Algorithmic and Programmation, Project
Autumn 2017
LO27

Centralized and secured transaction registry

Alexandre Lombard and Nicolas Gaud

1 Implementation and evaluation of the project

- Group of 1 or 2 persons at most.
- Implementation in C language.
- The project will be documented in a written report that outlines (15 to 20 pages) used data structures, the selected algorithms, optimizations and encountered difficulties. No copy-paste of entire pages of source code. Your report must at least contain:
  1. An introduction which recalls the subject, introduces the structure of the report and describes the major points of your work.
  2. A table of content
  3. The description of each new abstract data types used to implement your library
  4. The algorithmic description of each function described below. Each algorithm described in the report have to be documented using the template that have been introduced during the lesson.
  5. A conclusion providing an objective evaluation of your personal work, it summarizes your work, the introduced optimizations and the potential **remaining problems**.
- The source codes will be commented and the names of various authors of the project have to be mentioned at the beginning of each source file as well as a textual description specifying its purpose.
- The program will have a minimal GUI console. The GUI is not the core of the project, it should nevertheless be simple and user friendly.
- The report must be in **PDF** format and the corresponding source code have to be delivered no later than **January** 3^{rd} **2018 at 6:00pm** by email addressed to `nicolas.gaud@utbm.fr` and `alexandre.lombard@utbm.fr`, (Subject: LO27 Project - Group: STUDENTNAME1UPPERCASE and STUDENTNAME2UPPERCASE) in a ZIP or TAR.GZ archive named in the following way:

LO27_STUDENTNAME1UPPERCASE_STUDENTNAME2UPPERCASE.zip

Any delay or failure to comply with these guidelines will be penalized.

A special attention should be paid on the following points when writing the program: the program runs smoothly without bugs (it's better not to provide a feature rather than provide it if it does not work, negative points), readability and clarity of the code (comments and indentation), complexity and efficiency of proposed algorithms, choice of data structures, modularity of the code (development of small functions, distributed in various files comprising functions consistently and appropriately named).

2 Project goal

The purpose of this project is to implement a centralized and secured transaction registry. This registry has the role to keep the track of monetary exchanges between different actors, and to secure the transactions.

In fewer words, the registry is a list of transactions where a transaction is defined by an amount of money, an emitter and receiver (and some other data). Thus, by iterating over the registry we can compute the exact balance for each user, or display the total amount given by a user to another user.

But there is more: we want to secure these transactions. For this, we will rely on a simple solution: the registry will act as a *trusted third party*. Let's explain this by an example.

A user Recipient (the one who will receive money) wants to sell a car to user Emitter (the one who will send the money). Using standard transaction systems (like a check) this transaction is risky: the Emitter could go with the car with an invalid payment solution (a check with an insufficient balance). The trusted third party can be used to solve the problem.

In this scenario, the Recipient will perform a transaction request in the registry (the trusted third party) which will give him a transaction ID. The Emitter then will confirm the transaction request: once it is done, and if the Emitter has enough money, the amount of the transaction is withdrawn from the balance of the Emitter but isn't yet on the balance of the Recipient. At this point a private transaction key is given to the Emitter. When the Emitter and the Recipient meet together, and if everything is OK, the Emitter will give the private transaction key to the Recipient. The Recipient can check using the registry that this key is valid, if so, it means that the correct amount is available. The transaction is then executed and the amount is credited for the Recipient.

Like a bank check this transaction is secured because the amount of money is withdrawn from the account of the Emitter before the exchange, but the system of private key and the possibility to check instantaneously the validity of this key prevents the creation of a fake.

3 Transaction process

The figure (fig. 1) illustrate the transaction process.

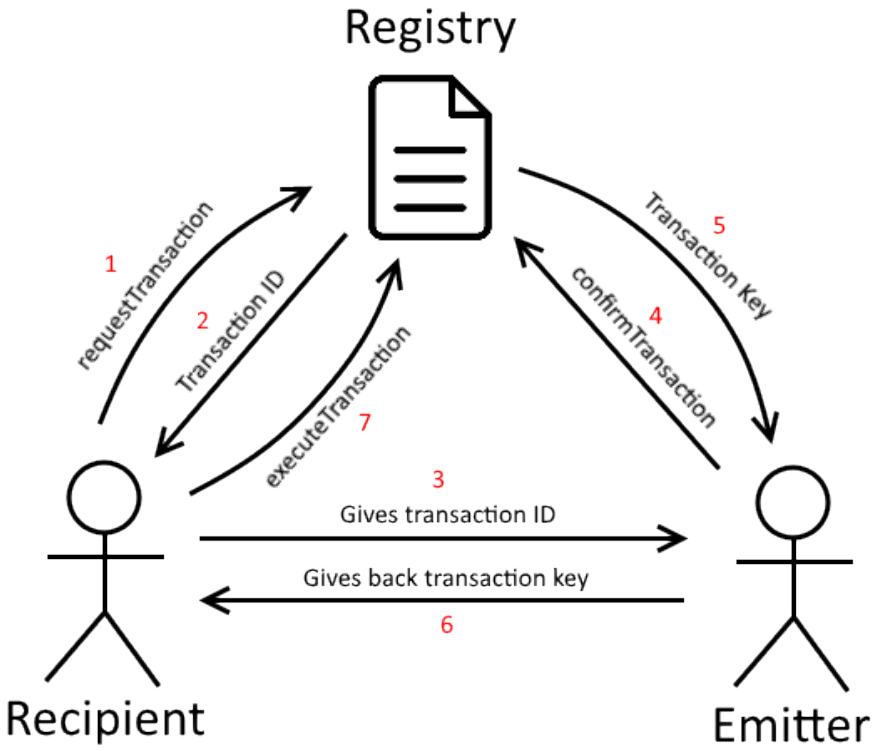

Figure 1: Illustration of the transaction process, the numbers are used to show the order of the operations

4 Abstract data types

4.1 User

A user is defined as $ID \times Name$ with:

- **ID : long**, a unique identifier
- **Name : string**, the name of the user

The type *User* must define the following functions:

- $createUser : Name \rightarrow User$, creates a new user with a randomly generated ID with the specified name
- $getID : User \rightarrow Long$, gets the ID of the user
- $getName : User \rightarrow String$, gets the name of the user
- $setName : User \times String$, specifies the name of the user

4.2 Transaction

A transaction is defined by:

- **ID : long**, a public transaction ID
- **Amount : double**, the amount of money of the transaction (a positive or negative non-null real value), a positive value is a credit for the recipient, a negative one is a debit
- **EmitterID : long**, an emitter ID
- **RecipientID : long**, a recipient ID
- **Status : TransactionStatus**, a status (pending, confirmed, executed or canceled)
- **StartDateTime : long**, the timestamp in seconds of the creation of the transaction
- **EndDateTime : long**, the timestamp in seconds of the end of the transaction
- **TTL: long**, a time-to-live in seconds
- **PrivateKey : string**, a private key used to secure the transactions

When created a transaction is *Pending*. When the emitter confirms it, it is *Confirmed*. Finally when the recipient executes it, it becomes *Executed*. If a transaction is *Pending*, a user knowing the transaction ID can decide to cancel it. The time-to-live defines how long a transaction can be confirmed. For instance if the transaction time is 08:15:00 and if the time-to-live is 00:10:00, then the

transaction cannot be confirmed after 08:25:00. When the transaction is over (either *Executed* or *Cancelled*), the end date time is set.

The type *Transaction* must define at least the following functions:

- $createTransaction \rightarrow Transaction$, creates a new transaction with a randomly generated ID
- $getAmount : Transaction \rightarrow double$, gets the amount of the transaction
- $getStatus : Transaction \rightarrow TransactionStatus$, gets the status of the transaction
- $getStartDateTime : Transaction \rightarrow long$, gets the timestamp of the creation of the transaction
- $getEndDateTime : Transaction \rightarrow long$, gets the timestamp of the end of the transaction
- $getTTL : Transaction \rightarrow long$, gets the time-to-live of the transaction
- $isExpired : Transaction \times long \rightarrow boolean$, returns *true* if the transaction is expired (according to the current date given as parameter), *false* otherwise

4.3 Registry

The data type registry is a doubly linked list of transactions. Each element of the list contains a pointer to the previous element, a pointer to the next element and a transaction. The registry record contains a pointer to the first element of the list, to its last element, and contains also the length of the list, and the last transaction date.

The data type TransactionElement is defined with the following attributes:

- **previous: TransactionElement***, a pointer to the previous element of the list
- **transaction: Transaction**, a transaction
- **next: TransactionElement***, a pointer to the next transaction

The data type registry is defined with the following attributes:

- **head: TransactionElement***, a pointer to the head of the list
- **tail: TransactionElement***, a point to the tail of the list
- **transactionCount: long**, the number of transactions
- **lastTransactionDate: long**, the last transaction date

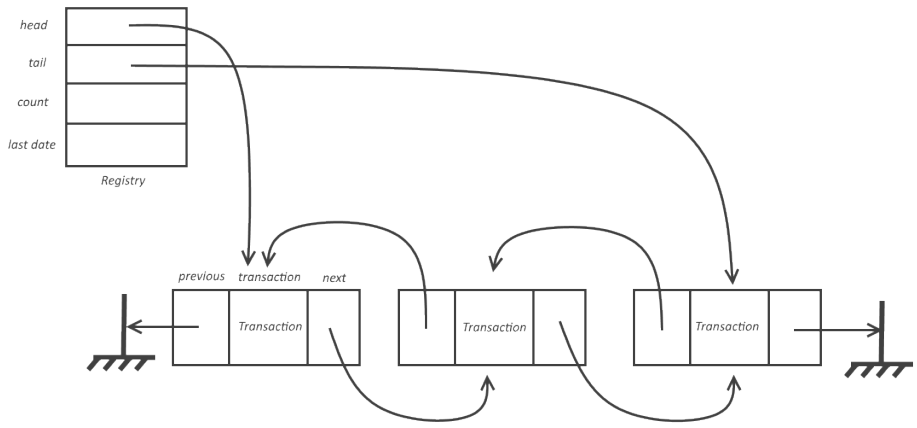

Figure 2: Data structure of TransactionElement and Registry

The figure 2 illustrates the data structure of *TransactionElement* and *Registry*.

The data type registry must at least provide the following functions:
Functions related to an account:

- $getBalance : Registry \times User \rightarrow Double$, gets the current balance for the specified user;

- $getHistory : Registry \times User \rightarrow List[Transaction]$, gets the complete list of the transactions for the specified user;

- $getHistoryLimit : Registry \times User \times Start : Long \times End : Long \rightarrow List[Transaction]$, gets the complete list of the transactions for the specified user between two dates: *Start* and *End*;

- $getAverageCredit : Registry \times User \times Start : Long \times End : Long \rightarrow Double$, gets the average credit of the user between two dates *Start* and *End*;

- $getAverageDebit : Registry \times User \times Start : Long \times End : Long \rightarrow Double$, gets the average debit of the user between two dates *Start* and *End*;

Functions related to a transaction:

- $getAmount : Registry \times Long \rightarrow Long$, gets the amount of the transaction knowing only its ID;

- $getStatus : Registry \times Long \rightarrow TransactionStatus$, gets the status of the transaction knowing only its ID;

Functions related to the registry:

- $load : String \rightarrow Registry$, loads a registry from a file;

- $save : Registry, String$, saves a registry to a file;

- $requestTransaction : Registry \times User \times Double \rightarrow Long$, performs a transaction request with the recipient user and an amount: registers a new transaction in the registry with the status *Pending* and returns the transaction ID, a private transaction key is generated. The request is performed by the recipient.;

- $confirmTransaction : Registry \times User \times Long \rightarrow String$, sets the status to *Confirmed* if the emitter balance allows it. The amount is withdrawn from the emitter balance, but is not on the recipient balance yet. It returns the SHA-256 of the private key (named further the *Transaction Key*). The request is confirmed by the emitter;

- $executeTransaction : Registry \times TransactionId : Long \times TransactionKey : String \rightarrow Boolean$, sets the status to *Executed* if the previous status was *Confirmed* and if the Transaction Key matches the SHA-256 of the private key, if so return *true*, otherwise, return *false*. The EndDateTime is set at this moment. The transaction must be executed by the recipient;

- $cancelTransaction : Registry \times Long \rightarrow Boolean$, sets the status to *Cancelled* if the previous status was *Pending*, return *false* otherwise. Can be called by either the emitter or the recipient;

- $getAverageTransactionTime : Registry \times Start : Long \times End : Long \rightarrow Double$, returns the average transaction time between two dates *Start* and *End*;

Note: To perform the SHA-256 hash, the libcrypto is used, see `https://wiki.openssl.org/index.php/Libcrypto_API`.

5 Work to achieve

5.1 Library

A library written in C containing the types and functions described above.

This library will be implemented using at least two files in C:

registry.h the header file which contains the prototype of the functions, types, constants and variables provided by the library;

registry.c the associated source file which contains the body of the various proposed functions.

You must also provide:

Makefile makefile to compile the sources, generate libraries and executable. At least, 3 targets must be defined in this Makefile:

- **all** compiles everything, generates the libraries and executable.
- **lib** generates the binary code of the matrix libraries *libRegistry.so*.
- **clean** cleans the tmp files generated during the compilation process and the various binary files

registrymain.c the main program, which contains the core of the program and its associated GUI.

The compilation of the entire project have to be managed thanks to a **Makefile**. The matrix dynamic library may be compiled and produced using the following command line (in the Makefile):
$gcc -Wall -Werror -ansi -pedantic "source files" -o libRegistry.so -shared -fpic

5.2 Main program and user interface

To provide a main program using the previous library and enabling a user to test all of its provided functionalities in a simple and friendly way.

Since your program should enable to simply test each library functions, it must provide a random generation mechanism of the various matrices required for each operation.

This program may be compiled using the following command line (in the Makefile):
$ gcc -Wall -Werror -ansi -pedantic -L"library directory" registrymain.c -o Registry.exe -lRegistry

In running the program, the variable LD_LIBRARY_PATH have to specify the directory containing the *Registry* library.

This program will take as parameter:

- The path of the registry file
- The user ID

At startup it will load the registry file and propose a menu allowing the user to perform the different operations.

We will assume that only one user can open the registry at a time.

Once all operations are done, the registry will be saved.

6 References

Libcrypto API: https://wiki.openssl.org/index.php/Libcrypto_API

SHA-256: https://en.wikipedia.org/wiki/SHA-256