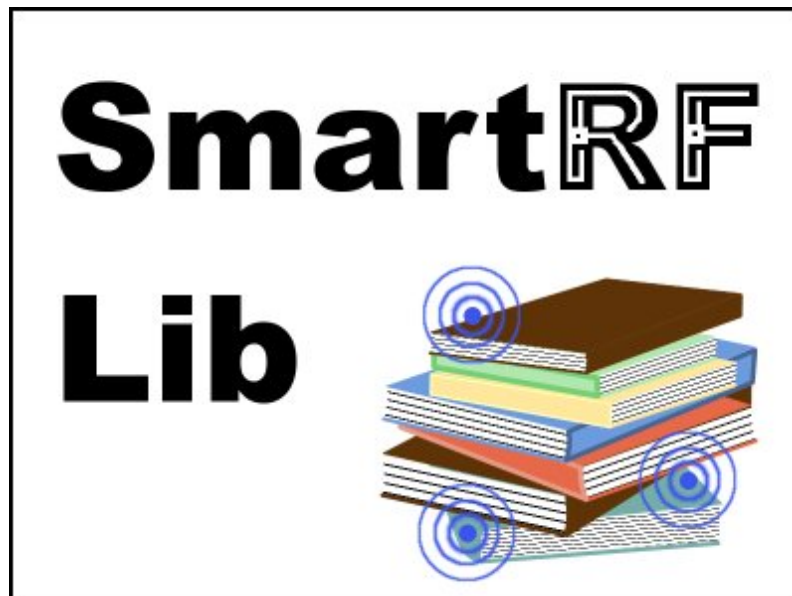


## Progress Report I



---

## SmartRF Lib

---

Cagri Balkesen  
Ali Sengül  
Nihal Dindar  
Florian Keusch  
Catharina Kromwijk  
Gautier Boder

November 21, 2007

# Contents

|     |  |    |
|-----|--|----|
| 1   | Visualization Layer . . . . .                              | 3  |
| 1.1 | Introduction . . . . .                                     | 3  |
| 1.2 | Second life virtual world . . . . .                        | 3  |
| 1.3 | Why we need a web interface . . . . .                      | 5  |
| 1.4 | Interaction between SmartRfid and second life . . . . .    | 5  |
| 1.5 | The web interface . . . . .                                | 9  |
| 1.6 | Technical background . . . . .                             | 9  |
| 1.7 | Timetable for VL (GUI) . . . . .                           | 11 |
| 2   | Event Processing Layer: . . . . .                          | 12 |
| 2.1 | RFID data simulator (package: trunk.src.qp.sim): . . . . . | 12 |
| 2.2 | Query Language: SRFLL (SmartRF Lib Language) . . . . .     | 13 |
| 2.3 | Event Processing . . . . .                                 | 15 |
| 2.4 | Database and predefined functions: . . . . .               | 17 |
| 3   | Data Acquisition layer . . . . .                           | 17 |
| 3.1 | Interface to upper layer: . . . . .                        | 17 |
| 3.2 | Reader, Tag modes: . . . . .                               | 18 |

# 1 Visualization Layer

## 1.1 Introduction

The visualization of this project consists of two parts: a virtual library in second life and a web-based application. The two interfaces are complementary and enable us to have an original way of viewing the current state of the library. We can, for instance, get lists of books which are currently borrowed through the web interface, while the position of a book can be seen in second life.

## 1.2 Second life virtual world

### Introduction

We are building a virtual library in second life. This is quite an unusual way of presenting information which has some advantages and disadvantages. The advantage is a quite realistic and intuitive display of the information. The disadvantage is that it is maybe harder to get an overview. This library will, like in real life, have shelves and books. The books of this library will, at first, appear and disappear in the shelves. Later, we will try to make them „fly“ from one place to another. The movements of the books do not correspond 1:1 to the real life books, since we don't have antennas everywhere, which would allow us to do real-time positioning. We are going to have a place where we can see the checked out books and the books which are currently not localizable are semi-transparent.

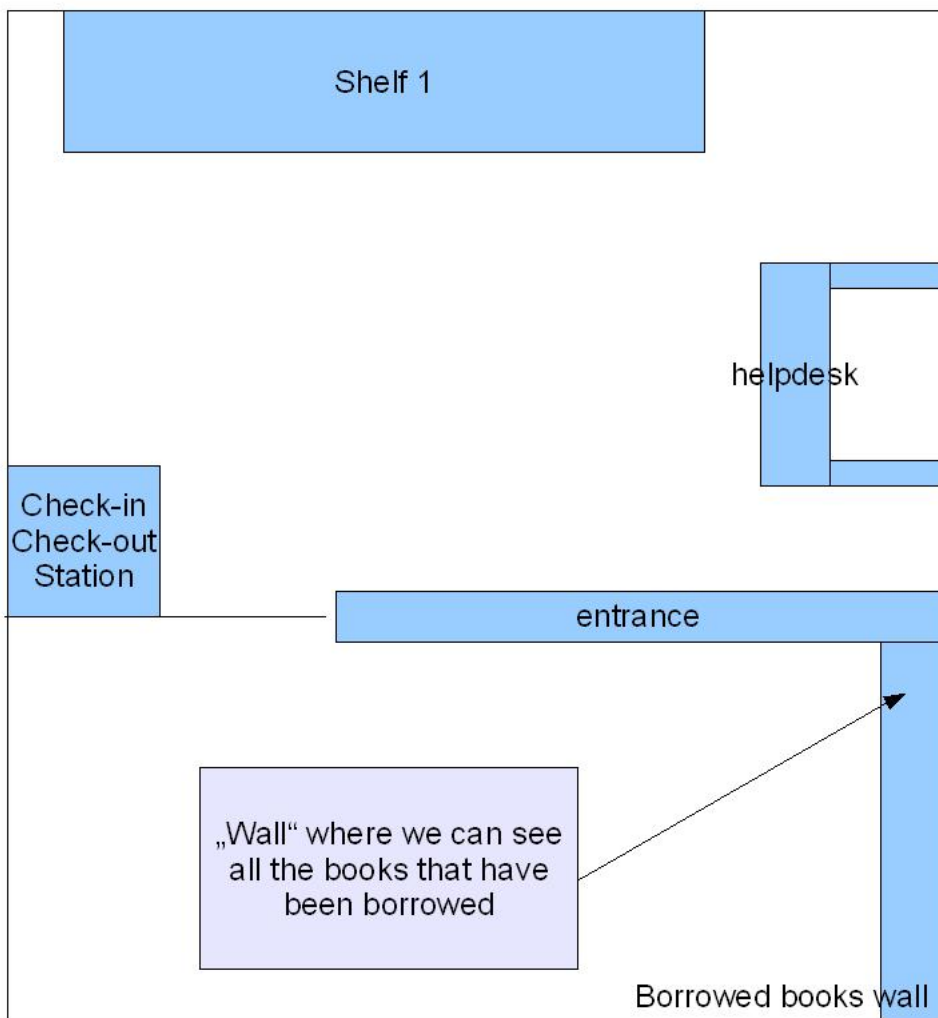


Figure 1: library architecture

### 1.3 Why we need a web interface

The 3D world gives us a nice and intuitive way of accessing information, however there are some pieces of information which need to be presented in another way. For instance if you want to search a book or see a list of books currently checked out, this cannot (for the moment) be done in second life. With further development in second life, the web interface might become obsolete or integrated totally in second life (watch a website on a computer screen). The web interface will let us do different query and define new query's in an easy way. t

### 1.4 Interaction between SmartRfid and second life

As mentioned in the introduction section our core application is based on two different layers and is driven by events coming from the lowest layer. These events are processed by a dedicated part and then available for the visualization layer that has to show them to the end user. Below is the description of the interaction method we choose between the main core of the application and the visualization layer located on the ETHZ's Second Life<sup>1</sup> island. We had to look for a method allowing external programs to send events to any object located anywhere in the SL world. After we having tried several approaches of transmitting data and events to this virtual world we found that Linden Labs provides a XML-RPC server that allows to communicate with any object from their world. This is the solution we chose.

However this communication method requires to split every XML-RPC system request. The first can be programmed in every language on the world that allows to send HTTP requests nevertheless the second needs to be programmed in LSL<sup>2</sup> as it is the receiver of the XML-RPC call for the target object. the XML-RPC call is sent from our client to defined server owned by LL<sup>3</sup> but this server needs to know to which object among the whole SL he has to transmit this call. This communication is done over a channel previously opened by the target object and using this channel's identifier specified into the XML-RPC call parameters. Using this the server forwards this XML-RPC call to the correct object. Remains now the last part of the structure, a piece of LSL script that handles this call. This script has to provide the method `llRemoteCall` with the strictly defined signature and processes the received data. Basically, the communication process described above can be summarized using the figure below.

---

<sup>1</sup>Second Life, furthermore under Google and abbreviated as SL starting this point.

<sup>2</sup>LSL := Linden Script Language

<sup>3</sup>LL := Linden Labs

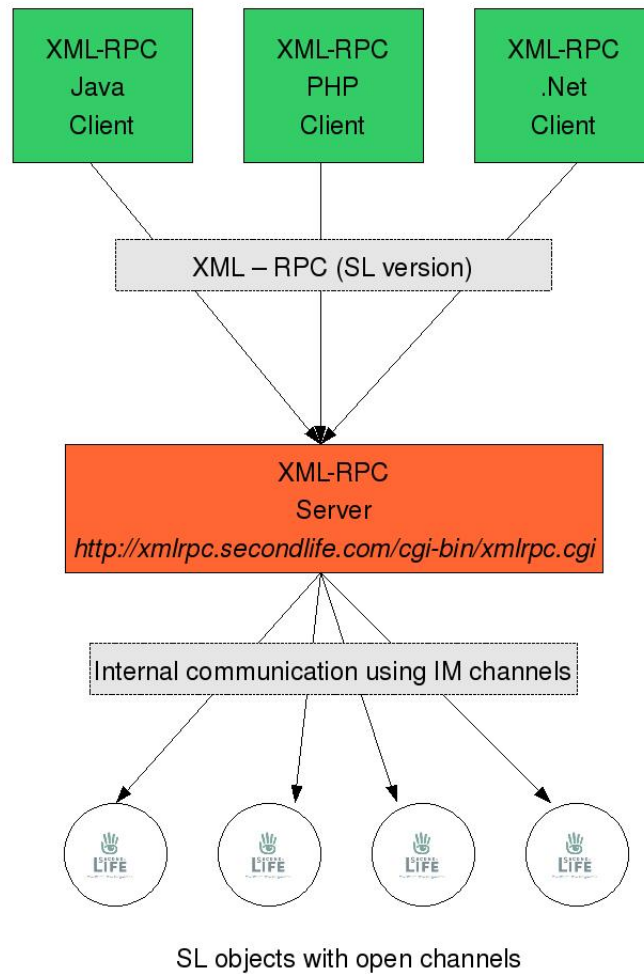


Figure 2: SL XML-RPC System

Nevertheless, this communication system has some restrictions. These restrictions are set by LL and not by the usage of the XML-RPC protocol. For example, it doesn't seem that you can send the number of parameters you like. The number is restricted to one string value and one integer value. However, this will be tested in the following days. Another important consideration is that the channel's identifier is random and that every time an object opens a new channel listening to the server calls, this identifier will be generated from scratch. For small tests and usage this isn't a big issue. However, for our system it is a major problem.

In fact before talking about this problem we need to describe in details how will our communication system work within the SmartRfid project. The QP <sup>4</sup>receives events from the above layer and processes it. As soon as it detects an event that requires an update for the visualization layer it will raise a new event destined at the VL <sup>5</sup>. The channel used for this communication is not yet defined but it will be either RMI, Webservice, XML-RPC or basic sockets. The receiver of this event will process the

<sup>4</sup>QP := Query Processing layer

<sup>5</sup>VL := Visualization Layer

message and transfer the corresponding event to the SL world using the XML-RPC mechanism. Since we already need to implement a web site for some functionalities (see dedicated section) the RMI approach is not the best solution. The exchange between QP and VL is summarized into the figure below.

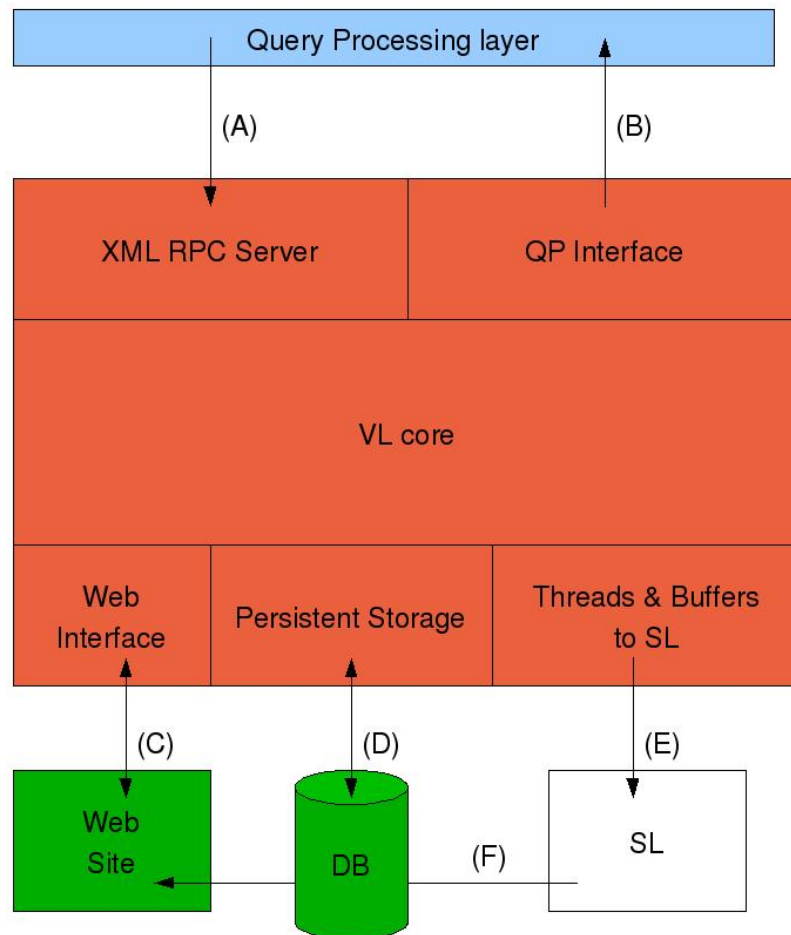


Figure 3: VL System schema

This communication will exchange only the required information to describe the current event. These events and their related information are described into the following subsections. For further information the VL query the QP using [using what?].

#### Communication links of the VL's core

- (A) The QP team will receive an interface that they can use to send XML-RPC events to the VL's core.

- (B) The QP team will provide an interface / protocol that we can use to query their database and information.
- (C) The communication between the VL and the web application is not clear yet. We need to finalize the thinking of the global system and maybe the Website and the VL core will fusion and the Web site will only be an internal of the VL's core.
- (D) The VL core has to store some information for its processing about objects, their location and virtual library positional information. To retrieval / storing of these data will be done by a well known JDBC system.
- (E) After being processed by the Core the received events are transmitted to the SL world using the defined XML-RPC protocol.
- (F) Every object into SL will need to register on the VL's core to able to received events through XML-RPC. The registration will save its channel in the available channels. This registration process is not planned being automatic in a first time instead of we will provide a page on the web application to manage the objects' channels identifiers.

**Everything about the events sent by the QP is a presumption. We need to discuss as soon as possible of these events with the QP team.**

### **Events from QP**

This section describes all events that are exchanged between the application QP layer and the Visualization layer. After their processing by the VL these events are retransmitted to the SL world to be shown to the end user.

Now we detail each event and the data that are associated to these events. Mainly the data exchanged is information about the new location of an object and its identifier. Maybe the object's type identifier and status could also be useful in some case like when the move doesn't describe a book but a person or when a book is defined as stolen or missing.

### **Processing of events inside the VL (Core)**

Not all events received from the QP describe move or location changes. Some of these events describe changes of book's state and these have to be converted to move events for SL. Although events that describe location changes need to be processed, since the XML-RPC system ask for a Channel identifier to point an object inside the SL world (See beginning of section 1.4) and the QP layer does not know this identifier and it uses the rfid patch's identifier as primary identifier. We need to convert this identifier to the corresponding object's channel identifier.

To be able to convert this rfid to the channel's id the VL needs to know the channel identifiers of every object in SL world and to associate an rfid to them. This requires three specifications for the VL layer; A listener on which the object can register their channel identifier, a administrative web page that allows users to associate channel identifier with a rfid identifier and the latest a persistent storage that store all these relations.

The persistent storage stores information about current location in space of every objects into the SL world. In fact, not only for objects that could move but also for the furnitures. This is due to the fact that moving objects using LSL is done on the client side and not on the server side of the SL system and there is no collision detection on this side. We need to calculate this collision detection within the VL before sending

move events to SL. Therefore, we need to know the location of every objects to be able to compute a valid new location of any of them. For example, if we want to move a book B to shelf A, we need to place the book where there is room left on the shelf and not over another book or inside the shelf's cupboard.

Once all these conditions are satisfied and all the processing is done the VL core will create a new set of XML-RPC events that are sent to the SL world to affect the corresponding object. Once the SL world has processed the event we set the object's new location inside the persistent storage. As this process of sending XML-RPC orders is pretty complex it has its own section after this one.

### **Sending events to SL using XML-RPC**

This section described how events are sent to the SL world using the LL given XML-RPC framework. As we said before (See section 1.4) the XML-RPC implementation of LL is quite different of the widely used standards. The xml document sent cannot have more than one defined set of parameters: a string and an integer value. It is also required that every parameters are named and the first parameter as to be the channel identifier of the target object.

The other weakness of the LL implementation is the availability and reliability of the whole platform. The elapsed time between the send and the answer from on XML-RPC can be greater than 45 seconds. Furthermore, as forums list the whole system has a error rate greater than 60%. These facts require that we handle the sending process correctly inside the VL to be sure we know whether an event has been sent and processed by the target object or not! This is supposed to be assured by a multi-threaded buffer structure in which every object would have its own buffer with a queue of all events that have to be executed. This queue will be read by an object dedicated thread that tries to send the event. If the transmission fails, the thread will try to send the event again until a maximal number of tries is reached. Then it will report that this event has not been processed and mark the given object as unsynchronized and will not process any more events on this object.

## **1.5 The web interface**

The web interface is used to allow user inputs and queries. It allows users to request for a certain books and displays their location graphically (using bitmaps). If a searched books should be on shelve A, but it is located on the shelve B it will be marked as mistakenly placed. Books that are taken out of the library without authorization or that aren't authorized to leave the library need also to be listed.

The functionalities that the web interface will offers are the following

- Query for book and indicate their current status and location (We could provide a sl-link to the corresponding location of our virtual library)
- Display real-time information on book movements and status change
- Offer an unique url that will be called by the SL Objects to register their XML-RPC channel into the VL layer.

## **1.6 Technical background**

This chapter covers all technical knowledge we had or we needed to acquire to realize this project. These learnings were various and for some of them tricky to learn. Nevertheless, Google is often your friend and after some queries you get the document that you were expecting. We can split all the topics in different categories. Grouped by domain it became clearer how some of them are inter-dependent.

## Second Life

**The system** Second Life is based on client-server architecture. Each of them has its defined tasks and they communicate using a dedicated protocol. The information exchanged is as minimal as possible to ensure a strong efficiency and a low network usage. This chapter doesn't pretend to teach you the whole structure of Second Life but only the main concepts that will help you to understand the project.

Another point differentiates strongly these parts. Server side must be seen as a black-box without having any idea about how it is works or programmed. On the other hand, the client side (viewer) code is open source and ready to be edited by anybody that wants to add new functionalities. However, Linden labs edited some rules asking to edit the viewer as less as possible to keep one version around the world and deeply recommend using LSL to program SL environment.

Now study a little more deeply the specific tasks of each part. This enumeration is extracted from this page on the Second Lifes' Wiki [http://wiki.secondlife.com/wiki/Server architecture](http://wiki.secondlife.com/wiki/Server_architecture) (Simulator vs. Viewer).

- Simulator's job (server):
  - Runs physics engine
  - Collision detection
  - keeps track of where everything is
  - Sends locations of stuff to viewer
  - Sends updates to viewers only when needed (only when collision occurs or other changes in direction, velocity etc.)
- Viewer's job (client):
  - Handle locations of objects
  - Gets velocities and other physics information, and does simple physics to keep track of what is moving where

Web interface. For more information about this system and their components please refer to these websites from Linden Lab. At first, the information seems to be unstructured (though its a little the case) but after a while, you will discover how much information is hidden in these wiki pages.

**Outside interaction** The interaction of external applications with the world of SL has to be done using the XML-RPC implementation provided by LL. Since this seems to be a new functionality it isn't well documented and used yet. This communication system requires knowledge about the following domains.

- XML (Creating, manipulating and reading documents)
- How to send HTTP POST request with your programing language (in our case Java)
- The constraints applied to the XML-RPC "protocol" by LL

You can find information about each of these point either on Google or in our references. We are still trying to include in an XML schema for the Second Life XML-RPC implementation in our project. However, we don't know yet if we will have the time to create this or not and we need to communicate with SL to be sure that all the possibility are documented on their wikipedia page.

## Web interface

The background knowledge cant be defined yet because the web interface is not designed and we don't know which technology will be used. It will be PHP / JS or J2EE or anything to do with web application.

## Divers

**Communication** Communication is the key word of our whole project, we have several systems that might be located on several computers and we need to get and send information between all these entities. For the VL, a different protocol is used and listed below. Information on this protocol can be found again under Google (its definitely the Oracle) and maybe in our references.

- Interaction between the VL's core and the SL world -> XLM-RPC
- Interaction between the VL's core and the QP layer -> ! TBD !
- Interaction between the VL's core and the Web application -> ! TBD !
- Interaction between the QP and the VL's core -> XML-RPC (We will provide a whole interface for the QP layer that allows sending of events)
- Interaction between the VL's core and the Database -> Probably JDBC and DAO pattern.

**Visualization's core** The VL's core will be programmed in Java (quite sure version 1.6) however it hasn't been engineered yet and we don't have much more ideas on how it would like in term of patterns or different technologies.

## 1.7 Timetable for VL (GUI)

### Previous objectives

| Date       | Description                                | Status      |
|------------|--|-------------|
| 24.10.2007 | Starts a LSL script on a outside events    | completed!  |
| 24.10.2007 | Create a script that move objects around   | completed!  |
| 24.10.2007 | Define the web application functionalities | in progress |
| 31.10.2007 | Design the whole visualization layer       | in progress |
| 31.10.2007 | Find a place where to build                | completed!  |
| 31.10.2007 | Build our virtual library                  | completed!  |
| 07.11.2007 | Write the first progress report            | in progress |

## Futures objectives

| Date       | Description                                       | Status      |
|------------|---|-------------|
| 14.11.2007 | Implements the SL communication component for VL  | in progress |
| 14.11.2007 | Finalize the functionalities of the web interface | -           |
| 14.11.2007 | Finalize the whole design of the VL               | -           |
| 21.11.2007 | Finalize the communication with the other layers  | -           |
| 21.11.2007 | Begin implementation and testing                  | -           |
| 28.11.2007 | Write the second progress report                  | -           |
| 05.12.2007 | Final testing with the other layers               | -           |
| 12.12.2007 | First presentation to restricted public           | -           |
| 17.12.2007 | Second working demo                               | -           |

## 2 Event Processing Layer:

In the last few weeks, we have spent a lot of time in modeling our event processing layer. We had many meetings discussing the important issues of our layer. Event processing from streams is still an open research issue, we've seen that there exist many different possible solutions. So far we have made progress in the following parts of our event processing layer:

### 2.1 RFID data simulator (package: `trunk.src.qp.sim`):

We wrote a simple RFID data simulator that can generate us RFID data streams as they will later occur in the real life system of our library. The simulator takes an input file where the following data is given:

- the reader tags (R1, R2, R3)
- the RFID book tags
- the RFID person tags

From this data the simulator can randomly generate events in a specified averaged time interval. The simulator is implemented as a publish/subscribe system. An observer can subscribe the simulator that publishes the random events. Right now the simulator publishes primitive events. But these can later easily be mapped into the desired stream that is needed for the input for the event detection engine.

open issues:

- ▷ generate stream data

## 2.2 Query Language: SRFL (SmartRF Lib Language)

We defined the language SRFL in which we define our events to be put in the system. It was not so easy to figure out all the functionality we need to provide for our library system. We decided not to give the possibility to write own SQL queries in SRFL. So we came up introducing predefined functions for writing such events. The library administrator can then somehow write a new event in our defined language and add it to the system to be processed in further time. Many ideas from our EBNF for this language are taken from the SASE papers. There they define the SASE language to query the sequential data for certain events happening. We adapted this SASE language to be applicable for our SmartRF Lib library application. We introduced having a name for each defined event. The SASE language lacks having an action part, where some functions update the current state of the database (for example when we do a checkout then we have to store that the books  $b[]$  are borrowed by person  $p$  in our database). As far as we know we won't use the SASE tags `HAVING` and `WITHIN` for our library implementation. Our definitions seem to be working more or less for now. When we proceed with the implementation we might detect some lacks and update them.

open issues:

- ▷ define the ACTION part → update event definitions
- ▷ adapt the sequence

## SRFLL - EBNF

```
eventDecl ::= NAME string PATTERN onDecl patternDecl ACTION
           actionDecl RETURN returnDecl
           | NAME string PATTERN onDecl patternDecl WHERE whereDecl ACTION
           actionDecl RETURN returnDecl
           | NAME string PATTERN onDecl patternDecl WITHIN withinDecl ACTION
           actionDecl RETURN returnDecl
           | NAME string PATTERN onDecl patternDecl WHERE whereDecl
           WITHIN withinDecl ACTION actionDecl RETURN returnDecl

onDecl    ::= | ON IDENTIFIER:i
returnDecl ::= boolExpr
actionDecl ::= functionCall | functionCall , actionDecl
patternDecl ::= SEQ [ qItemList ]
qItemList  ::= qItem | qItem , qItemList

qItem     ::= string string | string+ string []
           | string* string []
whereDecl ::= expr | expr AND whereDecl
withinDecl ::= 00..24 00..60 00..60
expr      ::= numExpr | setExpr
           | boolExpr | NOT boolExpr
numExpr   ::= numTerm = numTerm | numTerm <> numTerm
           | numTerm < numTerm | numTerm <= numTerm
           | numTerm > numTerm | numTerm >= numTerm
setExpr   ::= setItem IN setTerm | setItem NOT IN setTerm
           | numTerm = ANY setTerm | numTerm = SOME setTerm
           | numTerm = ALL setTerm | numTerm <> ANY setTerm
           | numTerm <> SOME setTerm | numTerm <> ALL setTerm
           | numTerm < ANY setTerm | numTerm < SOME setTerm
           | numTerm < ALL setTerm | numTerm <= ANY setTerm
           | numTerm <= SOME setTerm | numTerm <= ALL setTerm
           | numTerm > ANY setTerm | numTerm > SOME setTerm
           | numTerm > ALL setTerm | numTerm >= ANY setTerm
           | numTerm >= SOME setTerm | numTerm >= ALL setTerm

setItem   ::= functionCall | number | bool | string
setTerm   ::= functionCall
boolExpr  ::= boolExpr OR boolExpr | boolExpr AND boolExpr
           | functionCall
numTerm   ::= numTerm + numTerm | numTerm - numTerm
           | numTerm * numTerm | numTerm / numTerm
           | functionCall | number
functionCall ::= string.string() | string.string (arglist)
           | string[number].string() | string[number].string(arglist)
           | string (arglist)
arglist   ::= string | string , arglist
number    ::= integer | real
integer   ::= digit | digit,digit
real      ::= integer . digit
digit     ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
string    ::= letter | letter letter | letter digit
letter    ::= A .. Z | a .. z
```

## SRFL - Events

### Not borrowable

User wants to take book which has to be used in library only

```
NAME not_borrowable ON r1
PATTERN SEQ(Book+ b[ ])
WHERE NOT borrowable(b[i])
ACTION alarm(), b[.id]
RETURN True
```

### Too much books

User wants to take books more than defined book limit

```
NAME too_much_book ON r1
PATTERN SEQ(Person p, Book+ b[ ])
WHERE getBookLimit(p.id) < countBorrowedBooks(p) + b[.size]
ACTION alarm(), p.id, getBookLimit(p.id)
RETURN true
```

**Book theft** A book that is not yet borrowed leaves the library

```
NAME book_theft ON r3
PATTERN SEQ(Book+ b[ ])
WHERE NOT borrowed(b[i].id)
ACTION alarm(), b.id
RETURN true
```

**Checkout of book** A normal event when a user borrows a book

```
NAME book_checkout ON r1
PATTERN SEQ(Person p, Book+ b[ ])
WHERE NOT not_borrowable(b) AND NOT too_much_book(p,b)
ACTION checkout(p,b)
RETURN true
```

**Return of book** A user returns his borrowed books

```
NAME book_checkin ON r1
PATTERN SEQ(Person p, Book+ b[ ])
WHERE isBorrowedBy(b,p)
ACTION checkin(p,b)
RETURN true
```

## 2.3 Event Processing

### Event logic generation

We use the tools jlex and java\_cup (cup) to build the parser that builds the event processing logic from an event description file. The event processing logic will be built

the following way: An event is defined in a simple text file in the representation of SRFL. This event file is then passed to the lexer which generates symbols in a first part. The output of the lexer (Yylex) is handled by the parser. The parser then produces a new object that can detect this particular event in a stream and can be added to our event detection logic. We do so for all our text defined events in order to get the full event detection logic. For the lexer we had to define a lex file to detect the keywords and convert them to symbols. For the parser using the library java\_cup we can provide a .cup file where we defined the EBNF according to SRFL. We managed to solve all the conflicts and have now a parser just doing nothing at the moment. In further steps we have here to generate recursively the object that will later evaluate streams. We have to set all the conditions needed to evaluate correctly. This also means to build kind of an evaluation tree for every state we can detect in a such automaton. We're right now on the task of defining such a data structure that can be built in this parser and is able to detect events in our streams.

open issues:

- ▷ fill in parser code to generate automaton (RESULT)
- ▷ whole logic generator class
- ▷ throw parse Exceptions

#### **Finite state automaton to evaluate stream data:**

We decided to use finite state automaton to handle the detection of our SRFL. Each event is basically a such state machine. The internal states changes according to the given transitions of the events. We can reach accepting states. Then the event is triggered and we do the specified action to keep our database up to date or give messages (predefined functions). We can also reach rejecting states. Then we reset the state machine. A state machine can have return results. If we push some stream data into a state machine then we can check if the event is triggered (and the actual state) by calling a function. To evaluate conditions that use more than 1 different attributes we have to use a data structure like a match buffer. There we can store all the streams we had before from the same window to process further conditions in the state machine. A finite state automaton basically takes a stream and can also give back a stream of data. This would enable the definition of nested events (complex events). When designing such finite state automaton several performance issues arise. We clearly see that some conditions are evaluated more than once. We just stared thinking about the implementation of these finite state automaton. There may arise questions and unsolved problems soon.

open issues:

- ▷ implement this idea
- ▷ connect states to meaningful events

## 2.4 Database and predefined functions:

We adapted the basic database layout. Some redundancy was eliminated. Now we use the RFID tags as a primary key where we had an extra id as a primary key. Since the RFID tags have to be unique we can take them as a primary key. This also saves us unnecessary joins when looking up data in the database.

We introduced many predefined functions as connections to the database. These functions will be used as a connections between the database and the business logic. These functions can be used by the library administrator to place them in event definitions. What these predefined functions basically do is just run a specific SQL query to manipulate our MySQL database. Many of these functions have not been tested so far. We're using junit tests to test the correctness of these functions, but that's still work in progress.

open issues:

- ▷ test predefined functions

## 3 Data Acquisition layer

### 3.1 Interface to upper layer:

After our data is cleaned and compressed we will provide an interface to the upper layer to access this data. In order to provide an interface to the upper layer to access the data we will generate primitive event tuples to the upper layer which are more reliable than the raw readings. We will create object streams over TCP/IP sockets for upper layer to access the data. We will provide Time, ObjectID and ReaderID ordered by time and grouped by ReaderID object streams to the upper layer. Cleaning of dirty data in the data acquisition layer will be done with using both static and adaptive methods. Data cleaning will be done static for the places where no movement and less accurate data is needed such as shelves. Window size will be defined with Database, Query Processing Layer. Implementation of the static window is planned to implement as putting read tags into a hash table with the read time. And reader has a variable named persist time. The persist time defines the duration between the time a tag was last read and the time it is automatically purged from the tag list. Static window size is planned to change by changing persist time variable of the reader. Security gates at the exit of the library are one of the most important components of the library. When a person taking out library items without checking-out, it will signal an alarm. The data at the security gate should be very accurate therefore, adaptive data cleaning is planned to implement for the data in the security gate readers. There are some challenges and problems for implementing adaptive cleaning. We still could not figure out the epoch which is specified as a number of interrogation cycles or as a unit time. Adaptive method uses per-epoch sampling property which is the probability of each tag to be detected in each epoch. This value still could not be figured out.

## 3.2 Reader, Tag modes:

The Alien ALR-9800 RFID reader is designed to read and program any EPC Class 1 Generation 1 or 2 tag and issue event reports to a host computer system. The host computer can be locally connected to the reader via RS-232, or at a remote network location. For our project we use remote network connection option which will make easier for us to send the cleaned data tuples to the upper layer. The reader's TCP/IP interface mimics a basic telnet interface, so you can use any telnet client to connect to the reader. Console can be used to interact with the reader using telnet and ip address of the reader.

The Alien ALR-9800 RFID reader has four digital inputs and eight digital outputs. This input pins can be used connecting a sensor which can be used to detect the distance to reader. The output pins can be used to send alarm signal when a person taking out library items without checking-out.

The Alien RFID reader provides two methods with which to read tags: Interactive Mode and Autonomous Mode.

Interactive Mode - the controlling application issues commands to the reader to read tags. This command will always immediately return with a list of tags in the reader's field of view. Autonomous Mode - the reader constantly reads tags, and may initiate a conversation with a network listener when certain events arise. We are using autonomous mode because it that enables automated monitoring and handling of tag data. The main advantage of autonomous mode is when a reader has been configured for Autonomous Mode, interactive communication with the reader is unnecessary and it can be left to work on its own. If it is known that a reader exists on a specified network address, a new reader object can be created directly. One way to do this is to instantiate the reader object using the constructor with the ip address as an input. Such as:

```
AlienClass1Reader reader = new AlienClass1Reader("129.132.242.177",23);
```

Once a reader object is instantiated, reader connection need to be opened in order to send commands and ask for tag data. To open a reader connection, following method is used:

```
reader.open();
```

Tags play a very important part in the RFID reader and tag system. There are two distinct methods for reading tags and the choice of one method over another depends on the application at hand.

**Global Scroll** Global Scroll is the most primitive of tag ID reading operations supported by the Alien RFID Reader system. When a global scroll command is issued, the RFID Reader sends a single command over the air to all and any tags. That command is simply a request for any tag to immediately send back its ID to the RFID Reader. However, because the command is so simple, problems may arise if there is more than one tag in the field.

**Inventory** The inventory command is a full-featured system for discerning the IDs of multiple tags in the field at the same time. This single high-level command transforms itself into a complex series of reader-tag interrogations that eventually resolve themselves into a single list of tag IDs seen by the RFID Reader. This method of interrogation and evaluation of multiple tags is known as an anti-collision search. For our project in order to avoid anti-collisions we will use the inventory methods.

The reader classes allow tags to be read by the reader and return the results either as a single Tag object or an array of Tag objects:

```
Tag[] tagList = reader.getTagList();
```

When a tag is in the range of two antennas, this tag can be read both of them. It is possible to combine or separate readings from each antenna using TagListAntennaCombine command. The TagListAntennaCombine command turns on or off the antenna combine mode. When TagListAntennaCombine is ON, the reader combines tag IDs into a single TagList even if the IDs are read by different antennas. Setting this value to OFF forces the TagList to keep multiple copies of a tag ID for each antenna where it is seen.

The notify classes work in conjunction with a reader running in autonomous mode. In autonomous mode the reader is configured to read tags over and over again without the need for human interaction. The reader can be configured to send messages to listening services on the network when specific events occur, such as a timer expiring, tags added/removed from the tag list, successful/unsuccessful programming, etc. The key class in the notify package is MessageListenerService. This is a service that listens at a specified port for incoming reader notification messages. The listener service as set up above constantly listens for these messages on its own thread until told to stop. When a message is received, it is parsed and converted into a Message object, which is passed to the messageReceived() method of the registered MessageListener. In order for an object to receive notification events from a MessageListenerService, it must implement the MessageListener interface. Only one method to implement and it simply receives a Message object from the MessageListenerService. A Message object encapsulates a collection of meta data about the notification message itself, and an array of Tag objects extracted from the taglist portion of the notification message.