreases compatibility. In essence, any user relies on the systems that are provided strictly by Cicero. The majority of Danish libraries are already using this system and as such it will be more difficult to get in on this market. However, this system is not without its problems. (Version2, 2016)

Today, another commonly used Integrated Library System(ILS) is KOHA, which has been on the market for eighteen years and can therefore be referenced as a long standing product, that has proven itself over time in a practical perspective to the end user. (Ojedokun, A, Olla, G, & Adigun, S 2016)

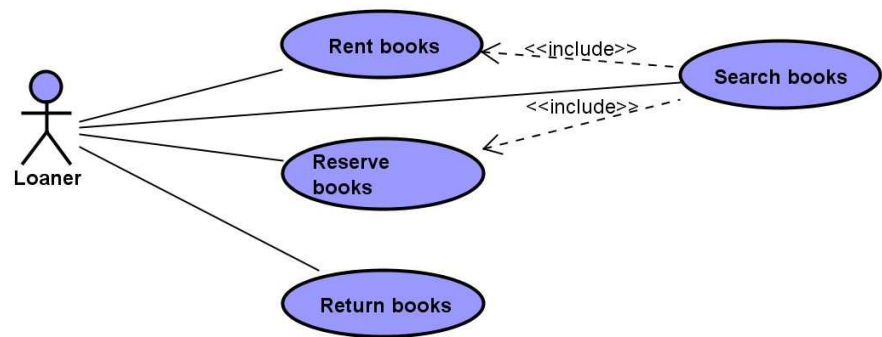
KOHA is a server-less system, in which the clients has to do all database management themselves, which in turn limits the project's scope and does not allow for any centralization. Furthermore KOHA is implemented in a way which makes it platform dependent and requires installation work from the client. (Statisticstimes.com, 2017).

Advantages to KOHA is that it is open source and free. A possible upgrade which can be put into practice, in comparison to the feature set provided by KOHA, is the inclusion of a server side which is provided for the end user to set up as they see fit.

Another system similar to KOHA is Evergreen. It has largely the same disadvantages as KOHA, offering no service, but only a stand-alone software. Being written in C and Perl, it is also not platform independent, without some work from the client themself.

Exlibris Voyager is a client-server payment service, with a free open-source client, using open standards to send and store data. It is used by some large institutions in the US, such as the Library of Congress and has existed since 1986. (Librarytechnology.org, 2015)

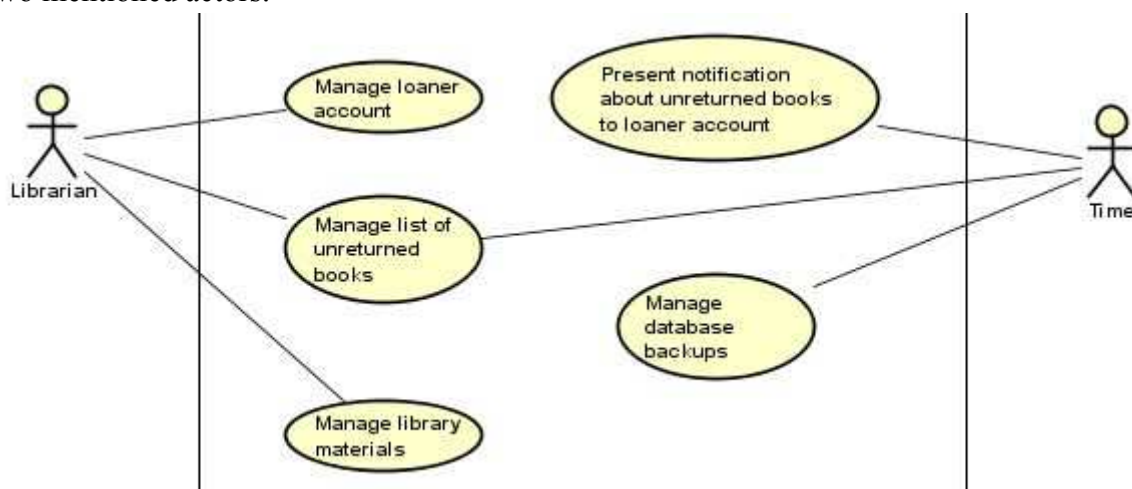
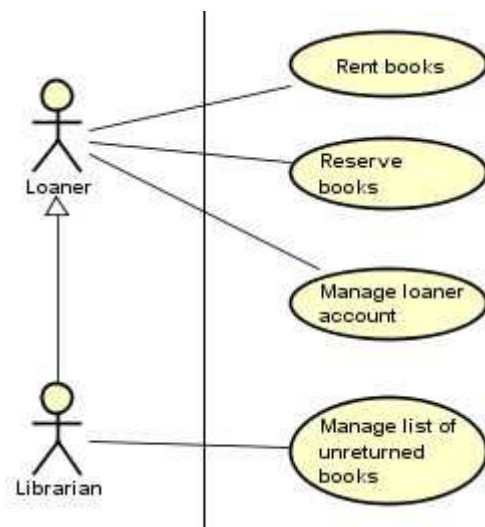
3.1 Use-case Modeling



The Use Cases are divided into groups that are accessible by specific actors. Some actors extend other actors, allowing them access to the same Use Cases. For example, the Administrator actor shares Use Cases with the Librarian actor, and the Librarian actor extends the Loaner because the Librarian actor is an extension of the Loaner and should be able to access all Use Cases associated with the Loaner.

- **Actor Description**

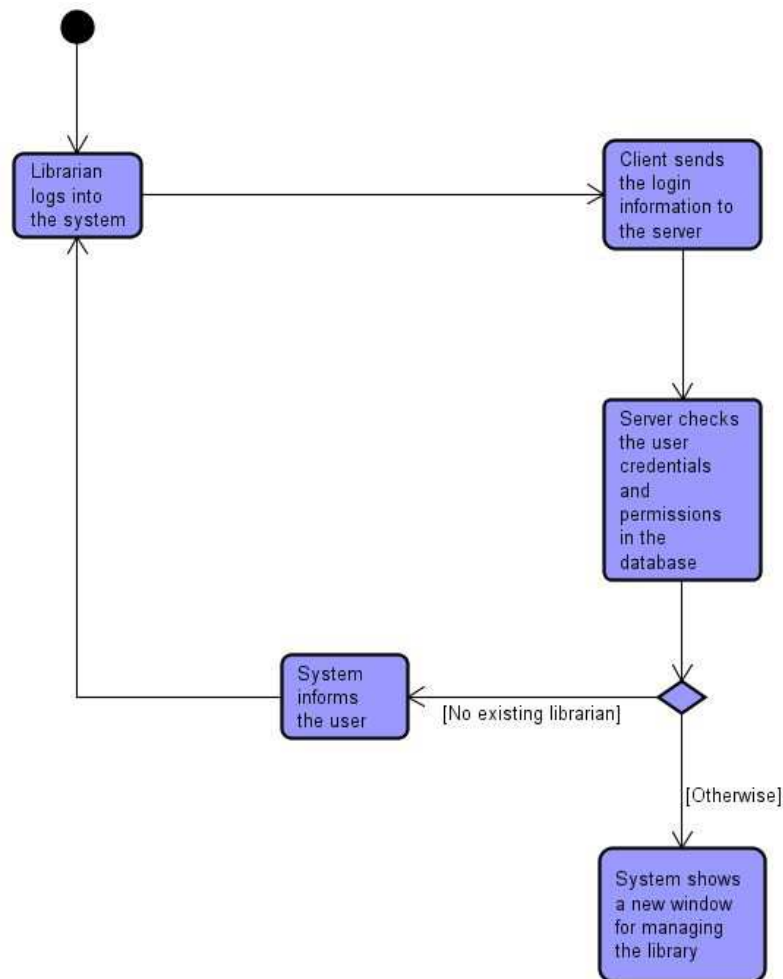
The system is a multi-user system with several actors such as Loaner, Librarian, Local/Global Administrator. Each actor has their own roles / access within the system, while inheriting from actors with lower ranks in the hierarchy. For example, the Global Administrator will inherit roles / access from the Local Administrator which in turn inherits from the Librarian who is in turn lower in the hierarchy than the other two mentioned actors.



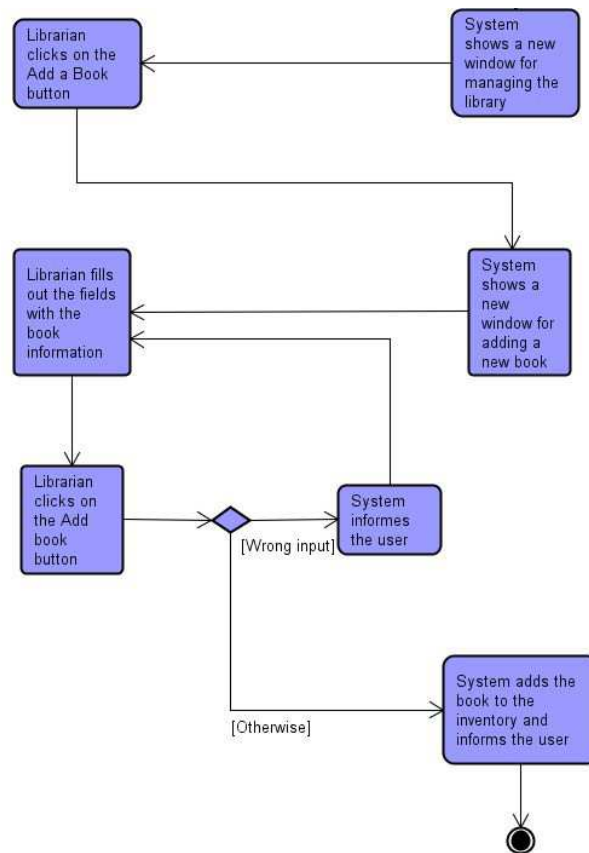
Time is also an actor in the system. It takes some roles like notifying the users about the borrowed books before the deadline for loaning period and adding the unreturned books to a list with the loaner number which could be used to send delaying fees to the loaners later.

3.2 Activity Diagrams

The following activity diagram shows the interaction between the user and the system when doing a specific task, in this case: adding a new book to the library.

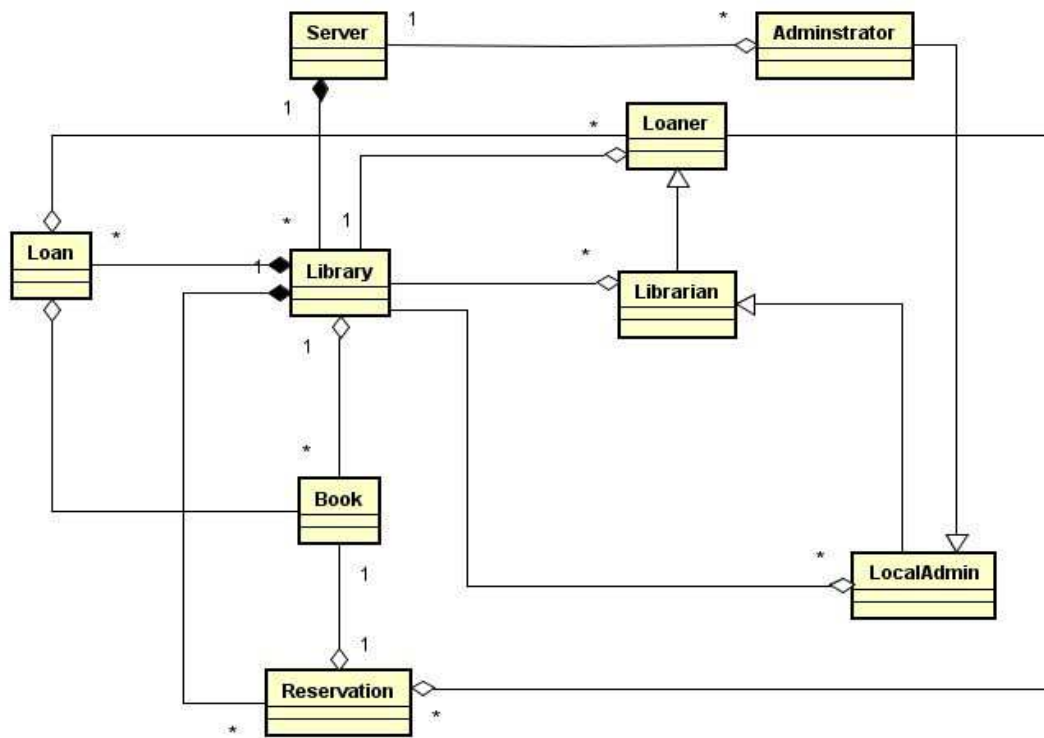


This is the first part of the diagram. The activity starts with a login on the client side, the client then sends the information to the server upon login request to check and authenticate the user credentials and permissions compared to those in the database. If the credentials parsed by the client are incorrect or non-existent, the server informs the client about this and the user will have to repeat the process again. Otherwise, the system will give the user access to administrative sections of the system.



After getting access to the management window, the librarian can have access to certain features like registering a new user “loaner account”, adding, editing and removing books from the library. This diagram shows specifically how the user can add a book, so they will need to click on the “Add a new book” button, the system will then show a new dedicated window for adding a book. The librarian should then fill in all the required text fields with the information of the new book, and then clicks “Add”. If anything is wrong with the input, the system will prompt the user to fill in the fields again, otherwise, the system will add the new book to the library and inform the librarian that the new book has been added successfully.

3.3 Model Class Diagram



- **Server:**
The server with which the client software will communicate
- **Library:**
Represents a single library in the system.
- **Loaner:**
The base user type representing a human user with certain access permissions for the system.
- **Librarian:**
A user associated with one or more specific libraries, that extends the base user type, Loaner.
- **LocalAdmin:**
A further extension of the Librarian user type, with super user privileges for the specific library.
- **Administrator:**
The global administrator who works for CLMS, not for a library. This user further extends the LocalAdmin and has super user privileges for all libraries and system features, both client and server side.
- **Book:**
The representation of a book release with a specific ISBN number. It does not represent the individual physical book, but the archetype of all books with this ISBN number.

- **Reservation:**
A representation of a single reservation for a specific user, in regard to a specific library and Book entity.
- **Loan:**
Representation of a loan made by a specific user, in regard to a specific library and Book entity.

3.4 SWOT Analysis

	Helpful	Harmful
Internal	Strengths <ul style="list-style-type: none">• Scalable• Lightweight (Runs on most hardware)• Open Data Model• No forced hardware bundling• Third party API integration enables an extensive database from the start.	Weaknesses <ul style="list-style-type: none">• No option for hardware bundling• Project targets specific part of market
External	Opportunities <ul style="list-style-type: none">• Currently used product does not satisfy customer• Multiple third party databases are available• Competitor weak in a particular area	Threats <ul style="list-style-type: none">• Established strong competitor already on the market• Difficult for customers to upgrade due to incompatibility with former data model

Table 1: SWOT Analysis

4. Design

User Privileges

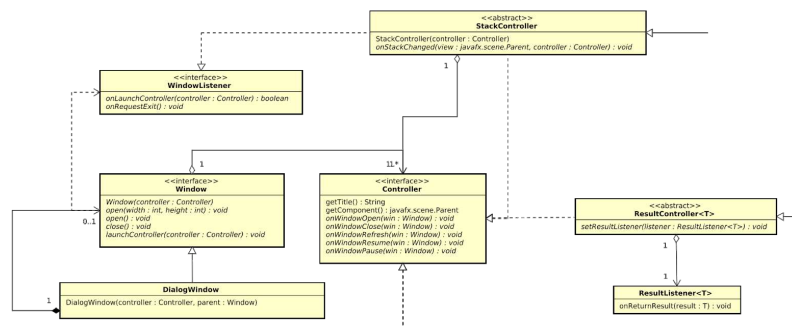


This system needs to be able to deal with multiple actors such as local/global administrators, librarians, loaners etc. To accommodate this and to make the system

flexible for future changes, this system implements role based privileges using bit flags. All roles are represented via single bits using a single integer. This also enables the use of bit masks to check multiple roles at a time. For example the administrator role is a bit mask of -1 according to two's complement, which basically just adds all possible roles by setting all bits to active.

Since the system will also manage multiple libraries, privileges are assigned to libraries. This means that a user can be an administrator at one library while being a meer loaner at other libraries. Global administrators are assigned to a library id of 0, which indicates privileges at all libraries.

Loaners are identified by the lack of privileges.



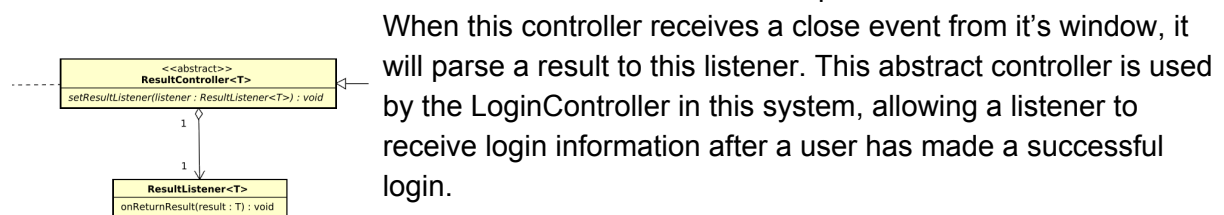
GUI Separation

The system uses an abstraction on top of JavaFX's window features that handles the creation and styling of windows and also internally deals with the required UI Thread.

The actual content of the windows are handled by a window controller that each window requires. On top of that the window abstraction implements simplified lifetime callbacks for the controller which allows them to deal with events such as open/close and pause/resume when windows are being opened, closed, minimized etc.

This separation provides more flexibility by not tying a specific UI component to a specific window. For example two controllers that each open in their own window, can easily be moved to a tapped window without any changes to the components themselves.

The controller interface can easily be extended to support additional features while keeping them compatible with the base window implementation. For example this system has an extended abstract controller named ResultController that accepts a listener to be added.



When this controller receives a close event from it's window, it will parse a result to this listener. This abstract controller is used by the LoginController in this system, allowing a listener to receive login information after a user has made a successful login.

RMI connection and interfaces

The connection between the client and server is based on Java's RMI package. The "bind" and "lookup" functionality is handled by a ServiceManager class, on both client- and server side. When a class needs to make use of an RMI interface, it requests it from the ServiceManager. This way, connection problems and handling of exceptions thrown by the RMI classes, are taken care of internally in the ServiceManager, allowing the class using the service to receive an RMI interface, or null upon failure, in just one line of code.

Further, the server-side has access to an extended number of methods, that the client does not have access to. These methods are not included in the interface available to the client and can only be accessed by casting the service to the concrete class, which is not shipped with the client software. Each method in the interfaces, take a token as their first parameter, as this is the servers way of validating that the specific user has permission to perform the action performed by the method. If not, the method return a negative response value, such as null or -1, depending on the return type of the method

The server supports three different services, in the form of RMI interfaces, which the client has access to. User-service, inventory-service and library-service.

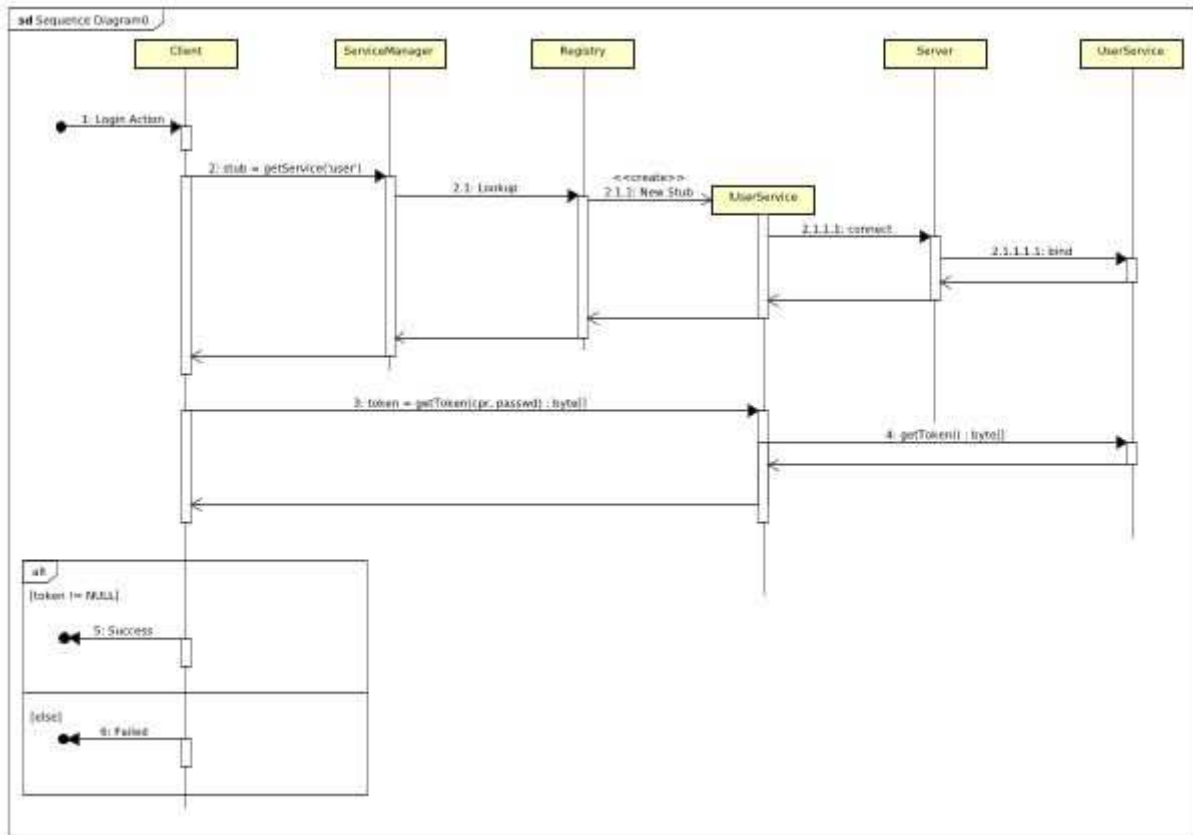
The User-service primarily deals with authenticating and logging in users. The client has the ability to request a token for a user, to check the validity of a specific token or the permissions tied to a certain user. Users with sufficient permissions, as in administrators, can also requests lists of users and add or remove users.

The inventory-service deals with the inventory of the libraries as well as renting an reserving. Like with user-service, the methods that can be used by a certain session on the client-side, depends on the permissions of the logged in user. Basic loaners can use the rent, return and reserve functionality, while more privileged users can also add and remove books.

The library-service is used only as an administrative feature and by the initial login screen. It contains methods for listing, adding and removing libraries. It has no methods that relevant to loaner interaction. Once the loaner is using the system, the client software will already be logged into a particular library.

Finally the ServiceManager can return a DatabaseService, which is an abstraction upon the interaction with the database. This service is naturally only available on the server side.

Sequence diagram



This diagram shows the sequence of an employee or administrator login.

5. Implementation

User Authentication

User authentication is done using a token. The system does not store any password information in the database. Instead a users CPR number and their password is used to create a 256bit HMAC hashed byte array, which is stored

in the database. During login, this token is re-created and used for further authentication during a users session. This will also ensure that a users password is not continuously transferred between the client and server.

```
public static byte[] generateUserToken(Long cpr, String passwd) {  
    try {  
        SecretKeySpec keySpec = new SecretKeySpec(passwd.getBytes(), "HmacSHA256");  
        Mac mac = Mac.getInstance("HmacSHA256");  
        mac.init(keySpec);  
        mac.update((Long.toString(cpr) + passwd).getBytes());  
  
        return mac.doFinal();  
    } catch (Exception e) {  
        throw new RuntimeException(e.getMessage(), e);  
    }  
}
```

RMI and ServiceManager

As explained, the ServiceManager wraps the RMI implementation. When a client-side class needs to use an RMI interface, calls the `.getService()` method on the client-side ServiceManager.

```
public static Remote getService(String name) {  
    try {  
        return REGISTRY.lookup(name);  
    } catch (Exception e) {  
        Log.error(e);  
    }  
    return null;  
}
```

The class calling it avoids having a try-catch statement every time it needs to get a service interface and avoids having to have an internal reference to registry.

Before the client is able to use any methods, it must first acquire a token from the user-service. Further, before a loaner is able to perform any actions, he must login, by which the client acquires a token for that loaner. Both of these tokens are acquired using the `.getUserToken()` method in user-service.

IUserService:

```
byte[] getUserToken(long cpr, String passwd) throws RemoteException;
```

UserService:

```
@Override  
public byte[] getUserToken(long cpr, String passwd) {  
    byte[] token = generateUserToken(cpr, passwd);  
    String tokenStr = Utils.tokenToString(token);  
    DatabaseService db = (DatabaseService) ServiceManager.getService("database");  
    ResultSet result = db.query("SELECT COUNT(*) AS cCount FROM Users WHERE cToken = ?", tokenStr);  
    try {  
        if (result.next()) {  
            if (result.getInt("cCount") > 0) {  
                return token;  
            }  
        }  
    } catch (SQLException e) {  
        Log.error(e);  
    }  
    return null;  
}
```

As the client calls the method with a cpr and password, the service generates a token from these parameters. It then generates a hexadecimal string representation of the token and queries the database for the number of registered users, that have this token attached to them. If the number is 0, the method return null. If it is 1, it returns the token.

Once the client has acquired a service instance and a token, it can use methods in the service, that are allowed for the particular user. When a loaner tries to rent a book, he will be prompted for cpr and password, which will result in the client receiving a token or the loaner. The client is now able to performs rentals for that loaner using the inventory-service.

InventoryService:

```
public int addRental(byte[] reqToken, int lid, int bid, int uid) throws RemoteException;
```

InventoryService:

```
@Override
public int addRental(byte[] reqToken, int lid, int bid, int uid) throws RemoteException {
    UserService uService = (UserService) ServiceManager.getService("user");
    if(!uService.checkToken(reqToken)) {
        return -1;
    }
    Calendar cal = Calendar.getInstance();
    int year = cal.get(Calendar.YEAR);
    int month = cal.get(Calendar.MONTH) + 1;
    int day = cal.get(Calendar.DAY_OF_MONTH);
    LocalDate ld = LocalDate.of(year, month, day);
    long offset = ld.atStartOfDay().atZone(ZoneOffset.UTC).toEpochSecond();
    long length = 31 * 24 * 3600;
```

The client calls the addRental method with the loaners token, the library's id (lid), the book's id (bid) and the user's id (uid). First the token's validity is checked using UserService. Then the Unix timestamp representing the current date is generated. The system does not deal with time zones, since the timestamp will only be converted to a text date format on the client side and a client can not exist in more than one time zone. Therefore, UTC is used as time zone for all timestamp generation, meaning no offset is used, since Unix timestamps are UTC by definition. The duration of the rental is hardcoded to 31 days. 31 is used, instead of 30, to compensate for the fact that the timestamp represents the beginning of the rental date and this inaccuracy should be in the loaners favour.

Finally, after having checked, that books are available, the rental is added to the database:

```
q = "INSERT INTO BookRental (cBid, cLid, cUid, cDateoffset, cDateduration)
VALUES(?,?,?,?,?);";
return dbs.execute(q, bid, lid, uid, offset, length);
```

6. Testing

Testing on the system has been conducted in two different ways: white box testing and black box testing. White box testing was conducted on all major methods for a given service in the system with JUnit version 4.12. Black box testing was conducted from the perspective of a user who would apply normal operations on the system.

White Box testing

Table 2: White box testing

Test class name	Target test class	Test class method	Target method	Assert outcome
LibraryServiceTest	LibraryService	createLibraryTest	createLibrary	PASS
LibraryServiceTest	LibraryService	deleteLibraryTest	deleteLibrary	PASS
LibraryServiceTest	LibraryService	getLibraryByIdTest	getLibraryById	PASS
LibraryServiceTest	LibraryService	getLibrariesTest	getLibraries	PASS
UserServiceTest	UserService	checkTokenTest	checkToken	PASS
UserServiceTest	UserService	getTokenTest	getUserToken	PASS
UserServiceTest	UserService	checkPermissionsTest	checkPermissions	PASS
UserServiceTest	UserService	getUserByCPRTTest	getUserByCPR	PASS
UserServiceTest	UserService	getUserByUIDTest	getUserByUID	PASS
InventoryTest	InventoryService	addGetRemoveBook	getBooks	PASS
InventoryTest	InventoryService	addGetRemoveBook	getBooksByTitle	PASS
InventoryTest	InventoryService	addGetRemoveBook	getBookByISBN	PASS
InventoryTest	InventoryService	addGetRemoveBook	getBooksByDate	PASS
InventoryTest	InventoryService	addGetRemoveBook	getBookByBID	PASS
InventoryTest	InventoryService	addGetRemoveRentals	getRentalsByUID	PASS
InventoryTest	InventoryService	addGetRemoveRentals	getRentalsByBID	PASS
InventoryTest	InventoryService	addGetRemoveReservation	getReservationsByUID	PASS
InventoryTest	InventoryService	addGetRemoveReservation	getReservationsByBID	PASS
InventoryTest	InventoryService	add/full/simpleAddBook	addBook	PASS
InventoryTest	InventoryService	addGetRemoveBook	removeBook	PASS
InventoryTest	InventoryService	addGetRemoveReservation	addReservation	PASS
InventoryTest	InventoryService	addGetRemoveRentals	addRental	PASS
InventoryTest	InventoryService	addGetRemoveReservation	removeReservation	PASS
InventoryTest	InventoryService	addGetRemoveRentals	removeRental	PASS
InventoryTest	InventoryService	addGetRemoveBook	getAllBooks	PASS

Black box testing

Legend:




Conclusion	Sign
Pass / Success	
Not applicable / None	
Fail	

Table 3: Core functionality testing







































Test ID	Target component	Test procedure	Comments / Remarks	Event type	Outcome	System reacts on error?
1	System client	Execute client main runnable	Server must be running first	Application launch		
2	System client	Run client from multiple computers	Server must be running first	Application operation		

Table 4: System content testing based on user perspective

Test ID	Target test section	Test procedure	Comments / Remarks	Event type	Outcome	System reacts to wrong input?
1	LoginController	Input CPR number	-	Data type Long input		
2	LoginController	Input password	Password is not visible	Data type String input		
3	LoginController	Select library from dropdown	Dropdown length depends on added libraries	Mouse press sequences		
4	LoginController	Press "Login" button	Login action performed on correct input	Button press		
5	LoginController	Press "Help" button	-	Button press		
6	HomeController	Input book search criteria	Redirects to RentBookController	Data input		
7	HomeController	Press "Search" button	Redirects on valid input of search criteria	Button press		
8	HomeController	Press "Rent" button	Redirects to RentBookController	Button press		
9	HomeController	Press "Return" button	Redirects to ReturnBookController	Button press		
10	HomeController	Press "Renew" button	Does not function	Button press		

11	HomeController	Press "Administrative features" button	Redirects to AdministrativeFeatureSetController	Button press		
12	HomeController	Press "Logout" button	Logs out user from session	Button press		
13	HomeController	Press "My Profile" button	Redirects to ProfileController	Button press		
14	LoanerLoginController	Input CPR number	-	Data type Long input		
15	LoanerLoginController	Input password	Password is not visible	Data type String input		
16	LoanerLoginController	Press "Login" button	Login action performed on correct input	Button press		
17	LoanerLoginController	Press "Cancel" button	Closes current controller	Button press		
18	RentBookController	Press "Back" button arrow	Closes current controller and goes back	Button press		
19	RentBookController	Input book ISBN	-	Data input		
20	RentBookController	Press "Rent" button	Displays outcome on press in text area below	Button press		
21	AdministrativeFeatureSetController	Press "Back" button arrow	Closes current controller and goes back	Button press		
22	AdministrativeFeatureSetController	Input book ID	Under "All Current Libraries"	Data type Integer input		
23	AdministrativeFeatureSetController	Press "Search libraries by ID" button	Updates table below on valid input	Button press		
24	AdministrativeFeatureSetController	Double click libraries table	Double click has to be on an item in table	Double click		
25	AdministrativeFeatureSetController	Press "Create library" button	-	Button press		
26	AdministrativeFeatureSetController	Input user CPR	Under "All Current Users"	Data type Integer input		
27	AdministrativeFeatureSetController	Press "Search users by CPR" button	Updates table below on valid input	Button press		
28	AdministrativeFeatureSetController	Double click users table	Double click has to be on an item in table	Double click		
29	AdministrativeFeatureSetController	Press "Create user" button	-	Button press		
30	AdministrativeFeatureSetController	Input book ISBN	Under "All Current Users"	Data type Integer input		
31	AdministrativeFeatureSetController	Press "Search books by ISBN" button	Updates table below on valid input	Button press		
32	AdministrativeFeatureSetController	Double click books table	Double click has to be on an item in table	Double click		
33	AdministrativeFeatureSetController	Press "Add book" button	-	Button press		
34	CreateLibraryController	Input library name	-	Data input		
35	CreateLibraryController	Input library location	-	Data input		

36	CreateLibraryController	Press "Create library" button	Creates library on valid input	Button press		
37	CreateLibraryController	Press "Cancel" button	Closes current controller	Button press		
38	EditLibraryDetailsController	Input library name	-	Data input		
39	EditLibraryDetailsController	Input library location	-	Data input		
40	EditLibraryDetailsController	Press "Edit library" button	Edits a library on valid input	Button press		
41	EditLibraryDetailsController	Press "Cancel" button	-	Button press		
42	EditLibraryDetailsController	Press "Delete library" button	Deletes a library	Button press		
43	CreateUserController	Input user CPR	-	Data input		
44	CreateUserController	Input user name	-	Data input		
45	CreateUserController	Input user password	Password is visible	Data input		
43	CreateUserController	Input user email	-	Data input		
44	CreateUserController	Select user role	Dropdown with roles	User selection		
45	CreateUserController	Press "Create user" button	Creates a new user on valid input	Button press		
46	CreateUserController	Press "Cancel" button	Closes current controller	Button press		
47	EditUserController	Input user name	-	Data input		
48	EditUserController	Input user password	Password is visible	Data input		
49	EditUserController	Input user email	-	Data input		
50	EditUserController	Select user role	Dropdown with roles	User selection		
51	EditUserController	Press "Edit user" button	Edits a user on valid input	Button press		
52	EditUserController	Press "Cancel" button	Closes current controller	Button press		
53	AddBookController	Input book name	-	Data input		
54	AddBookController	Input book author	-	Data input		
55	AddBookController	Input ISBN	-	Data type Long input		
56	AddBookController	Select calendar date	-	User selection		
57	AddBookController	Input description	Optional	Data input		
58	AddBookController	Press "Add book" button	Action performed on valid input	Button press		
59	AddBookController	Press "Cancel" button	Closes current controller	Button press		

60	EditBookDetailsController	Press "Back" button arrow	Closes current controller and goes back	Button press		
61	ProfileController	Press "Back" button arrow	Closes current controller and goes back	Button press		
62	ProfileController	Input email	-	Data input		
63	ProfileController	Press "Update" button	Updates email on valid input	Button press		
63	ProfileController	Input current password	Password is not visible	Data input		
64	ProfileController	Input new password	Password is not visible	Data input		
65	ProfileController	Input repeat password	Password is not visible	Data input		
66	ProfileController	Press "Change password" button	Updates user password	Button press		

Test conclusions

A total of sixty six tests have been done. Tests with a higher focus have been marked in orange color in table two. The outcome shows that two of the tests do not pass user expectations of the said feature being tested.

7. Results and Discussion

Originally, the system was supposed to have the ability to acquire information about new books from the OpenLibrary API, easing the work of the librarians. The system proved to be more complex than first anticipated and thus it was decided to move the feature to Project Future.

According to internal assessment in the group, the user interface satisfies the goal to develop a user-friendly interface, which is broadly usable by many types of users with different levels of technical background and experience.

The RMI based services function as intended, however they proved ultimately to be inadequately designed in regard to what the client software needed to function. The services do not support any form of searching, even though the client software is supposed to support such features.

Further, the server does not allow the client to update books, due to the fact that the services do not support actually removing a book universally and also does not have an update method for books.

The database makes use of several reference tables, that map ids between tables. As an example, the Books table is a universal table that all libraries use to extract information about books. The BookInventory table is used to map books from the Books table to the Libraries table and also contains the amount of each book available in each specific library. This proved to be a good design. Increasing the inventory, does not increase the number of entries in the database. Relations between data entities is separated from the information.

RMI is a reliable and easy way of creating communication between server and client, however in hindsight it inhibits the third party client development, as it requires at least a java wrapper to run on the client side. A less restrictive implementation would have been to utilize sockets for a custom protocol and a system independent standardized data format like JSON or XML.

8. Conclusion

In conclusion, the overall performance of the system is excellent. The UI works and the RMI connection works as intended. The client software does manage to perform the most important tasks, using the functionality offered by the services, despite of the services lacking in certain aspects.

The purpose of the project has been met, as the system does use an open centralized data model, the system is long term maintainable and the client can use third party developed software to interact with the server.

The data model does not use any proprietary formats or similar, that would legally limit the development of a third party client.

The system is written in a major programming language and uses a connection technology, that will likely continue to be supported in the long term.

It is possible to implement a third party client, as long as it is able to handle java RMI and objects.

9. Project future

There are several improvements that are likely to be made in the future:

1. It should be possible for loaners to renew book rentals before the due date of their rental.
2. Implementation of a search function, that allows the search buttons in the client GUI to perform actual meaningful searches, in which not just one result is returned.
3. The system should be able to get information about books from other open resources, as was originally intended, possibly from several different resources.
4. It would be necessary to add hardware capabilities, so the system can make use of barcode scanners, RFID scanners or similar and not rely on manual input.

10. References

Version2, 2016. Bibliotekarar: »Det, der tog 20 minutter i vores gamle system, bruger vi nu en hel formiddag på«. [online] Available at:

<<https://www.version2.dk/artikel/bibliotekarar-det-der-tog-20-minutter-i-vores-gamle-system-bruger-vi-nu-en-hel-formiddag-paa>> [Accessed 1 March 2018].

Ojedokun, A, Olla, G, & Adigun, S 2016. *'Integrated Library System Implementation: The Bowen University Library Experience with Koha Software'*. [pdf] Available

at:<<https://drive.google.com/file/d/1sOG6vsjk7owMS7kM5gWh8qpRXAO1-k4L/view?usp=sharing>> [Accessed 7 March 2018].

Librarytechnology.org. 2015. *Ex Libris company profile*. [online] Available at:

<<https://librarytechnology.org/vendors/exlibris/>> [Accessed 8 Mar. 2018].

Statisticstimes.com. 2017. *Top Computer Languages 2017 - StatisticsTimes.com*. [online] Available

at: <<http://statisticstimes.com/tech/top-computer-languages.php>> [Accessed 8 Mar. 2018]

11. List of appendices

A: System code (CLMS.7z)

B: System class diagram (Group_5_CLMS_Class_Diagram.svg)

C: System activity diagrams (Group_5_CLMS_Activity Diagrams.7z)

D: Server user guide (Group_5_CLMS_Server_User_Guide.pdf)

E: Client user guide (Group_5_CLMS_User_Guide.pdf)